University of Edinburgh
Division of Informatics

Effect Systems for Exceptions and Mutable Variables

4th Year Project Report
Computer Science

Paul Flanagan

June 6, 2003

Abstract: Most programming languages contain constructs producing a computational effect as well as yielding a result. These effects include assigning to mutable variables, raising exceptions and performing input and output. The incorrect implementation and use of these effects introduce a class of errors such as failing to catch an exception or performing I/O to an inappropriate device. An effect system is an augmented type system which contains additional information about computational effects. These effect systems can be used to statically analyse a program. The results of this analysis may be used to ensure the absence of a class of errors related to a particular computational effect or to suggest code optimisations.

In this project an effect system is designed, implemented and evaluated for a single computational effect, raising and handling exceptions. The framework of this system is used to create a system considering a different computational effect, allocating, assigning and dereferencing mutable variables. These two effect systems are then combined to produce a system which can analyse both classes of computational effect.

Acknowledgements

Special thanks to Don Sannella for his time and for the supervision of this project. I would also like to thank family, school mates (Andrew Johnston, Pete Mail, Michael Orr), team mates (Steve Easton, Andrew Gauley, Andrew Harrison, Ray Stronge, Ryan Wilson, Davy Young) and flat mates (Hugo Craig, Dave Jarvie, Jeff Knox, Olly Watt).

Contents

1. Introduction

1.1 Overview

Almost all modern programming languages contain statements producing computational effects as well as possibly yielding a result. Examples of such effects include assigning to mutable variables, performing input and output and non-local transfer of control. Often the primary objective when a programmer uses a statement with an associated effect is that this effect occurs. For example the main aim of a programmer using a print statement is the effect, printing to a specified device. In such cases any returned result is usually discarded or used as a check to ensure the desired effect occurred. The correct implementation and use of effects may be of critical importance in ensuring a program behaves as expected.

Type systems classify program expressions into different types. Possible types include the primitive types of the language, polymorphic types, constructed types and function types. A type system statically checks that the type of each program phrase is within the allowable type constraints at this point in the program. This checking is static as oppose to the dynamic runtime behaviour of the system. It therefore provides a tractable syntactic method for proving that a program does not violate the type constraints of a program. For example a string expression cannot be stored in an integer variable or a character expression cannot be used as a parameter to a function expecting an integer. Type systems are often included in compilers to catch errors such as these.

The behaviour of effects can also be statically analysed. Although a type system ensures that the type of a value stored in a mutable variable is correct it does not normally say anything about the possible effects of this assignment. An effect system is an augmented type system which provides additional information about effects. Clearly the class of effect considered by the system determines the information which can be ascertained by any analysis. An exception effect system may determine whether the program contains any uncaught exceptions. An I/O effect system could determine whether a certain device was read from or written to. Effect systems may therefore be used to detect and/or prevent a different class of errors than the typing errors described earlier. Like type systems, effect systems may be included in compilers to discover such errors. Effect systems (and type systems) may also be of use in code optimisation. For example if a program does not alter the store then perhaps a local and more efficent representation of the hard disk can be used. Such information may be returned by an effect system considering assignment to mutable variables.

The class of effect considered by an effect system determines the information which needs to be stored and manipulated by this system. An assignment effect system must represent the store or mutable variables whereas an I/O effect system must examine devices on which I/O may be performed. The complexity of the system also has ramifications on the applicability of any returned information. An exception effect system could just indicate that any pre-declared or user defined exception may raise out of any program phrase. However this does not add much value to our understanding of the meaning of a program phrase.

The effects themselves may be quite complicated. For example performing I/O to one device may cause another form of I/O to occur which may be of interest to our system. Exceptions can be caught and then cause other exceptions to be raised. Assigning to one mutable variable may cause the store associated with another mutable variable to be changed. It therefore may be necessary to consider only a subset of possible effects or a single class of effects when initially designing an effect system. For example considering only the effect of raising and handling an exception or considering only mutable variables which reference base types. This basic effect system can then be built upon. This is the approach that will be taken in this project.

1.2 Aims and Objectives

The principal goal of this project is to design and implement a simple language with a type system that includes information about effects. This goal will be achieved by accomplishing the following sub-goals:

1. Examine, extend and implement an existing effect system designed by Leroy and Pessaux [5] for a single computational effect.

2. Experiment with the implemented system and analyse the produced results.

3. Use the existing framework for the implemented effect system to design and implement an effect system for a different class of computational effect.

4. Experiment with the effect system for this new class of computational effect and evaluate the produced results.

5. Combine the two implemented effect systems to produce a system which can analyse both classes of computational effect.

6. Experiment with the combined system and again analyse any produced results.

1.3 Project Layout

A brief introduction to the history of effect systems and the theory behind them will be provided in chapter 2.

Chapter 3 contains a description of the extended Leroy and Pessaux type system for the exception computational effect which was implemented. The source language for which a type and effect may be inferred is described and the rules governing the analysis performed by the system are given. Experiments are carried out on the implemented system and the results discussed.

Chapter 4 contains a description of an effect system considering allocation, assignment and dereferencing of mutable variables. The source language and the inference rules (along with any necessary auxiliary rules) for the implemented system are given. Again experiments are carried out and the results discussed.

Chapter 5 contains a description of the effect system which is a combination of the effect systems detailed in chapters 3 and 4. An overview of the system is given and experiments illustrating the product of the amalgamation of the system are performed and evaluated.

Chapter 6 contains the conclusions drawn from this project.

2. Effect Systems

This chapter provides an introduction to the history of and theory behind effect systems.

Many of the key ideas in modern effect systems were introduced by Lucassen and Gifford in their Polymorphic Effect Systems [3] paper. The aim of this paper was "to combine the advantages of functional and imperative programming". The effect system in this paper considered the storage effect, the allocation or altering of the store during the execution of a program.

In the terminology of the paper the "effect system is part of a kinded type system". The three base kinds in the system are the type of a program, the (computational) effect of a program and the region in the store where these effects may occur. All locations in the store are associated with a region.

The system deals with allocation, dereferencing and assignment to the store described by the three kinds of effect `ALLOC`, `READ` and `WRITE` respectively, achieved by the `NEW`, `GET` and `SET` operations respectively. The system distinguishes between `PURE` functions (which do not produce a computational effect) and effect producing functions. Effect producing functions are annotated by their latent effect. The latent effect being the set of computational effects which may occur when the function is applied.

A first class value is a value which may be a parameter to a function, returned by a function or stored in memory. An advantage of the system detailed in this paper compared to earlier effect systems was the ability to use both functions and references as first class values. The annotation of a function with its latent effect gives the system this ability to treat functions as first class values. This is a key trait in analysing functional languages in which this higher-order style of programming is prevalent.

The effect systems considered in this project all annotate function types with the effect of their application and all have the ability to treat functions as first class values.

A key advance in the design of this type system was the ability to mask computational effects not observable outside out of a program or a given expression. This means that the system returns an estimate to the actual effect of the program rather than the complete set of effects which may occur in the program and all its sub-expressions. Note the authors prove the soundness of their system. The system is sound in that the set of possible effects is a conservative approximation to the set of actual effects that may arise during execution of the program.

Effect masking is very important in increasing the accuracy of the analysis returned by an effect system. This concept of effect masking is used in the effect systems investigated in this project to produce better results when pattern matching is applied to a term.

The Lucasen and Gifford system was polymorphic in terms of both the effect and region kinds. Note however that the system distinguishes between the application of a normal function and a polymorphic function and could not easily be extend to a complex implicitly typed functional language such as ML. A more implicit style of typing supporting type polymorphism was introduced in a later paper by Talpin and Jouvelot [6]. In this paper the same kinded type system is used as in the Lucassen and Gifford system. Type polymorphism is a property of the effect systems considered in this project.

In the described papers and in the effect systems investigated in this paper the effect system is itself introduced as a series of inference rules for inferring the type and effect of a language phrase. If the premises to these rules hold then the conclusion to the rules also holds. Other auxillary algorithms may also be provided as part of the system such as unification algorithms necessary for type polymorphism.

Lucassen and Gifford believed their effect system had the following applications:

- Specifying the computational effects that a program may produce in a machine-verifiable way.
- Identifying hard to detect optimisations either for evaluation order (for example if two programs do not have interferring effects they may run concurrently) or for common sub-expression elimination.
- Enforcing computational effect constraints allowing a safe integration of imperative programming features into functional languages.

Talpin and Jouvelot [7] consider another interesting application of effect systems to allow more efficient type generalization to be performed in `let` $x = a_1$ `in` a_2 construct where the type of a_1 may be generalized.

3. Exception Effect System

3.1 Overview

Non-local transfers of control are a class of computational effect. Examples include exceptions, jumps and continuations. In this chapter I will extend and evaluate an effect system for a simple language containing exceptions.

An exception is a means by which a called function can signal to its caller that it was unable to perform its desired task. If an exception is not handled or caught by a function it is propagated up the call stack until a function which handles the exception is found. If the exception is not handled by any function in the call stack (ie. it escapes) the program is terminated. Languages such as Java and ML contain exceptions.

Exceptions provide an alternative to the C style method of error reporting. In C return values have associated meanings indicating whether a function executed successfully or not. Exceptions allow easy propagation of an error to an appropriate place in the code. This can take considerable effort using the C style method of error reporting. Exceptions have the favourable property of allowing easy separation of normal and error handling code.

The Java tutorial defines an exception as "an event that occurs during the execution of a program that disrupts the normal flow of instructions." In Java terminology an exception is thrown when an exception object is created and passed to the java runtime system. Java divides its exceptions into two different classes, checked exceptions and runtime exceptions.

Checked exceptions are analysed by the compiler and must be caught or specified to raise out of any method in which they may be thrown but are not caught. In Java an exception is caught using the `try` {} `catch`() {} language construct. An exception is specified that it may escape out of a method using the `throws` keyword. User defined exceptions are often checked exceptions as are I/O exceptions indicating information such as a file system is full.

Runtime exceptions represent a class of exceptions which are thrown when errors are detected by the runtime system. Examples include a calculation involving division by zero, arithmetic overflow, an array being indexed out of bounds or I/O failing. These may escape in numerous places throughout a program (eg. everytime a division or array access occurs). Therefore the cost of checking for such errors outweigh the benefits of catching or specifing them.

As noted by Leroy and Pessaux [5] the Java method of specifying possibly escap-

ing exceptions is not appropriate for ML due to the "higher-order, polymorphic programming" style that ML promotes. This is illustrated if the `map` function is considered. The `map` function iterates through a list applying a function (it takes as a parameter) to every element in the list. If the Java method of specifying exceptions which may escape was used then the map function would have to specify that all possible exceptions may escape to maintain the generic nature of this function. As stated in the introduction such an effect system would not be of much value in detecting uncaught exceptions.

This chapter will consider an effect system based on a system designed by Leroy and Pessaux [5]. The aim of this effect system is to analyse a simple language based on ML to detect uncaught exceptions. The key advantage of this method of exception analysis is that it does not rely on the programmer to specify exceptions which may raise out of a function. As in ML the program is implicitly typed. The effect of the program is also implicitly typed in this effect system.

The effect system has been implemented, experiments run and analysis carried out as detailed in this chapter.

3.2 The Effect System

This section provides a description of the implemented effect system. A more detailed analysis can be found in the original paper [5].

3.2.1 The Source Language

The source language is based on a small subset of ML. Like ML it allows higher-order functions, raising and handling exceptions and (simplified) pattern matching. The original source language has been extended to include the boolean data type in addition to integers and exceptions. The enlarged language also contains recursive functions, a key trait of ML which was not included in the original language.

