LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-408

TYPES AND EFFECTS

TOWARDS THE INTEGRATION OF FUNCTIONAL AND IMPERATIVE PROGRAMMING

John M. Lucassen

August 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TR-408 LUCASSEN TYPES & EFFECTS - TOWARDS THE INTEGRATION OF FUNCTIONAL & IMPERATIVE PROGRAMMING

Types and Effects

Towards the Integration of Functional and Imperative Programming

by

John M. Lucassen

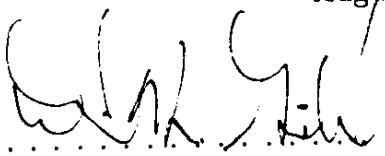
S.B., Electrical Engineering and Computer Science
S.M., Computer Science and Engineering
Massachusetts Institute of Technology
(1983)

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 1987

Signature of Author 
Department of Electrical Engineering and Computer Science
August 6, 1987

Certified by 
David K. Gifford
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students

Types and Effects

Towards the Integration of Functional and Imperative Programming

John M. Lucassen

Abstract

We propose a new class of programming languages in which every expression has both a *type* and an *effect*: the type describes what sort of value the expression may return, and the effect describes what sort of side-effects the expression may have. Effects are described in terms of *regions*, which describe what part of the state of the computation may be affected.

We show how the type system of the second-order lambda-calculus can be generalized to incorporate effect and region specifications. This leads to polymorphism with respect to types, effects, and regions.

Our type and effect system makes it possible to embed functional program fragments within imperative programs, and vice versa, while retaining the benefits of the chosen programming style in each program fragment. The type and effect system can also be used to verify the encapsulation invariant of a monitored data type. It also offers a clean solution to the problem of first-class polymorphism in an imperative language.

To test our ideas, we have designed a small imperative language called MFX, and we have implemented a compiler that analyzes MFX programs and translates them into dataflow graphs. Our long-term objective is the integration of functional and imperative programming into a single programming model that retains the benefits of both programming styles.

Thesis supervisor: David K. Gifford
Title: Associate Professor of Computer Science

Key words and phrases: programming languages, types, effects, effect checking, side-effect specifications, polymorphism, concurrent programming, multiprocessing, flow analysis, FX

1987 CR Categories: D.1.3 [*Programming Techniques*] Concurrent Programming; D.1.m [*Programming Techniques*] Miscellaneous; D.3.2 [*Programming Languages*] Language Classifications; D.2.1 [*Software Engineering*] Requirements and Specifications — Languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and verifying and reasoning about programs — mechanical verification, specification techniques

Acknowledgements

I would like to thank my thesis supervisor, Professor David Gifford, for his guidance, encouragement and support, and my thesis readers, Professors Albert Meyer and Rishiyur Nikhil, for their supportive criticism.

I also wish to acknowledge the advice and support I have received from my friends and fellow students — in the Laboratory for Computer Science, the Graduate Student Council, the Ballroom Dance Club, and elsewhere. In particular, I would like to thank Steve Berlin, Mike Blair, Mark Day, Pierre Jouvelot, Gary Leavens, Brian Oki, Jonathan Reese, Mark Reinhold, Richard Schooler, Mark Sheldon, Jim Stamos, and Jim O'Toole.

Finally, I am grateful to the IBM Corporation for supporting part of my graduate study through the IBM Fellowship Program. This research was also supported in part by DARPA/ONR contract number N00014-83-K-0125.