Program:	*prog* ::=	*decs a* | *a*	program
Declarations:	*decs* ::=	*dec*, *decs*	multiple declarations
		| *dec*;	single declaration
Declaration:	*dec* ::=	`exception` *C*	constant exception
		| `exception` *D*(*type*)	parameterized exception
Type:	*type* ::=	`int`	integer type
		| `exn`	exception type
		| `bool`	boolean type

		$\mid type \rightarrow type$	function type
Term:	$a ::=$	x	identifier
		$\mid i$	integer constant
		$\mid$ `true` $\mid$ `false`	boolean constants
		$\mid \lambda x.a$	abstraction
		$\mid a_1(a_2)$	application
		$\mid$ `let` $x = a_1$ `in` a_2	the `let` binding
		$\mid$ `match` a_1 `with` $pats \mid x \rightarrow a_2$	pattern-matching
		$\mid C \mid D(a)$	exception constructors
		$\mid$ `try` a_1 `with` $x \rightarrow a_2$	exception handler
		$\mid$ `if` a_1 `then` a_2 `else` a_3	selection operation
		$\mid a_1 + a_2 \mid a_1 - a_2$	arithmetic operators
		$\mid a_1 * a_2 \mid a_1/a_2$	
		$\mid a_1$ `orelse` a_2	logical operators
		$\mid a_1$ `andalso` a_2	
		$\mid$ `not`(a_1)	
		$\mid a_1 = a_2 \mid a_1 <> a_2$	comparison operators
		$\mid a_1 > a_2 \mid a_1 < a_2$	
		$\mid a_1 >= a_2 \mid a_1 <= a_2$	
Patterns:	$pats ::=$	$p \rightarrow a \mid pats$	multi-case pattern
		$\mid p$	single case pattern
Pattern:	$p ::=$	x	variable pattern
		$\mid i \mid C$	constant patterns
		$\mid D(p)$	constructed pattern

The original source language did not permit the language user to declare their own exceptions but instead assumed a number of predeclared constant and parameterized exceptions. I have extended the language to allow the user to declare their own exceptions. Predeclared exceptions still exist, they can be considered to be analogous to the built in exceptions in ML such as `Div` and `SysErr`. A comma separated list of user defined exceptions is optionally included at the start of any program. The type of the argument to any parameterized exceptions must be specified.

The `let` $x = a_1$ `in` a_2 construct implements the `let` binding. As expected, any previous values bound to the idenifier x are overidden by a_1. Unlike in the effect system described in the original paper recursive functions are allowed in the augmented effect system.

The `match` a_1 `with` $pats \mid x \rightarrow a_2$ construct performs multi-case (or single-case) pattern matching on the value a_1. If no match is found then a_2 is evaluated. This is an extension of the original system which required the cascading of `match` constructs to implement multi-case pattern matching.

Exceptions are raised using the predefined function `raise`. The `try` a_1 `with` $x \rightarrow a_2$ construct catches any exception and binds it to x. Catching a specific exception is performed using a combination of the `try` and `match` constructs as shown.

```
exception C;
try raise (C)
with x → match x with C → 1 | y → raise (y)
```

The `if` a_1 `then` a_2 `else` a_3 construct has been added to the system in order to carry out more interesting experiments. Arithmetic, logical and comparison operators are also part of the augmented language. These operators, like `raise`, are implemented using predefined functions in the initial environment. This implementation is however hidden (by the lexer/parser and the `BOP` typing rule) from the language user meaning the operators can be used infix as shown in the above source language.

3.2.2 The Type Algebra

The type system is based upon the following type algebra.

Type Expressions:	$\tau ::=$	α	type variable
		$\mid$ `int`$[\varphi]$	integer type
		$\mid$ `exn`$[\varphi]$	exception type
		$\mid$ `bool`	boolean type
		$\mid \tau_1 \xrightarrow{\varphi} \tau_2$	function type
Type Schemes:	$\sigma ::=$	$\forall \alpha_i, \rho_j, \delta_k.\tau$	
Rows:	$\varphi ::=$	$\rho :: K(\rho)$	row variable and kind
		$\mid \top$	all possible elements
		$\mid \varepsilon; \varphi$	element ε plus what is in φ
Row elements:	$\varepsilon ::=$	$i : \pi$	integer constant
		$\mid C : \pi$	constant exception
		$\mid D(\tau)$	parameterized exception
Presence annotations:	$\pi ::=$	`Pre`	element is present
		$\mid \delta$	presence variable
Kinds:	$K ::=$	`INT`(*set of integers*)	integer kind
		$\mid$ `EXN`(*set of exceptions*)	exception kind

There are three base types; the integer type, the exception type and the boolean type as oppose to two in the original system (the integer type and the exception type).

Integer (and exception) types are annotated by the set of integers (or exceptions) that an expression of type `int`$[\varphi]$ (or `exn`$[\varphi]$) may evaluate to. These sets of

integers or exceptions are represented by rows made up of a sequence of row elements and terminated by a row variable. A row element may represent a constant or parameterised exception or an integer. A row can contain either a sequence of integer or exception row elements but not a mixture. The order of row elements in a row is unimportant. Constant exceptions (as opposed to parameterized exceptions) and integers have an associated presence annotation. An associated presence annotation of `Pre` indicates the element is in the set and an associated presence variable means the element is not in the set. However an element with an associated presence variable may be considered present if necessary during unification. The effectively [1] infinite set of integers may be represented by $\top$ when appropriate, no analogous representation exists for the finite set of predeclared and user defined exceptions. $\top$ is absorbing in that $i : \texttt{Pre}; \top = \top$.

The boolean type is an addition to the original system. For ease of implementation the boolean type was not annotated.

The function type, as in effect systems, is labelled with the effect of the application of this function. In this system the effect is the escaping of an unhandled exception.

3.2.3 The Kinding Rules

The kinding rules can be separated into two classes; those ensuring correct kinding and those ensuring an inferred type is well formed (and of the correct kind).

3.2.3.1 Kinding

$$\vdash \rho :: K(\rho) \qquad \vdash \top :: \texttt{INT}(S) \qquad \frac{i \notin S \qquad \vdash \varphi :: \texttt{INT}(S \cup \{i\})}{\vdash (i : \pi; \varphi) :: \texttt{INT}(S)}$$

$$\frac{C \notin S \qquad \vdash \varphi :: \texttt{EXN}(S \cup \{C\})}{\vdash (C : \pi; \varphi) :: \texttt{EXN}(S)} \qquad \frac{D \notin S \qquad \vdash \varphi :: \texttt{EXN}(S \cup \{D\}) \qquad \vdash \tau \ \texttt{wf}}{\vdash (D(\tau); \varphi) :: \texttt{EXN}(S)}$$

3.2.3.2 Well-formedness

$$\vdash \alpha \ \texttt{wf} \qquad \vdash \texttt{bool wf} \qquad \frac{\vdash \varphi :: \texttt{INT}(\emptyset)}{\vdash \texttt{int}[\varphi] \ \texttt{wf}} \qquad \frac{\vdash \varphi :: \texttt{EXN}(\emptyset)}{\vdash \texttt{exn}[\varphi] \ \texttt{wf}}$$

[1] Constrained by the representation of an integer in the language implementing the effect system

$$\frac{\vdash \tau_1\ \mathtt{wf} \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset) \qquad \vdash \tau_2\ \mathtt{wf}}{\vdash \tau_1 \xrightarrow{\varphi} \tau_2\ \mathtt{wf}}$$

In order to ensure the existence of prinipal unifiers and principal typings and to simplify the type inference rules the following structural invariants must hold on a row:

1. A given integer constant or exception constructor should occur at most once in a row.
2. A row variable ρ is preceded by the same set of integer constants and exception constructors in all row expressions where it occurs.
3. A row φ annotating an integer type $\mathtt{int}[\varphi]$ can only contain integer row elements.
4. A row φ annotating an exception type $\mathtt{exn}[\varphi]$ or a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ can contain only constant or parameterized exception row elements and must not end with $\top$.

(1) and (2) are necessary invariants for unification. (3) and (4) ensure a separation between the annotations on integer and exception types. Clearly as $\top$ is only absorbing for integer types it should only be present in a row annotating an integer type and therefore should not terminate a row containing exception row elements.

Kinds are used to enforce these invariants. The constants and constructors appearing in the set part of a kind ($\mathtt{INT}\{i_1, \ldots, i_n\}$ or $\mathtt{EXN}\{C_1, \ldots, C_n, D_1, \ldots, D_n\}$) cannot appear in a row of this kind. These constants and constructors have already appeared in row elements concatenated before this row. For example if $i : \pi; \varphi$ has kind $\mathtt{INT}\{S\}$ (where $i \notin S$) then the row φ has kind $\mathtt{INT}\{S \cup \{i\}\}$.

3.2.4 The Typing Rules

3.2.4.1 Initial Environment

The initial typing environment E_0 from the original effect system has been extended to include predefined functions which will implement the arithmetic, logical and comparison operators.

$$\begin{array}{lll}
E_0 = & \{\mathtt{raise}: & \forall\alpha, \rho.\mathtt{exn}[\rho] \xrightarrow{\rho} \alpha, \\
& +, -, * : & \forall\rho_1\rho_2\rho_3\rho_4.\mathtt{int}[\rho_1] \xrightarrow{\rho_2} \mathtt{int}[\rho_3] \xrightarrow{\rho_4} \mathtt{int}[\top], \\
& / : & \forall\rho_1\rho_2\rho_3\rho_4.\mathtt{int}[\rho_1] \xrightarrow{\rho_2} \mathtt{int}[\rho_3] \xrightarrow{\mathtt{Div:Pre};\rho_4} \mathtt{int}[\top], \\
& \mathtt{orelse}, \mathtt{andalso} : & \forall\rho_1\rho_2.\mathtt{bool} \xrightarrow{\rho_1} \mathtt{bool} \xrightarrow{\rho_2} \mathtt{bool},
\end{array}$$

$$\begin{array}{ll} \texttt{not}: & \forall\rho_1.\texttt{bool} \xrightarrow{\rho_1} \texttt{bool}, \\ =, <>, <, >, <=, >=: & \forall\rho_1\rho_2\rho_3\rho_4.\texttt{int}[\rho_1] \xrightarrow{\rho_2} \texttt{int}[\rho_3] \xrightarrow{\rho_4} \texttt{bool}\} \end{array}$$

A number of predeclared exceptions have also been included in the system.

3.2.4.2 Typing of Expressions

The typing rules for the system follow.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}\ \text{ID} \qquad \frac{\vdash \varphi' :: \texttt{INT}(\{i\}) \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash i : \texttt{int}[i : \texttt{Pre}; \varphi']/\varphi}\ \text{INT}$$

$$\frac{\vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash \texttt{true} : \texttt{bool}/\varphi}\ \text{TRUE} \qquad \frac{\vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash \texttt{false} : \texttt{bool}/\varphi}\ \text{FALSE}$$

$$\frac{E \vdash \texttt{not}\ (a) : \tau/\varphi}{E \vdash \texttt{not}\ a : \tau/\varphi}\ \text{NOT} \qquad \frac{E \vdash \circ\ (a_1)\ (a_2) : \tau/\varphi}{E \vdash a_1 \circ a_2 : \tau/\varphi}\ \text{BOP}$$

$$\frac{\vdash \tau_1\ \texttt{wf} \qquad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}\ \text{ABS}$$

$$\frac{E \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \qquad E \vdash a_2 : \tau'/\varphi}{E \vdash a_1(a_2) : \tau/\varphi}\ \text{APP}$$

$$\frac{E \oplus \{x : \texttt{Gen}(\tau_1, E, \varphi)\} \vdash a_1 : \tau_1/\varphi \qquad E \oplus \{x : \texttt{Gen}(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau/\varphi}\ \text{LET}$$

$$\frac{\begin{array}{c} E \vdash a_1 : \tau_1/\varphi \qquad \vdash\ p : \tau_1 \Rightarrow E' \\ \vdash \tau_1 - p \rightsquigarrow \tau_2 \qquad E \oplus E' \vdash a_2 : \tau/\varphi \qquad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E \vdash \texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi}\ \text{MATCH}$$

$$\frac{\vdash \varphi' :: \texttt{EXN}(\{C\}) \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash C : \texttt{exn}[C : \texttt{Pre}; \varphi']/\varphi}\ \text{EXC}$$

$$\frac{\tau \leq \mathit{TypeArg}(D) \qquad E \vdash a : \tau/\varphi \qquad \vdash \varphi' : \texttt{EXN}(\{D\})}{E \vdash D(a) : \texttt{exn}[D(\tau); \varphi']/\varphi}\ \text{PEXC}$$

$$\frac{E \vdash a_1 : \tau/\varphi' \qquad E \oplus \{x : \texttt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi}{E \vdash \texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2 : \tau/\varphi}\ \text{TRY}$$

$$\frac{E \vdash a_1 : \texttt{bool}/\varphi \qquad E \vdash a_2 : \tau/\varphi \qquad \vdash a_3 : \tau/\varphi}{E \vdash \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 : \tau/\varphi}\ \text{IF}$$

$E_1 \oplus E_2$ is the asymetric concatenation of the two environments E_1 and E_2. $E_1 \oplus E_2\ (x) = E_2\ (x)$ if $x \in Dom(E_2)$ and $E_1 \oplus E_2\ (x) = E_1\ (x)$ if $x \in Dom(E_1) \setminus Dom(E_2)$.

3.2.4.3 Typing of Patterns

$$\vdash x : \tau \Rightarrow \{x : \tau\} \qquad \vdash i : \texttt{int}[i : \pi; \varphi] \Rightarrow \{\}$$

$$\vdash C : \texttt{exn}[C : \pi; \varphi] \Rightarrow \{\} \qquad \frac{\tau \leq \mathit{TypeArg}(D) \qquad \vdash p : \tau \Rightarrow E}{\vdash D(p) : \texttt{exn}[D(\tau); \varphi] \Rightarrow E}$$

3.2.4.4 Pattern Subtraction

$$\vdash \texttt{int}[i : \pi; \varphi] - i \rightsquigarrow \texttt{int}[i : \pi'; \varphi] \qquad \vdash \texttt{exn}[C : \pi; \varphi] - C \rightsquigarrow \texttt{exn}[C : \pi'; \varphi]$$

$$\frac{\vdash \tau' \ \texttt{wf}}{\vdash \tau - x \rightsquigarrow \tau'} \qquad \frac{\vdash \tau - p \rightsquigarrow \tau'}{\vdash \texttt{exn}[D(\tau); \varphi] - D(p) \rightsquigarrow \texttt{exn}[D(\tau'); \varphi]}$$

3.2.4.5 Instantiation and Generalization

$\tau' \leq \forall \alpha_i \rho_j \delta_k . \tau$ if and only if there exists τ_i, φ_j, π_k such that $\vdash \tau_i$ `wf` and $\vdash \varphi_j :: K(\rho_j)$ and $\tau' = \tau\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\}$

$\texttt{Gen}(\tau, E, \varphi)$ is $\forall \alpha_i \rho_j \delta_k . \tau$ where $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$

Most of the rules are the standard typing rules for these language constructs with an effect component, φ. The inference rules which cannot raise an effect (`ID`, `INT`, `TRUE`, `FALSE`, `ABS`, `EXC`, `PEXC`) have an effect of component of $\texttt{EXN}(\emptyset)$ which can be used as needed during unfication.

The `LET` rule has been altered from the original system so that a type may be inferred for recursive functions [1].

The `NOT` and `BOP` define how the `APP` inference rule is used so that the language user cannot detect a function is actually being applied when arithmetic, logical and comparison operators are used. These rules are additions to the effect system. The `IF` rule is also not found in the original type system.

The `ABS` rule uses the effect of the body of the abstraction as the effect of the application of the function (which is applied using the `APP` rule).

In the integer rule, `INT`, and the constant exception rule, `EXN`, the value of the expression is recorded in the row annotating the corresponding integer and exception type respectively.

The rules concerning type inference and pattern matching of parameterized excepions make use of the auxiliary function *TypeArg*(D). This function returns the type scheme of D. *TypeArg*$(D) = \forall\rho.\texttt{int}[\rho]$ for an exception D with an integer parameter. *TypeArg*$(D) = \forall\rho.\texttt{exn}[\rho]$ for an exception D with an exception parameter. *TypeArg*$(D) = \texttt{bool}$ for an exception D with a boolean parameter. Function parameterized exceptions are more complex but can be intuitively built from these base types, for example *TypeArg*$(D) = \forall\rho_1\rho_2\rho_3.\texttt{int}[\rho_1] \xrightarrow{\rho_2} \texttt{int}[\rho_3]$ for an exception D with a function parameter mapping an integer to an integer.

Some of the power of this system comes from the accuracy of the pattern matching rule, `MATCH`, for the $\texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3$ construct. If the second alternative a_3 is evaluated we can ascertain that the term a_1 did not match the pattern `p`. Therefore the type of the identifier x cannot be any type matching pattern `p`. The possible types of x can therefore be calculated by subtraction of the pattern from the type of a_1.

3.3 Implementation

The implementation completed is based on the effect system described in the original paper and its appendices. A number of extensions have been made to the original system as noted above. Some stages of the implementation required a slight deviation from the system described in the paper either to overcome bugs or realise these extensions.

3.3.1 Difficulties

A number of errors and ambiguities were uncovered in the original forms of the algorithms during implementation. A lot of the errors were related to the unification variables and the maintenance of a list detailing which unification variables had already been used. An example of such an error is in the auxiliary function `Patsubtr`. In the incorrect version when pattern subtraction is performed on integers and constant exceptions the returned list of used unification variables is incomplete. Depending on how this list is maintained this may lead to the unification algorithm failing or returning incorrect results.

Similarly the `Inst` auxiliary function in the original paper fails to keep track of unification variables. This function is used to instantiate identifiers from the typing environment and in type inference on parameterized exceptions. Again this fault may lead to problems during unification or type inference.

A major source of ambiguity in the original algorithms is in the unification of two rows (of the form $\varepsilon;\varphi$). The original pseudocode for the algorithm states a new

(unbound) row variable must be created which is "not free" in either of the rows being unified. Clearly this is not possible as a variable cannot be bound (ie. not free) if it has not been used in any expression. I decided that as the algorithm did not give a methodology for selecting a bound variable from either of the rows and the fact the new variable was assigned a new type that an unbound variable was actually required for this stage in the unification.

3.3.2 Using the implementation

The implementation is divided into a number of ML source files. The code for unification and type inference are separated into different files. Leroy and Pessaux consider extension of their simple language to the full ML language. In order to realise features of ML such as recursive datatypes this may require a different form of unification. The completed implementation uses term unification however to incorporate these more complex features graph unification may be necessary. The division of the unification and type inference code will aid in any such project extensions. Other key source files include an environment ML source file containing predeclared exceptions, functions and variables and a pretty printer. The environment file allows easy manipulation of the initial environment. The pretty printer transforms the output of the original effect system into a more human friendly form. A parser and lexer have also been implemented using mosmllex and mosmlyacc [2]. The required files are compiled and linked using mosmlc. The effect system is started using a single command which begins the mosml environment. Program analysis can be carried out on a specified file or on user input to the command line.

3.3.3 Possible implementation extensions

The main aim of this project is to design and implement a simple language with a type system that includes information about effects. The implementation produced allows experiments to be carried out and the results to be evaluated. A pretty printer has been implemented as stated as an extension to the basic system. Another relatively simple but useful augmentation would be to improve the error information returned by the system for programs for which a type and effect cannot be inferred.

The system could also be built upon to create a compiler or interpreter for the source language. The results of running a program could be compared to the static analysis of the program produced by the effect system.

3.4 Experimentation

The following section contains experiments illustrating the behaviour and output of the implementation. Analysis of this behaviour and output is also carried out.

3.4.1 Example Typings

In order to verify the correctness of the implementation and its concurrence with the original Leroy and Pessaux paper analysis of a couple of their examples has been carried out.

Consider catching and handling a constant exception (note this example takes advantage of the ability of the programmer to define exceptions).

```
exception C;
try raise (C)
with x → match x with C → 1 | y → raise (y)
```

The type of this program is $\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]$ and the effect is $\texttt{C} : \delta; \rho'$. The pretty printer returns the following output.

```
The program evaluates to the integer 1.
No exception escapes from this program.
```

Consider a more complex iteration of this example [2] involving the `let` construct and treating exceptions as first class values.

```
exception C1, exception C2;
let test = λex. try raise (ex)
  with x →  match x with C1 → 1 | y → raise (y)
in test (C2)
```

The type of this program is $\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]$ and the effect is $\texttt{C1} : \delta; \texttt{C2} : \texttt{Pre}; \rho'$. The pretty printer returns the following output.

```
The program can evaluate to the integer 1.
The exception C2 may escape from this program.
```

Note if the `test` function had been applied to the exception `C1` then the output would have been the same as for the previous example.

[2] In the original example the parameter to the test function was named `exn`. This is no longer possible as `exn` is a keyword used for specifying the type of the parameter to parameterized exceptions.

3.4.2 Pattern Matching

The language has been extended to include multi-case pattern matching (as noted in the Leroy and Pessaux paper this has been implemented using cascading `match` constructs).

```
exception C1, exception C2, exception C3;
let multicase = λex. try raise (ex)
  with x → match x with
  C1 → raise (C1) | C2 → raise (C2) | C3 → raise (C3) | y → raise (y)
in multicase (C1)
```

The type of this program is α and the effect is $\texttt{C3} : \texttt{Pre}; \texttt{C2} : \texttt{Pre}; \texttt{C1} : \texttt{Pre}; \rho$. The pretty printer returns the following output.

```
An exception always escapes from this program.
The exceptions C3, C2 and C1 may escape from this program.
```

As in ML any exception can be pattern matched again both constant and parameterized exceptions.

```
exception C, exception D(int);
match D(1) with C → 1 | y → 2
```

The type of this program is $\texttt{int}[2 : \texttt{Pre}; 1 : \texttt{Pre}; \rho]$ and the effect is ρ'. The pretty printer returns the following output.

```
The program evaluates to the integers 2 or 1.
No exceptions escape from this program.
```

Note that it is not possible to pattern match an integer against an exception or vice versa. No type or effect can be inferred for the following example.

```
exception C;
match C with 1 → 1 | y → 2
```

3.4.3 Handling Exceptions

Exceptions are handled by the `try` a_1 `with` $x \rightarrow a_2$ language construct. Any raised exception is bound to x and a_2 evaluated. This is different from the ML method of handling exceptions. In ML exceptions are handled by the `handle` construct and rather than catching all exceptions, exceptions to be caught are listed by name.

Type inference on the `try` construct is somewhat naive.

`exception C;`
`try 1 with x` $\rightarrow$ `raise (C)`

The type of this program is `int`$[1 : \texttt{Pre}; \rho]$ and the effect is `C` $: \texttt{Pre}; \rho'$. The pretty printer returns the following output.

`The program can evaluate to the integer 1.`
`The exception C may escape from this program.`

Clearly the program phrase 1 will never raise an exception and so `raise (C)` should never be evaluated. The effect of the program phrase 1 is ρ indicating that no exception can escape. However the above typing results as the a_2 (from the `try` construct) is always evaluated and its effect is the effect of the whole phrase in the `TRY` inference rule. A more powerful system could analyse the effect of the term a_1 (from the `try` construct) and then determine whether or not to evaluate term a_2. A simple but improved implementation may be achieved using the following two type inference rules instead of the original `TRY` type inference rule.

$$\frac{E \vdash a_1 : \tau/\rho}{E \vdash \texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2 : \tau/\rho}\ \text{TRY1}$$

$$\frac{E \vdash a_1 : \tau/\varphi_1 \qquad E \oplus \{x : \texttt{exn}[\varphi_1]\} \vdash a_2 : \tau/\varphi}{E \vdash \texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2 : \tau/\varphi}\ \text{TRY2}$$

These rules do not take into account when exceptions are indicated to be handled. For example a term having effect `Div` $: \delta; \rho$ indicates the `Div` exception may be raised within this term but this is always handled and so `Div` can never raise out of this term. A possible extension to the system would be to build upon these simple rules to implement a system which provides more accurate exception handling by taking into account when exceptions are indicated to be handled.