Table of Contents

Table of Symbols

A	type assignment	Dom	domain
b	Boolean (*Bool*)	$\mathcal{P}ow$	power set
B	kind assignment	FV	free ordinary variables
C	context	FDV	free description variables
d	description variable (*Dvar*)	FL	free locations
δ	description (*Desc*)	FRC	free region constants
e	expression (*Exp*)	$:$	kind of a description
ϵ	effect description (*Effect*)	$:$	type of an expression
γ	region map	$!$	effect of an expression
κ	kind (*Kind*)	R	region tag of a location
l	location (*Loc*)	T	type tag of a location
m	monitor (*Mon*)	$Reach$	reached locations
μ	permutation of locations	$\stackrel{\text{red}}{\Longrightarrow}$	reduction
r	region constant (*Rconst*)	$M[\![\ ]\!]$	meaning of an expression
ρ	region description (*Region*)	Acc	accessible region constants
σ	store (*Store*)	$\mathcal{A}$	allocated locations
t	type constant (*Tconst*)	$\mathcal{R}$	read locations
τ	type description (*Type*)	$\mathcal{W}$	written locations
θ	state (*State*)	$\doteq$	equivalence of states
v	value (*Val*)	$Erase$	type erasure
x	ordinary variable (*Var*)	$\stackrel{\text{erased}}{\Longrightarrow}$	typeless reduction

♣

Chapter 1. Introduction

1.1 Motivation

Our research focuses on two important classes of programming languages: *functional* languages and *imperative* languages. In a functional language, program fragments are referentially transparent, and their evaluation has no side-effects. In an imperative language, program fragments are not generally referentially transparent, and their evaluation may have side-effects that affect the evaluation of subsequent program fragments. Examples of functional languages include FP and ID as well as the pure dialects of Lisp; examples of imperative languages include a whole spectrum of languages ranging from assembly language and FORTRAN to Ada, as well as Lisp, Scheme and ML.

Each class of languages has its advantages and disadvantages. Functional languages, on the one hand, correspond closely to conventional mathematical notions such as variables, functions and functionals. As a result, they are quite amenable to mathematical analysis. This gives rise to a variety of benefits, including the ability to formally prove programs correct and, it is hoped, the ability to implement them efficiently on highly parallel hardware. On the negative side, even state-of-the-art functional languages seem to lack the expressive power needed to write programs that interact with users, with other programs, or with persistent data. Moreover, functional languages generally do not permit the programmer to specify explicitly when memory can be reused. This makes it difficult to implement such languages efficiently on conventional hardware.

Imperative languages, on the other hand, correspond closely to conventional *computational* notions, such as processors, memory, input and output and so on. As a result, they are quite amenable to efficient implementation on conventional hardware, and the relation between a program and its implementation is easily understood by the programmer, which is important if efficient programs are to be written. On the negative side, the use of side-effects in imperative programs tends to make it difficult to prove such programs correct, and it has been argued that imperative programs are in general more difficult to understand and to maintain than functional programs. Moreover, imperative languages generally force the programmer to combine the description of a computation with the description of how that computation should be organized, and the language constructs for doing so are generally based on conventional computer architectures. This tends to make it difficult to implement such languages efficiently on highly parallel hardware.

We believe that much can be gained by integrating functional and imperative programming into a single programming model, so that programmers can take advantage of the best aspects of both.

1.2 Background

We take the view that an imperative language is in essence a functional language extended with side-effect operators. According to this view, an imperative program actually consists of functional fragments and imperative fragments, all written in the same language. In order to achieve the full benefit of each of these programming styles in the corresponding program fragments, it is essential that the compiler as well as the programmer are capable of distinguishing between functional and imperative program fragments, and that they generally agree on the classification of individual program fragments.

The process of determining the side-effects of a program fragment is usually known as *flow analysis*; in keeping with the terminology used in the remainder of this thesis, we will refer to it as *effect analysis*. Conventional optimizing compilers employ effect analysis in order to identify optimization opportunities. We take the view that such analysis must be *interprocedural*, so that the programmer can make use of procedural abstraction without incurring a performance penalty. Moreover, we take the view that procedures should be first-class values, *i.e.* that it should be possible to pass procedures as parameters, return procedures as the values of procedure calls, assign procedures to variables and store procedures in data structures.