3.4.4 Functions

Functions are created in this language using the λ language construct. Consider the following example from the original paper.

λ`f.` λ`g.` λ`x. f (g (x))`

The type of this program is $(\alpha \xrightarrow{\rho} \alpha') \xrightarrow{\rho'} (\alpha'' \xrightarrow{\rho} \alpha) \xrightarrow{\rho''} \alpha'' \xrightarrow{\rho} \alpha'$ and the effect is ρ'''. The pretty printer returns the following output.

`The program evaluates to the function type` $(\alpha \xrightarrow{\rho} \alpha') \xrightarrow{\rho'} (\alpha'' \xrightarrow{\rho} \alpha) \xrightarrow{\rho''} \alpha'' \xrightarrow{\rho} \alpha'$.
`No exception escapes from this program.`

The following example is also taken from the original paper.

```
exception D(int);
λn. raise (D(n))
```

The type of this program is $\texttt{int}[\rho] \overset{\texttt{D(int}[\rho]\texttt{)};\rho'}{\rightarrow} \alpha$ and the effect is ρ''. The pretty printer returns the following output.

```
The program evaluates to the function type int[ρ] --D(int[ρ]);ρ'--> α
No exception escapes from this program.
```

The pretty printer does not give information about the possible effect of the application of the functions in the previous examples. This effect is the union of the possible effects of the parameters to the function and the possible effect of the function application. An extension to the project would be to implement a system giving more information about this possible effect. The difficulty arises from the fact that the effect of one parameter to a function may indicate an exception is always handled whereas another parameter may indicate an exception may escape. Determining the union of these effects requires an analysis of how these parameters are used. Also presenting the information, once this union is determined, in a easily readable way is also an issue. Consider the first function in this section. A pretty printer could return the following output.

```
The effect of the application of this function is the union of the
effects of the parameters f, g and x to the function.
```

Although correct this output does not have an intuitively understood meaning.

3.4.5 Predefined Functions

Raising exceptions and arithmetic, logical and comparison operators are all implemented as predefined functions. This is an easy way of enriching the language without introducing a large number of new type inference rules.

```
let x = 1
in let increment = λn. n + 1
  in increment (x) − x
```

The type of this program is $\texttt{int}[\top]$ and the effect is ρ. The pretty printer returns the following output.

```
The program evaluates to any integer.
No exception escapes from this program.
```

The `BOP` rule permits the arithmetic, logical and comparison operators to be used infix. In the implemented system it is not possible to use these predefined func-

tions directly using the APP rule due to constraints on the list of valid identifiers enforced by the lexer/parser.

Note that the type inference algorithm ensures the parameters to these predefined functions are valid. For example type inference on the following program fails.

```
exception C;
C + C
```

3.4.6 Predeclared Exceptions

Although user defined exceptions are now possible in the augmented language, a number of predeclared exceptions still exist in the effect system. One such predeclared exception is the Div exception (analogous to the ML built in exception of the same name). This exception may escape when a division by zero could have occurred.

```
let x = 1
in x/1
```

The type of this program is int[$\top$] and the effect is Div : Pre; ρ. The pretty printer returns the following output.

```
The program can evaluate to any integer.
The exception Div may escape from this program.
```

This example illustrates one of the weaknesses of this system. In that it (correctly) always states that any division may result in a Div exception being raised. If this system was being used to enforce that all exceptions are caught it may be necessary to use the try language construct everytime a division is carried out.

```
let x = 1
in try x/1 with y → x
```

The type of this program is int[1 : Pre; $\top$] and the effect is ρ. The pretty printer returns the following output.

```
The program evaluates to any integer.
No exception escapes from this program.
```

One solution to this problem is to use the java methodology and divide exceptions into runtime exceptions (which do not need to be caught or specified) and checked exceptions which are analysed by the compiler.

3.4.7 Boolean Type

The augmented source language contains the boolean type as well as the integer and exception types from the original system.

```
let x = true
in let double = λy. y + y
  in x andalso double (1) > 1
```

The type of this program is `bool` and the effect is ρ. The pretty printer returns the following output.

```
The program evaluates to a boolean value.
No exception escapes from this program.
```

Parameterized exceptions can have boolean parameters (and function types containing boolean types) as well as integer and exception parameters.

```
exception D(bool);
let x = true
in raise (D(x))
```

The type of this program is α and the effect is $\texttt{D(bool)}; \rho$. The pretty printer returns the following output.

```
An exception always escapes from this program.
The exception D(bool) may escape from this program.
```

3.4.8 Selection Operation

The `if` a_1 `then` a_2 `else` a_3 construct has been added to allow experiments involving selection to be carried out. As expected the term a_1 is constrained to be of boolean type.

```
(* simple function to raise an error if parameter is negative*)
exception Error(int);
let isError = λn. if n < 0 then raise (Error(n)) else n
in isError (1)
```

The type of this program is $\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]$ and the effect is $\texttt{error}(\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]); \rho'$. The pretty printer returns the following output.

```
The program can evaluate to the integer 1
The exception error(1) may escape from this program.
```

3.4.9 Function Parameterized Exceptions

Parameterized exceptions are included in the original Leroy and Pessaux paper however no analysis is carried out of their most interesting feature. Parameterized exceptions can accept functions as their arguments.

```
exception D(int → bool);
try raise (D(let positive = λn. n > 0 in positive))
with x → match x with D(func) → func (1) | y → raise (y)
```

The type of this program is `bool` and the effect is $\mathtt{D}(\alpha);\rho$. The pretty printer returns the following output.

```
The program evaluates to a boolean value.
No exception escapes from this program.
```

This example shows how the function parameter to an exception can be applied to a parameter using pattern matching. Note the binding which occurs in the pattern matching. In the following program the identifier `func` is declared to have a different type than that expected by the exception.

```
exception D(int → bool);
let func = 1
in try raise (D(let positive = λn. n > 0 in positive))
with x → match x with D(func) → func (1) | y → raise (y)
```

The type and effect of this program is the same as for the previous example. Clearly the type of the function parameter is bound to the identifier `func`.

Note more complicated examples are possible involving higher order functions.

```
exception D(int → bool);
let isTheSame = λn1. λn2. n1 = n2
in try raise (D(isTheSame (1)))
with x → match x withD(func) → func (1) | y → raise (y)
```

Again the same type and effect is returned.

3.4.10 Recursive Functions

The original effect system has been augmented so that it may infer types for programs involving recursive functions. The standard recursive example and the corresponding type and effect given by the system follows.

```
let factorial = λn. if n = 0 then 1 else n ∗ fact(n − 1)
in factorial(5)
```

The type of this program is int[⊤] and the effect is ρ. The pretty printer returns the following output.

```
The program evaluates to any integer.
No exception escapes from this program.
```

3.4.11 Complex Example

The following example is an adaptation of a non-trivial function involving exceptions from an ML textbook [4]. The function change (x) returns the number of ways of changing the monetary value x using the coins 1, 2, 5, 10, 20, 50. The example involves a large number of the language constructs working together.

```
exception Change, exception InvalidDenomination;
let denomination =
λn. match n
  with 1 → 1 | 2 → 2 | 3 → 5 | 4 → 10 | 5 → 20 | 6 → 50
  | y → raise(InvalidDenomination)
in let count =
  λamount. λkindsLeft.
  if amount < 0 orelse kindsLeft = 0
  then raise (Change)
  else
    if amount = 0 then 1 else
    try count (amount − denomination (kindsLeft)) (kindsLeft)
      with x → match x with Change → 0 | y → raise (y)
    +
    try count (amount) (kindsLeft − 1)
      with x → match x with Change → 0 | y → raise (y)
  in let change =
    λamount. count (amount) (6)
    in change (100)
```

The type of this program is int[⊤] and the effect is
Change : Pre; InvalidDenomination : Pre; ρ. The pretty printer returns the following output.

```
The program evaluates to any integer.
The exceptions Change and InvalidDenomination may escape from this program.
```

The fact that the system infers InvalidDenomination may escape shows how it can detect the propagation of exceptions up the call graph. Change may also escape from this program if the amount parameter to count is negative or the kindsLeft parameter is 0. It is not possible to handle the Change exception

within the `count` function as the raising of this exception is how the function detects that it needs to use a smaller denomination of coin (ie. how backtracking is implemented). It would be possible to ensure that Change never escapes using the `try` language construct and placing it around the application of the `count` function as shown.

```
exception Change, exception InvalidDenomination,
exception Error;
let denomination =
λn. match n
  with 1 → 1 | 2 → 2 | 3 → 5 | 4 → 10 | 5 → 20 | 6 → 50
  | y → raise(InvalidDenomination)
in let count =
  λamount. λkindsLeft.
  if amount < 0 orelse kindsLeft = 0
  then raise (Change)
  else
    if amount = 0 then 1 else
    try count (amount − denomination (kindsLeft)) (kindsLeft)
      with x → match x with Change → 0 | y → raise (y)
    +
    try count (amount) (kindsLeft − 1)
      with x → match x with Change → 0 | y → raise (y)
  in let change =
    λamount. try count (amount) (6)
      with x → match x with Change → raise (Error) | y → raise (y)
    in change (100)
```

The type of this program is $\texttt{int}[\top]$ and the effect is
$\texttt{Change} : \texttt{Delta}; \texttt{InvalidDenomination} : \texttt{Pre}; \texttt{Error} : \texttt{Pre}; \rho$. The pretty printer returns the following output.

```
The program can evaluate to any integer.
The exceptions InvalidDenomination and Error may escape from this program.
```

3.5 Conclusion

The examples from this chapter illustrate the power of the implemented effect system. Much of the power of this system comes from the use of rows to annotate the integer and exception base types. These rows indicate the possible values a term of integer or exception type may have. This allows more precise pattern matching and exception handling to be carried out than if unannotated types

where used.

In this system the boolean type was unannotated. This was a design decision in order to reduce the complexity of the necessary implementation. Due to the small size of the boolean set ($\{\texttt{true}, \texttt{false}\}$) if the boolean type had been annotated this may have allowed some powerful inference rules to be implemented for the selection operation and logical operators. This would be a relatively simple extension to the implementation but will not be considered in this project.

In the following chapter a system will be implemented for another type of computational effect using the ideas behind this exception effect system.

4. Storage Effect System

4.1 Overview

In this chapter we consider the "storage effect", the allocation or mutation of storage during the execution of a program. ML contains both mutable variables and immutable variables. A mutable variable is one whose value may be changed whereas the value of an immutable variable cannot. Each mutable variable has an associated mutable cell or container within the store which contains data values of a fixed type. During the evaluation of a program the contents of this cell may be altered or retrieved. This is in contrast to immutable variables which have an unchanging associated value. Note both mutable variables and immutable variables have a fixed type.

Ordinary (immutable) ML variables are declared with an associated fixed value using the ML `val` keyword. For example, `val pi = 3.145926`; declares a variable `pi` (of type `real`) to have value `3.145926`. The value of `pi` cannot be altered. However a variable of the same name with a different value can be created but again this new variable will be immutable. Note that the expression `pi = 3.0` evaluates to `false`, the result of the equality comparison operation between `3.145926` and `3.0`.

Mutable ML variables are known as references in ML and are declared using the ML `val` and `ref` keywords. For example, `val piref = ref 3.145926`; declares a variable `piref` (of type `real ref`) to have value `ref 3.145926`;. To use the contents of the mutable cell associated with `piref` in an expression the variable must be explicitly dereferenced using the ! operator. `!piref` has value `3.145926` and type `real`. Altering the contents of the mutable cell associated with a mutable variable is achieved with the := operator. For example to alter the value of `piref` to `3.0`, `piref := 3.0`.

Mutable variables do not necessarily have to reference values of a base type. In languages such as ML it is valid for a mutable variable to reference a value which has a function type, a constructed type or even another reference type. This introduces the problem of aliasing where two mutable variables are bound to the same mutable cell. Assigning to one of these variables will therefore alter the referenced value of the other.

An effect system analysing the behaviour of mutable variables is of interest for a number of reasons. Such an effect system could be used for optimisation by a compiler if, for example, a mutable variable is created but the associated store is never altered. An effect system may also be used to determine what parts of the

store are altered by a program. The ability of the effect system to perform these tasks is dependent on how the effect is represented. This representation may be based on either the variables referencing the store or the parts of the store being referenced or perhaps some combination of both.

In this chapter I will consider a storage effect system based on the variables referencing the store. This effect system is based on the framework developed in the exception effect system. As in ML the program is implicitly typed. The effect of the program is also implicitly typed in this effect system.

The effect system has been implemented, experiments run and analysis carried out as detailed in this chapter.

4.2 The Effect System

This section provides a description of the implemented effect system.

4.2.1 The Source Language

The source language is based on a small subset of ML. Like ML it allows mutable variable allocation, dereferencing and assignment. As in the exception effect system it also allows higher-order functions, recursive functions and (simplified) pattern matching.

Program:	$prog$ ::=	$decs\ a \mid a$	program
Declarations:	$decs$ ::=	$dec,\ decs$	multiple declarations
		$\mid dec;$	single declaration
Declaration:	dec ::=	`val` $x =$ `ref` a_1	mutable variable
Term:	a ::=	x	identifier
		$\mid i$	integer constant
		$\mid\ !(a_1)$	dereference
		$\mid a_1 := a_2$	assignment
		$\mid \lambda x.a$	abstraction
		$\mid a_1(a_2)$	application
		$\mid$ `let` $x = a_1$ `in` a_2	the `let` binding
		$\mid$ `match` a_1 `with` $pats \mid x \rightarrow a_2$	pattern-matching
		$\mid$ `if` a_1 `then` a_2 `else` a_3	selection operation
		$\mid\ a_1 + a_2 \mid a_1 - a_2$	arithmetic operators
		$\mid\ a_1 * a_2 \mid a_1/a_2$	
		$\mid\ a_1$ `orelse` a_2	logical operators
		$\mid\ a_1$ `andalso` a_2	

		$\mid$ `not`(a_1)	
		$\mid$ $a_1 = a_2 \mid a_1 <> a_2$	comparison operators
		$\mid$ $a_1 > a_2 \mid a_1 < a_2$	
		$\mid$ $a_1 >= a_2 \mid a_1 <= a_2$	
Patterns:	$pats ::=$	$p \rightarrow a \mid pats$	multi-case pattern
		$\mid p$	single case pattern
Pattern:	$p ::=$	x	variable pattern
		$\mid i$	constant pattern

Mutable variables are allocated at the start of a program using the `val` $x =$ `ref` a_1 language construct. Mutable variables have global scope. Dereferencing and assignment to a mutable variable is achieved, as in ML, using the ! and := operators respectively. Obviously the system will not allow an identifier to be assigned to or dereferenced if a mutable variable of this name has not been allocated.

The `let` $x = a_1$ `in` a_2 construct from the exception effect system is also included in the source language. As in the implemented exception effect system recursive functions are permitted in this language.

The `match` a_1 `with` $pats \mid x \rightarrow a_2$ construct has also transferred to this system from the exception effect system upon which this effect system is based. The set of permitted patterns has been reduced as exceptions are not part of this language and therefore exception patterns should not be part of this language.

To allow more interesting experiments to be carried out the `if` a_1 `then` a_2 `else` a_3 construct has been included in this language. Arithmetic, logical and comparison operators are also in the source language.

4.2.2 The Type Algebra

The type system is based upon the following type algebra.

Type Expressions:	$\tau ::=$	α	type variable
		$\mid$ `int`$[\varphi]$	integer type
		$\mid$ `ref`$[\varphi]$	reference type
		$\mid$ `bool`	boolean type
		$\mid$ `unit`	unit type
		$\mid \tau_1 \xrightarrow{\varphi} \tau_2$	function type
Type Schemes:	$\sigma ::=$	$\forall \alpha_i, \rho_j, \delta_k . \tau$	
Rows:	$\varphi ::=$	$\rho :: K(\rho)$	row variable and kind
		$\mid \top$	all possible elements
		$\mid \varepsilon ; \varphi$	element ε plus what is in φ

Row elements:	$\varepsilon ::=$	$i : \pi$	integer constant
		$\mid x : \tau$	mutable variable
Presence annotations:	$\pi ::=$	`Pre`	element is present
		$\mid \delta$	presence variable
Kinds:	$K ::=$	`INT`(*integer list*)	integer kind
		$\mid$ `VAR`(*variable list*)	variable kind

This system contains four base types; the integer type, the reference type, the boolean type and the unit type.

As in the exception effect system integer types are annotated with a row indicating the possible values that an expression of type $\texttt{int}[\varphi]$ may evaluate to. The reference type is also annotated with a row indicating the possible mutable variables that an expression of type $\texttt{ref}[\varphi]$ can evaluate to. As in the previous effect system a row is made up of a sequence of row elements and terminated by a row variable. However the set of possible row elements has changed. A row element may now represent an integer or a mutable variable. A row can contain either a sequence of integers or mutable variables but not a mixture. The mutable variable row element has an associated type indicating the possible values that may be in the mutable cell referenced by the mutable variable. Again $\top$ is used to represent the effectively infinite set of integers.

For ease of implementation the boolean type remains unannotated. The unit type is also unannoted as, like in ML, there is no value associated with this type.

The function type is labelled with a row indicating the set of mutable variables which may be assigned to by this function.

4.2.3 The Kinding Rules

The kinding rules ensure correct kinding and well-formedness.

4.2.3.1 Kinding

$$\vdash \rho :: K(\rho) \qquad \vdash \top :: \texttt{INT}(S) \qquad \frac{i \notin S \qquad \vdash \varphi :: \texttt{INT}(S \cup \{i\})}{\vdash (i : \pi; \varphi) :: \texttt{INT}(S)}$$

$$\frac{x \notin S \qquad \vdash \varphi :: \texttt{VAR}(S \cup \{x\}) \qquad \vdash \tau \ \texttt{wf}}{\vdash (x : \tau; \varphi) :: \texttt{VAR}(S)}$$

4.2.3.2 Well-formedness

$$\vdash \alpha\ \texttt{wf} \qquad \vdash \texttt{bool}\ \texttt{wf} \qquad \vdash \texttt{unit}\ \texttt{wf}$$

$$\frac{\vdash \varphi :: \texttt{INT}(\emptyset)}{\vdash \texttt{int}[\varphi]\ \texttt{wf}} \qquad \frac{\vdash \varphi :: \texttt{VAR}(\emptyset)}{\vdash \texttt{var}[\varphi]\ \texttt{wf}}$$

$$\frac{\vdash \tau_1\ \texttt{wf} \qquad \vdash \varphi :: \texttt{VAR}(\emptyset) \qquad \vdash \tau_2\ \texttt{wf}}{\vdash \tau_1 \xrightarrow{\varphi} \tau_2\ \texttt{wf}}$$

A similar set of structural invariants to those in the exception effect system must hold on a row.

1. A given integer constant or mutable variable constructor should occur at most once in a row.
2. A row variable ρ is preceded by the same set of integer constants or mutable variable constructors in all row expressions where it occurs.
3. A row φ annotating an integer type $\texttt{int}[\varphi]$ can only contain integer row elements.
4. A row φ annotating a reference type $\texttt{var}[\varphi]$ or a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ can contain only mutable variable row elements and must not end with $\top$.

These rules ensure the existence of principal unifiers and principal typings. They ensure separation between the annotations of integer types and reference types.

4.2.4 The Typing Rules

4.2.4.1 Initial Environment

The initial typing environment E_0 includes predefined functions which will implement the arithmetic, logical and comparison operators.

$$\begin{array}{lll}
E_0 = & \{\ +, -, * : & \forall\rho_1\rho_2\rho_3\rho_4.\texttt{int}[\rho_1] \xrightarrow{\rho_2} \texttt{int}[\rho_3] \xrightarrow{\rho_4} \texttt{int}[\top], \\
& / : & \forall\rho_1\rho_2\rho_3\rho_4.\texttt{int}[\rho_1] \xrightarrow{\rho_2} \texttt{int}[\rho_3] \xrightarrow{\texttt{Div:Pre};\rho_4} \texttt{int}[\top], \\
& \texttt{orelse}, \texttt{andalso} : & \forall\rho_1\rho_2.\texttt{bool} \xrightarrow{\rho_1} \texttt{bool} \xrightarrow{\rho_2} \texttt{bool}, \\
& \texttt{not} : & \forall\rho_1.\texttt{bool} \xrightarrow{\rho_1} \texttt{bool}, \\
& =, <>, <, >, <=, >= : & \forall\rho_1\rho_2\rho_3\rho_4.\texttt{int}[\rho_1] \xrightarrow{\rho_2} \texttt{int}[\rho_3] \xrightarrow{\rho_4} \texttt{bool}\}
\end{array}$$

Mutable variable allocation is carried out using the $\texttt{val}\ x = \texttt{ref}\ a_1$ language construct. This is achieved by mapping the identifier x to $\texttt{Gen}(\texttt{ref}[x : \tau_1; \rho], E_0, \varphi_1)$ in the inital environment, where τ_1 is the type of a_1, φ_1 the effect of a_1, ρ is a

new row variable of kind $\texttt{VAR}(x)$ and $\texttt{Gen}$ is the generalization auxiliary function.

4.2.4.2 Typing of Expressions

The typing rules for the system follow.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}\ \text{ID} \qquad \frac{\vdash \varphi' :: \texttt{INT}(\{i\}) \qquad \vdash \varphi :: \texttt{VAR}(\emptyset)}{E \vdash i : \texttt{int}[i : \texttt{Pre}; \varphi']/\varphi}\ \text{INT}$$

$$\frac{\vdash \varphi :: \texttt{VAR}(\emptyset)}{E \vdash \texttt{true} : \texttt{bool}/\varphi}\ \text{TRUE} \qquad \frac{\vdash \varphi :: \texttt{VAR}(\emptyset)}{E \vdash \texttt{false} : \texttt{bool}/\varphi}\ \text{FALSE}$$

$$\frac{E \vdash \texttt{not}\ (a) : \tau/\varphi}{E \vdash \texttt{not}\ a : \tau/\varphi}\ \text{NOT} \qquad \frac{E \vdash \circ\ (a_1)\ (a_2) : \tau/\varphi}{E \vdash a_1 \circ a_2 : \tau/\varphi}\ \text{BOP}$$

$$\frac{E \vdash a_1 : \texttt{ref}[\varphi']/\varphi}{E \vdash !a_1 : \texttt{ConcatenateRefs}(\varphi')/\varphi}\ \text{DEREF}$$

$$\frac{E \vdash a_1 : \texttt{ref}[\varphi']/\varphi \qquad E \vdash a_2 : \tau/\varphi \qquad E \vdash \varphi'' = \texttt{UpdateRefs}(\tau, \varphi')}{E \vdash a_1 := a_2 : \texttt{unit}/\varphi''}\ \text{ASSIGN}$$

$$\frac{\vdash \tau_1\ \texttt{wf} \qquad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}\ \text{ABS}$$

$$\frac{E \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \qquad E \vdash a_2 : \tau'/\varphi}{E \vdash a_1(a_2) : \tau/\varphi}\ \text{APP}$$

$$\frac{E \oplus \{x : \texttt{Gen}(\tau_1, E, \varphi)\} \vdash a_1 : \tau_1/\varphi \qquad E \oplus \{x : \texttt{Gen}(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau/\varphi}\ \text{LET}$$

$$\frac{\begin{array}{c} E \vdash a_1 : \tau_1/\varphi \qquad \vdash\ p : \tau_1 \Rightarrow E' \\ \vdash \tau_1 - p \rightsquigarrow \tau_2 \qquad E \oplus E' \vdash a_2 : \tau/\varphi \qquad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E \vdash \texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi}\ \text{MATCH}$$

$$\frac{E \vdash a_1 : \texttt{bool}/\varphi \qquad E \vdash a_2 : \tau/\varphi \qquad \vdash a_3 : \tau/\varphi}{E \vdash \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 : \tau/\varphi}\ \text{IF}$$

$E_1 \oplus E_2$ is the asymetric concatenation of the two environments E_1 and E_2. $E_1 \oplus E_2\ (x) = E_2\ (x)$ if $x \in Dom(E_2)$ and $E_1 \oplus E_2\ (x) = E_1\ (x)$ if $x \in Dom(E_1) \setminus Dom(E_2)$.

4.2.4.3 Typing of Patterns

$$\vdash x : \tau \Rightarrow \{x : \tau\} \qquad \vdash i : \texttt{int}[i : \pi; \varphi] \Rightarrow \{\}$$

4.2.4.4 Pattern Subtraction

$$\vdash \texttt{int}[i : \pi; \varphi] - i \rightsquigarrow \texttt{int}[i : \pi'; \varphi] \qquad \frac{\vdash \tau' \texttt{ wf}}{\vdash \tau - x \rightsquigarrow \tau'}$$

4.2.4.5 Instantiation and Generalization

$\tau' \leq \forall \alpha_i \rho_j \delta_k . \tau$ if and only if there exists τ_i, φ_j, π_k such that $\vdash \tau_i$ `wf` and $\vdash \varphi_j :: K(\rho_j)$ and $\tau' = \tau\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\}$

$\texttt{Gen}(\tau, E, \varphi)$ is $\forall \alpha_i \rho_j \delta_k . \tau$ where $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$

4.2.4.6 Reference Operations

These reference operations are necessary as a term may evaluate to more than one mutable variable.

`ConcatenateRefs`(φ) returns the union of all the types in the mutable variable constructors in a row, φ, made up of a sequence of mutable variable row elements. For example if φ is $\texttt{x} : \texttt{int}[\texttt{1} : \texttt{Pre}; \rho]; \texttt{y} : \texttt{int}[\texttt{2} : \texttt{Pre}; \rho']); \rho''$ indicating that the mutable variable `x` is a reference to the integer `1` and the mutable variable `y` is a reference to the integer `2` then `ConcatenateRefs`(φ) returns $\texttt{int}[\texttt{1} : \texttt{Pre}; \texttt{2} : \texttt{Pre}; \rho''']$ the union of the types associated with the mutable variables `x` and `y` (a term of this type may evaluate to the integer `1` or the integer `2`).

`UpdateRefs`(τ, φ) updates the type in each mutable variable constructor in a row, φ, made up of a sequence of mutable variable row elements to indicate that each mutable variable may evaluate to a type of τ. The updated row is returned by the function. If φ is of the form $x : \tau'; \rho$ (ie. contains only one mutable variable constructor) then the row $x : \tau; \rho$ is returned. If φ is of the form $x_1 : \tau_1'; \ldots; x_n : \tau_n'; \rho$ (ie. contains more than one mutable variable constructor) then the row $x_1 : \tau_1' \otimes \tau; \ldots; x_n : \tau_n' \otimes \tau; \rho$ is returned where $\tau' \otimes \tau$ is the union of the two types τ' and τ. For example if τ is $\texttt{int}[\texttt{3} : \texttt{Pre};]$ and φ is $\texttt{x} : \texttt{int}[\texttt{1} : \texttt{Pre}; \rho]; \texttt{y} : \texttt{int}[\texttt{2} : \texttt{Pre}; \rho']); \rho''$ inidicating that the mutable variable `x` is a reference to the integer `1` and the mutable variable `y` is a reference to the integer `2` then `UpdateRefs`(τ, φ) returns $\texttt{x} : \texttt{int}[\texttt{1} : \texttt{Pre}; \texttt{3} : \texttt{Pre}; \rho]; \texttt{y} : \texttt{int}[\texttt{2} : \texttt{Pre}; \texttt{3} : \texttt{Pre}; \rho']); \rho''$ indicating inidicating that the mutable variable `x` is a reference to the integer `1` or `3` and the mutable variable `y` is a reference to the integer `2` or `3`.

Most of the typing rules are identical to the rules for the exception effect system discussed earlier (except using `VAR`($\emptyset$) rather than `EXN`($\emptyset$) as the effect component

of rules which have no effect). The most interesting rules are the new `DEREF` and `ASSIGN` rules.

The `DEREF` rule governs how the system analyses the dereferencing of a term. This uses the auxiliary function `ConcatenateRefs`(φ) which returns the union of all the type annotations in the row φ which is a sequence of mutable variable row elements. This is necessary as the mutable variable being dereferenced may be any of the mutable variables in this row so the type returned must be the union of all the types referenced by these variables.

The `ASSIGN` rule assigns to a mutable variable. However the system may believe that a term may evaluate to multiple mutable variables. Therefore the `UpdateRefs`(τ, φ) is used to update the value associated with each mutable variable which may possibly be assigned to.

4.3 Implementation

The implemented system is based on the implemented exception effect system.

4.3.1 Using the implementation

Much of the code base for the exception effect system was used for the implementation of this storage effect system. The division of the source code is the same in both implementations. Again a parser and lexer have also been implemented using mosmllex and mosmlyacc. The required files are compiled and linked using mosmlc. The effect system is started using a single command which begins the mosml environment. Program analysis can be carried out on a specified file or on user input to the command line.

4.4 Experimentation

The following section contains experiments illustrating the behaviour and output of the implementation. Analysis of this behaviour and output is also carried out.

4.4.1 Simple Typings

The `val` x = `ref` a construct is used to create mutable variables (it must appear at the beginning of a program).

`val x = ref 1; x`

The type of the program is $\texttt{ref}[\texttt{x} : \texttt{int}[\texttt{1} : \texttt{Pre}; \rho]; \rho']$ and the effect is ρ''.

```
The program evaluates to the mutable variable x.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
```

Dereferencing is achieved, as in ML, using the ! operator.

`val x = ref 1; !x`

The type of the program is $\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]$ and the effect is ρ'. The pretty printer returns the following output.

```
The program evaluates to the integer 1.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
```

Again adopting ML syntax, assignment to a mutable variable is accomplished using the := operator.

`val x = ref 1; x := 2`

The type of the program is `unit` and the effect is $\texttt{x} : \texttt{int}[\texttt{2} : \texttt{Pre}; \rho]; \rho'$. The pretty printer returns the following output.

```
The program evaluates to the unit type.
The mutable variable x is altered by this program.
The mutable variable x is a reference to the integer value 2.
```

4.4.2 Dereferencing

Dereferencing becomes more interesting when the term being dereferenced is not a variable. Considering the dereferencing of a function in the following example (note the function being dereferenced is somewhat artificial in that it always returns the mutable variable `x` irrespective of its argument).

```
val x = ref 1;
let func = λn. x
in !(func (2))
```

The type of this program is $\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]$ and the effect is ρ'. The pretty printer returns the following output.

```
The program evaluates to the integer 1.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
```

4.4.3 Assignment

Analysis of assignment, like dereferencing, is more difficult when the term being assigned to is not a variable. For example again consider assignment to the result of a function application (as in the previous example the function being consider is in itself not very interesting).

```
val x = ref 1;
let func = λn. x
in (func (2)) := 3
```

The type of this program is `unit` and the effect is $\texttt{x} : \texttt{int}[3 : \texttt{Pre}; \rho]; \rho'$. The pretty printer returns the following output.

```
The program evaluates to the unit type.
The mutable variable x is altered by this program.
The mutable variable x is a reference to the integer value 3.
```

4.4.4 Selection Operation

The $\texttt{if a}_1\ \texttt{then a}_2\ \texttt{else a}_3$ construct may mean that a term may evaluate to more than one mutable variable.

```
val x = ref 1, val y = ref 2;
if !x >!y then x else y
```

The type of this program is $\texttt{ref}[\texttt{y} : \texttt{int}[2 : \texttt{Pre}; \rho]; \texttt{x} : \texttt{int}[1 : \texttt{Pre}; \rho']; \rho'']$ and the effect is ρ'''. The pretty printer returns the following output.

```
The program evaluates to the mutable variables x or y.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
The mutable variable y is a reference to the integer value 2.
```

Dereferencing must take into account all mutable variables which a term (such as the previous example) may evaluate to. Consider the following example.

```
val x = ref 1, val y = ref 2;
!(if !x >!y then x else y)
```

The type of this program is $\texttt{int}[2 : \texttt{Pre}; 1 : \texttt{Pre}; \rho]$ and the effect is ρ'. The pretty printer returns the following output.

```
The program evaluates to the integer values 1 or 2.
```

```
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
The mutable variable y is a reference to the integer value 2.
```

Assigning to a term evaluating to multiple mutable variables must also consider all possible variables which may be altered.

```
val x = ref 1, val y = ref 2;
(if !x > !y then x else y) := 3
```

The type of this program is `unit` and the effect is $\texttt{y} : \texttt{int}[3 : \texttt{Pre}; 2 : \texttt{Pre}; \rho]; \texttt{x} : \texttt{int}[3 : \texttt{Pre}; 1 : \texttt{Pre}; \rho']; \rho''$. The pretty printer returns the following output.

```
The program evaluates to the unit type.
The mutable variable x or y may be altered by this program.
The mutable variable x is a reference to the integer value 1 or 3.
The mutable variable y is a reference to the integer value 2 or 3.
```

4.4.5 Pattern Matching

Due to the use of similar type representations and algorithms as those in the exception effect system much of the power of this system is also present in this storage effect system. The (simplified) pattern matching construct `match` from the exception effect system has transferred across to this effect system.

```
val x = ref 1;
match !x with 1 → 2 | y → 3
```

The type of this program is $\texttt{int}[3 : \texttt{Pre}; 2 : \texttt{Pre}; \rho]$ and the effect is ρ'. The pretty printer returns the following output.

```
The program evaluates to the integer values 3 or 2.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
```

Note that the `match` language construct (like the `if` construct) can evaluate to multiple mutable variables.

```
val x = ref 1, val y = ref 2;
(match !x with 1 → x | a → y) := 3
```

The type of this program is `unit` and the effect is $\texttt{y} : \texttt{int}[3 : \texttt{Pre}; 2 : \texttt{Pre}; \rho]; \texttt{x} : \texttt{int}[3 : \texttt{Pre}; 1 : \texttt{Pre}; \rho']; \rho''$. The pretty printer returns the following output.

```
The program evaluates to the unit type.
The mutable variable x or y may be altered by this program.
The mutable variable x is a reference to the integer value 1 or 3.
The mutable variable y is a reference to the integer value 2 or 3.
```

4.4.6 Function References

References to functions are also possible in the system

```
val x = ref(let func = λn. n in func);
!x (1)
```

The type of this program is $\texttt{int}[\texttt{1} : \texttt{Pre}; \rho]$ and the effect is ρ'. The pretty printer returns the following output.

```
The program evaluates to the integer value 1.
No mutable variables are altered by this program.
The mutable variable x is a reference to a function of type int → int.
```

4.4.7 References to References

References to references are allowed in the system.

```
val x = ref 1, val y = ref x
!y
```

The type of this program is $\texttt{ref}[\texttt{x} : \texttt{int}[\texttt{1} : \texttt{Pre}; \rho]; \rho']$ and the effect is ρ''. The pretty printer returns the following output.

```
The program evaluates to the mutable variable x.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
The mutable variable y is a reference to the mutable variable x.
```

As stated in the introduction references to references introduce the problem of aliasing where two mutable variables are bound to the same reference cell. Assigning to one of these variables will therefore alter the referenced value of the other. Consider the following example.

```
val x = ref 1, val y = ref x
let val useless = x := 2
in !(!y)
```

Clearly the mutable cell associated with x has been changed however the effect system does not detect this. The effect systems types this program as

$\texttt{int}[1 : \texttt{Pre}; \rho]$ with effect $\texttt{x} : \texttt{int}[2 : \texttt{Pre}; \rho']; \rho''$. This is incorrect as the type of the program should be $\texttt{int}[2 : \texttt{Pre}; \rho]$. An interesting but difficult extension to this project would be to implement a storage effect system which deals with this aliasing issue.

In order to prevent this problem in the implemented system the type of the term a_1 bound to x in the $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ contract cannot be `unit`. This however prevents sequencing in this language.

4.4.8 Conclusion

The experiments carried out and the results produced show the power of the implemented storage effect system. As noted in the conclusion to the previous chapter much of this power is obtained due to the annotation of the analysed base types (excluding the unit and bool base type). Once the effect to be analysed was chosen and the appropriate representation of this effect decided the original system was easily manipulated in order to analyse the storage effect.

Although the experiments performed and the return results appeared to be correct no formal proof of the soundness of this effect system has been provided.

5. Combined Effect System

5.1 Overview

The final effect system considered is a combination of the exception and storage effect systems. The amalgamated system can analyse both the effect of raising and handling an exception and also the effect of allocating, dereferencing and assigning to a mutable variable.

The effect system has been implemented and experiments run as detailed in this chapter.

5.2 The Effect System

This section provides a description of the implemented effect system.

5.2.1 The Source Language

The source language is based on a small subset of ML which allows both the raising and handling of exceptions and the allocation, assignment and dereferencing of mutable variables. As in the previously implemented effect systems it allows higher-order functions, recursive functions and (simplified) pattern matching.

Program:	*prog* ::=	*edecs vdecs a* | *edecs a*	program
		vdecs a | *a*	
Exception Decs:	*edecs* ::=	*edec*, *edecs*	multiple declarations
		| *edec*;	single declaration
Exception Dec:	*edec* ::=	`exception` C	constant exception
		| `exception` D(*type*)	parameterized exception
Variable Decs:	*vdecs* ::=	*vdec*, *vdecs*	multiple declarations
		| *vdec*;	single declaration
Variable Dec:	*vdec* ::=	`val` x = `ref` a	mutable variable
Type:	*type* ::=	`int`	integer type
		| `exn`	exception type
		| `bool`	boolean type
		| `ref`	reference type
		| *type* → *type*	function type
Term:	a ::=	x	identifier

		$\mid i$	integer constant
		$\mid\ !(a_1)$	dereference
		$\mid a_1 := a_2$	assignment
		$\mid \lambda x.a$	abstraction
		$\mid a_1(a_2)$	application
		$\mid$ `let` $x = a_1$ `in` a_2	the `let` binding
		$\mid$ `match` a_1 `with` $pats \mid x \rightarrow a_2$	pattern-matching
		$\mid C \mid D(a)$	exception constructors
		$\mid$ `try` a_1 `with` $x \rightarrow a_2$	exception handler
		$\mid$ `if` a_1 `then` a_2 `else` a_3	selection operation
		$\mid\ a_1 + a_2 \mid a_1 - a_2$	arithmetic operators
		$\mid\ a_1 * a_2 \mid a_1/a_2$	
		$\mid\ a_1$ `orelse` a_2	logical operators
		$\mid\ a_1$ `andalso` a_2	
		$\mid$ `not`(a_1)	
		$\mid\ a_1 = a_2 \mid a_1 <> a_2$	comparison operators
		$\mid\ a_1 > a_2 \mid a_1 < a_2$	
		$\mid\ a_1 >= a_2 \mid a_1 <= a_2$	
Patterns:	$pats ::=$	$p \rightarrow a \mid pats$	multi-case pattern
		$\mid p$	single case pattern
Pattern:	$p ::=$	x	variable pattern
		$\mid i \mid C$	constant patterns
		$\mid D(p)$	constructed pattern

The source language is a concatenation of the source languages as described in the two previous chapters. Note that any user defined exceptions are declared before the allocation of any mutable variables.

5.2.2 The Type Algebra

The type system is based upon the following type algebra.

Type Expressions:	$\tau ::=$	α	type variable
		$\mid$ `int`$[\varphi]$	integer type
		$\mid$ `exn`$[\varphi]$	exception type
		$\mid$ `ref`$[\varphi]$	reference type
		$\mid$ `bool`	boolean type
		$\mid$ `unit`	unit type
		$\mid \tau_1 \overset{\varphi_1,\ \varphi_2}{\rightarrow} \tau_2$	function type
Type Schemes:	$\sigma ::=$	$\forall \alpha_i, \rho_j, \delta_k.\tau$	
Rows:	$\varphi ::=$	$\rho :: K(\rho)$	row variable and kind
		$\mid \top$	all possible elements

		$\mid \varepsilon; \varphi$	element ε plus what is in φ
Row elements:	$\varepsilon ::=$	$i : \pi$	integer constant
		$\mid C : \pi$	constant exception
		$\mid D(\tau)$	parameterized exception
		$\mid x : \tau$	mutable variable
Presence annotations:	$\pi ::=$	`Pre`	element is present
		$\mid \delta$	presence variable
Kinds:	$K ::=$	`INT`(*integer list*)	integer kind
		$\mid$ `EXN`(*set of exceptions*)	exception kind
		$\mid$ `VAR`(*variable list*)	variable kind

The type algebra contains five base types; the integer type, the exception type, the reference type, the boolean type and the unit type. The integer, exception and reference types are annotated with rows indicating the possible values a term of this type may evaluate to. As before a row is made up of a sequence of row elements and terminated with a row variable. The set of row elements are integer constants, exception constructors and mutable variable constructors. A row cannot contain a mixture of these row elements. As before the boolean and unit types remain unannotated.

Note that the function type is now annotated with two rows indicating both the exception effect and the storage effect which may occur on application of this function.

5.2.3 The Kinding Rules

The kinding rules ensure correct kinding and well-formedness.

5.2.3.1 Kinding

$$\vdash \rho :: K(\rho) \qquad \vdash \top :: \texttt{INT}(S) \qquad \frac{i \notin S \qquad \vdash \varphi :: \texttt{INT}(S \cup \{i\})}{\vdash (i : \pi; \varphi) :: \texttt{INT}(S)}$$

$$\frac{C \notin S \qquad \vdash \varphi :: \texttt{EXN}(S \cup \{C\})}{\vdash (C : \pi; \varphi) :: \texttt{EXN}(S)} \qquad \frac{D \notin S \qquad \vdash \varphi :: \texttt{EXN}(S \cup \{D\}) \qquad \vdash \tau \ \texttt{wf}}{\vdash (D(\tau); \varphi) :: \texttt{EXN}(S)}$$

$$\frac{x \notin S \qquad \vdash \varphi :: \texttt{VAR}(S \cup \{x\}) \qquad \vdash \tau \ \texttt{wf}}{\vdash (x : \tau; \varphi) :: \texttt{VAR}(S)}$$

5.2.3.2 Well-formedness

$$\frac{\vdash \alpha\ \mathtt{wf}}{\vdash \mathtt{int}[\varphi]\ \mathtt{wf}} \quad \vdash \varphi :: \mathtt{INT}(\emptyset) \qquad \vdash \mathtt{bool}\ \mathtt{wf} \qquad \frac{\vdash \varphi :: \mathtt{EXN}(\emptyset)}{\vdash \mathtt{exn}[\varphi]\ \mathtt{wf}} \qquad \vdash \mathtt{unit}\ \mathtt{wf} \qquad \frac{\vdash \varphi :: \mathtt{VAR}(\emptyset)}{\vdash \mathtt{var}[\varphi]\ \mathtt{wf}}$$

$$\frac{\vdash \tau_1\ \mathtt{wf} \qquad \vdash \varphi_1 :: \mathtt{EXN}(\emptyset) \qquad \vdash \varphi_2 :: \mathtt{VAR}(\emptyset) \qquad \vdash \tau_2\ \mathtt{wf}}{\vdash \tau_1 \xrightarrow{\varphi_1,\ \varphi_2} \tau_2\ \mathtt{wf}}$$

These rules enforce similar structural invariants as those enforced by the kinding rules in chapters 3 and 4. The invariants ensure the existence of principal unifers and principal typings. They also guarantatee that the annotations of integer types, exception types and reference types remain separate.

The most important change to these rules from previous iterations in earlier chapters is the annotation of function types by two effects and the kinding constraints placed upon both of these effects.

5.2.4 The Typing Rules

5.2.4.1 Initial Environment

The initial typing environment E_0 includes predefined functions which will implement the arithmetic, logical and comparison operators and the `raise` function to raise an exception.

$$\begin{array}{lll}
E_0 = & \{\mathtt{raise}: & \forall\alpha,\rho.\mathtt{exn}[\rho] \xrightarrow{\rho} \alpha, \\
& +,-,*: & \forall\rho_1\rho_2\rho_3\rho_4.\mathtt{int}[\rho_1] \xrightarrow{\rho_2} \mathtt{int}[\rho_3] \xrightarrow{\rho_4} \mathtt{int}[\top], \\
& /: & \forall\rho_1\rho_2\rho_3\rho_4.\mathtt{int}[\rho_1] \xrightarrow{\rho_2} \mathtt{int}[\rho_3] \xrightarrow{\mathtt{Div:Pre};\rho_4} \mathtt{int}[\top], \\
& \mathtt{orelse}, \mathtt{andalso}: & \forall\rho_1\rho_2.\mathtt{bool} \xrightarrow{\rho_1} \mathtt{bool} \xrightarrow{\rho_2} \mathtt{bool}, \\
& \mathtt{not}: & \forall\rho_1.\mathtt{bool} \xrightarrow{\rho_1} \mathtt{bool}, \\
& =, <>, <, >, <=, >=: & \forall\rho_1\rho_2\rho_3\rho_4.\mathtt{int}[\rho_1] \xrightarrow{\rho_2} \mathtt{int}[\rho_3] \xrightarrow{\rho_4} \mathtt{bool}\}
\end{array}$$

A number of predeclared exceptions have also been included in the system. As in the storage effect system, mutable variable allocation is achieved by updating the initial environment.

5.2.4.2 Typing of Expressions

The typing rules for the system follow.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}\ \text{ID}$$

$$\frac{\vdash \varphi' :: \texttt{INT}(\{i\}) \qquad \vdash \varphi_1 :: \texttt{EXN}(\emptyset) \qquad \vdash \varphi_2 :: \texttt{VAR}(\emptyset)}{E \vdash i : \texttt{int}[i : \texttt{Pre}; \varphi']/\varphi_1,\ \varphi_2}\ \text{INT}$$

$$\frac{\vdash \varphi_1 :: \texttt{EXN}(\emptyset) \qquad \vdash \varphi_2 :: \texttt{VAR}(\emptyset)}{E \vdash \texttt{true} : \texttt{bool}/\varphi_1,\ \varphi_2}\ \text{TRUE}$$

$$\frac{\vdash \varphi_1 :: \texttt{EXN}(\emptyset) \qquad \vdash \varphi_2 :: \texttt{VAR}(\emptyset)}{E \vdash \texttt{false} : \texttt{bool}/\varphi_1,\ \varphi_2}\ \text{FALSE}$$

$$\frac{E \vdash \texttt{not}\ (a) : \tau/\varphi_1,\ \varphi_2}{E \vdash \texttt{not}\ a : \tau/\varphi,\ \varphi_2}\ \text{NOT} \qquad \frac{E \vdash \circ\ (a_1)\ (a_2) : \tau/\varphi_1,\ \varphi_2}{E \vdash a_1 \circ a_2 : \tau/\varphi,\ \varphi_2}\ \text{BOP}$$

$$\frac{E \vdash a_1 : \texttt{ref}[\varphi_2']/\varphi_1,\ \varphi_2}{E \vdash !a_1 : \texttt{ConcatenateRefs}(\varphi_2')/\varphi_1,\ \varphi_2}\ \text{DEREF}$$

$$\frac{\begin{array}{c}E \vdash a_1 : \texttt{ref}[\varphi_2']/\varphi_1,\ \varphi_2 \\ E \vdash a_2 : \tau/\varphi_1,\ \varphi_2 \qquad E \vdash \varphi_2'' = \texttt{UpdateRefs}(\tau, \varphi_2')\end{array}}{E \vdash a_1 := a_2 : \texttt{unit}/\varphi_1,\ \varphi_2''}\ \text{ASSIGN}$$

$$\frac{\begin{array}{c}\vdash \tau_1\ \texttt{wf} \\ E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi_1',\ \varphi_2' \qquad \vdash \varphi_1 :: \texttt{EXN}(\emptyset) \qquad \vdash \varphi_2 :: \texttt{VAR}(\emptyset)\end{array}}{E \vdash \lambda x.a : (\tau_1 \overset{\varphi_1',\ \varphi_2'}{\rightarrow} \tau_2)/\varphi_1,\ \varphi_2}\ \text{ABS}$$

$$\frac{E \vdash a_1 : (\tau' \overset{\varphi_1,\ \varphi_2}{\rightarrow} \tau)/\varphi_1,\ \varphi_2 \qquad E \vdash a_2 : \tau'/\varphi_1,\ \varphi_2}{E \vdash a_1(a_2) : \tau/\varphi_1,\ \varphi_2}\ \text{APP}$$

$$\frac{\begin{array}{c}E \oplus \{x : \texttt{Gen}(\tau_1, E, \varphi_1,\ \varphi_2)\} \vdash a_1 : \tau_1/\varphi_1,\ \varphi_2 \\ E \oplus \{x : \texttt{Gen}(\tau_1, E, \varphi_1,\ \varphi_2)\} \vdash a_2 : \tau/\varphi_1,\ \varphi_2\end{array}}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau/\varphi_1,\ \varphi_2}\ \text{LET}$$

$$\frac{\begin{array}{c}E \vdash a_1 : \tau_1/\varphi_1,\ \varphi_2 \qquad \vdash\ p : \tau_1 \Rightarrow E' \qquad \vdash \tau_1 - p \rightsquigarrow \tau_2 \\ E \oplus E' \vdash a_2 : \tau/\varphi_1,\ \varphi_2 \qquad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi_1,\ \varphi_2\end{array}}{E \vdash \texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi_1,\ \varphi_2}\ \text{MATCH}$$

$$\frac{\vdash \varphi_1' :: \texttt{EXN}(\{C\}) \qquad \vdash \varphi_1 :: \texttt{EXN}(\emptyset) \qquad \vdash \varphi_2 :: \texttt{VAR}(\emptyset)}{E \vdash C : \texttt{exn}[C : \texttt{Pre}; \varphi_1']/\varphi_1,\ \varphi_2}\ \text{EXC}$$

$$\frac{\tau \leq \mathit{TypeArg}(D) \qquad E \vdash a : \tau/\varphi_1,\ \varphi_2 \qquad \vdash \varphi_1' : \texttt{EXN}(\{D\})}{E \vdash D(a) : \texttt{exn}[D(\tau); \varphi_1']/\varphi_1,\ \varphi_2}\ \text{PEXC}$$

$$\frac{E \vdash a_1 : \tau/\varphi_1' \qquad E \oplus \{x : \texttt{exn}[\varphi_1']\} \vdash a_2 : \tau/\varphi_1,\ \varphi_2}{E \vdash \texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2 : \tau/\varphi_1,\ \varphi_2}\ \text{TRY}$$

$$\frac{E \vdash a_1 : \texttt{bool}/\varphi_1,\ \varphi_2 \qquad E \vdash a_2 : \tau/\varphi_1,\ \varphi_2 \qquad \vdash a_3 : \tau/\varphi_1,\ \varphi_2}{E \vdash \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3 : \tau/\varphi_1,\ \varphi_2}\ \text{IF}$$

$E_1 \oplus E_2$ is the asymetric concatenation of the two environments E_1 and E_2. $E_1 \oplus E_2\ (x) = E_2\ (x)$ if $x \in Dom(E_2)$ and $E_1 \oplus E_2\ (x) = E_1\ (x)$ if $x \in Dom(E_1) \setminus Dom(E_2)$.

5.2.4.3 Typing of Patterns

$$\vdash x : \tau \Rightarrow \{x : \tau\} \qquad \vdash i : \texttt{int}[i : \pi; \varphi] \Rightarrow \{\}$$

$$\vdash C : \texttt{exn}[C : \pi; \varphi] \Rightarrow \{\} \qquad \frac{\tau \leq \mathit{TypeArg}(D) \qquad \vdash p : \tau \Rightarrow E}{\vdash D(p) : \texttt{exn}[D(\tau); \varphi] \Rightarrow E}$$

5.2.4.4 Pattern Subtraction

$$\vdash \texttt{int}[i : \pi; \varphi] - i \rightsquigarrow \texttt{int}[i : \pi'; \varphi] \qquad \vdash \texttt{exn}[C : \pi; \varphi] - C \rightsquigarrow \texttt{exn}[C : \pi'; \varphi]$$

$$\frac{\vdash \tau'\ \texttt{wf}}{\vdash \tau - x \rightsquigarrow \tau'} \qquad \frac{\vdash \tau - p \rightsquigarrow \tau'}{\vdash \texttt{exn}[D(\tau); \varphi] - D(p) \rightsquigarrow \texttt{exn}[D(\tau'); \varphi]}$$

5.2.4.5 Instantiation and Generalization

$\tau' \leq \forall \alpha_i \rho_j \delta_k . \tau$ if and only if there exists τ_i, φ_j, π_k such that $\vdash \tau_i\ \texttt{wf}$ and $\vdash \varphi_j :: K(\rho_j)$ and $\tau' = \tau\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\}$

$\texttt{Gen}(\tau, E, \varphi)$ is $\forall \alpha_i \rho_j \delta_k . \tau$ where $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$

5.2.4.6 Reference Operations

These reference operations are necessary as a term may evaluate to more than one mutable variable.

$\texttt{ConcatenateRefs}(\varphi_2)$ returns the union of all the types in the mutable variable constructors in a row, φ_2, made up of a sequence of mutable variable row elements.

$\texttt{UpdateRefs}(\tau, \varphi)$ updates the type in each mutable variable constructor in a row, φ, made up of a sequence of mutable variable row elements to indicate that each mutable variable may evaluate to a type of τ. The updated row is returned by the function. If φ is of the form $x : \tau'; \rho$ (ie. contains only one mutable variable constructor) then the row $x : \tau; \rho$ is returned. If φ is of the form $x_1 : \tau'_1; \ldots; x_n : \tau'_n; \rho$ (ie. contains more than one mutable variable constructor)

then the row $x_1 : \tau_1' \otimes \tau; \ldots; x_n : \tau_n' \otimes \tau; \rho$ is returned where $\tau' \otimes \tau$ is the union of the two types τ' and τ.

The type inference rules are the combination of the type inference rules for both of the previous effect systems. They have been altered so that they have an effect component for both the exception effect and the storage effect.

5.3 Implementation

The implemented system is based on the combination of the source code for both of the previously implemented effect system.

5.3.1 Using the Implementation

The implemented combined effect system is used in the same way as both the implemented exception effect system and the implemented storage effect system. Again the effect system is started using a single command starting the mosml environment. Analysis is then carried out on command line input or a specified file. A parser, lexer and pretty printer have all been implemented.

5.4 Experimentation

The following section contains experiments illustrating some of the behaviour of and the resulting output produced by the implementation.

5.4.1 Exception References

A mutable variable which is a reference to an exception can be allocated, dereferenced and assigned to by this system.

```
exception C1, exception C2;
val x = ref 1, val y = ref 2, val c1ref = ref C1, val c2ref = ref C2;
raise (!(if !x >!y then c1Ref else c2Ref))
```

The type of this program is α, the exception effect is $\texttt{C2} : \texttt{Pre}; \texttt{C1} : \texttt{Pre}; \rho$ and the storage effect is ρ'. The pretty printer returns the following output.

```
An exception always escapes from this program.
The exceptions C2 and C1 may escape from this program.
No mutable variables are altered by this program.
The mutable variable x is a reference to the integer value 1.
The mutable variable y is a reference to the integer value 2.
The mutable variable c1ref is a reference to the exception C1.
The mutable variable c2ref is a reference to the exception C2.
```

5.4.2 Reference Parameterized Exceptions

An exception may be parameterized with a reference variable.

```
exception D(ref);
val xRef = ref 1;
(try raise (D(xRef))
with x → match x with D(x) → x | y → raise (y)) := 2
```

The type of this program is `unit`, the exception effect is $\mathtt{D}(\alpha);\rho$ and the storage effect is $\mathtt{xRef} : \mathtt{int}[2 : \mathtt{Pre}; \rho'];\mathtt{rho}''$. The pretty printer returns the following output.

```
The program evaluates to the unit type.
No exception escapes from this program.
The mutable variable x is altered by this program.
The mutable variable x is a reference to the integer value 2.
```

5.5 Conclusion

The implemented system is powerful in that it accurately analyses the effects of a program even when the behaviour of the possible exception effects and storage effects are in some way related as shown in the examples. The small set of examples detailed is far from a complete representation of the complete behaviour of the system. As with the augmented effect system and the storage effect system an important extension may be to prove the soundness of this type system.

6. Conclusion

In this project effect systems were implemented and evaluated for the effect of raising and handling exceptions and for the effect of allocating, assigning and dereferencing mutable variables. These system were then combined to create an effect system able to analyse both of these computational effects. This combined system was also implemented and tests performed.

The experiments detailed in this document and the analysis carried out in the relevant chapters show that the implemented effect systems gave quite detailed information concerning the possible effects of executing a program. Much of the power of these systems came from the annotation of some of the base types and the function type in these systems. These annotations increased the accuracy of the analysis carried out by the effect systems as they indicated the values a term may evaluate to and the effects that a function application may cause.

The annotated base types had associated rows indicating the values that a term of this base type may have. In order to reduce the complexity of the project not all base types were implemented. The boolean type was one such unannotated type. A relatively simple extension of the implemented system would be to annotate this type as due to the small size of the boolean set some powerful inference rules could be designed for the selection operation and logical operators. Note the unit type was also unannotated as, like in ML, this has not associated value. Function types are annotated with rows indicating the possible effects of the application of this function (although the effect of the parameters to a function must also be considered).

Some problems highlighted with the implemented systems included the issue of pretty printing the possible result of the application of a function discussed in chapter 3. Also the problem of aliasing of mutable variables still has not been satisfactorily resolved analysed. The simple solution in place currently removes the possibility of imperative sequencing which may be desirable in the source language to be analysed.

Important extensions to this project would to prove the soundness of the augmented exception effect system, the storage effect system and the combined effect system. Such a proof was carried out for the original effect system described in the Leroy and Pessaux paper [5].

The experiments performed and described in this document are somewhat artifical (except perhaps the `change` function). It would be interesting to consider the applicability of the described effect systems to real programs.

Leroy and Pessaux also suggest in their paper extension of the original exception

effect system to the entire ML language. This could be considered for both the storage effect system and the combined effect system as well as the augmented exception effect system.

Bibliography

[1] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[2] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Unix Programming Tools. O' Reilly, 1995.

[3] John Lucassen and David Gifford. Polymorphic effect systems. In *15th Symposium Principles of Programming Languages*, pages 47–57. ACM Press, 1988.

[4] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[5] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[6] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type region and effect inference. Technical Report EMP-CRI E/150, 1991.

[7] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.