Although there has been a great deal of research on interprocedural effect analysis, none of the standard methods are effective when procedures are first-class values [Ban78] [Bar78] [Wei80]. Specifically, all the algorithms we encountered in our survey require that the call graph of the program be known statically. When procedures are first-class values, however, the call graph of a program cannot in general be computed in advance, and can only be approximated. The use of approximation algorithms in a performance-critical optimization creates uncertainties and discontinuities in the behavior of the compiler, and may therefore discourage the use of procedural abstraction, which is contrary to our philosophy.

Since the call graph of a program cannot in general be computed in advance, we sought to find some way to perform effect analysis without using a call graph at all. One way to do this is to require the programmer to supply enough information to turn the effect estimation problem into an effect *verification* problem that can be solved exactly. Our proposal for doing this is described in the next section.

1.3 Proposal

In this thesis we propose a new approach to programming language design. The core of our proposal is a polymorphic *type and effect system*, in which type and effect specifications are integrated into single, coherent framework. The effect specifications are sufficiently detailed to permit interprocedural effect analysis without the use of a call graph.

The central idea is that every expression in a program should have both a *type* and an *effect* specification: the type specification describes the value returned by the expression, and the effect specification describes the potential side-effects of the expression. Most of these specifications can be inferred automatically by the compiler; the programmer must supply specifications only at certain points in the program. We propose to enlist the power of the type system to ensure that the effect specification of each subroutine is propagated to all the points in the program where it may be called, even if the subroutine is passed as a parameter, returned as a value, stored in a data structure, or compiled separately. This permits interprocedural effect analysis using exact, well-defined, and efficient algorithms, and avoids the uncertainties and performance discontinuities associated with approximation algorithms.

The benefits of type checking have long been recognized:

- *programmers* can use types to express machine-verifiable assertions and interface specifications;
- *compilers* can take advantage of type information to produce a more efficient implementation; and
- *language designers* can use types to express constraints in the language design.

Although many programming languages support type checking, most do not provide any support at all for *effect checking*, *i.e.* for the use of machine-verifiable effect assertions and specifications. Notable exceptions include Euclid [Lam77] and the work of Reynolds [Rey78].

By incorporating effect specifications into the type system, we hope to achieve three major advantages:

- A type and effect system helps *programmers* specify the potential side-effects of program modules in a way that is machine-verifiable. We believe that the use of effect specifications may lead to improvements in the design and maintenance of imperative programs.
- A type and effect system helps *compilers* identify opportunities for optimizations that would be difficult to perform otherwise in a higher-order imperative language, such as concurrent evaluation and memoization. We believe that the ability to perform these optimizations effectively in the presence of side-effects represents a major step towards the integration of functional and imperative programming.

- A type and effect system lets *language designers* express and enforce effect constraints as part of the language definition. We believe that this will substantially widen the set of programming errors that can be detected by a compiler. Moreover, it permits a clean resolution of the interaction between side-effects and first-class polymorphism.

Types and effects are surprisingly similar; most strikingly, all the language constructs of Reynolds' second-order lambda-calculus [Rey74], McCracken's higher-order lambda-calculus [McC79] [McC82], and Cardelli and Wegner's' second-order lambda-calculus with bounded quantification [Car85] can be generalized to effects. Consequently, we believe that effect systems are a natural companion to conventional type systems.

1.4 Outline

The rest of this thesis is organized as follows.

In Chapter 2 we present the objectives for our effect specification language. We survey a variety of possible forms of effect specifications, and we assess the extent to which they meet our objectives. We next describe and motivate the new effect specification language that has emerged from our research. We conclude with a discussion of related research.

In Chapters 3 through 6 we present a higher-order, imperative demonstration language with a type and effect system. We have called this language *MFX*, for Mini-FX. (*FX* is a complete programming language with a type and effect system that is currently being developed at MIT by the Programming System Research Group [Gif87].) In an attempt to introduce the features of *MFX* gradually, we introduce the language in three stages.

In Chapter 3 we present the syntax of *MFX-1* (Mini-FX level 1), a subset of *MFX* that has polymorphism and side-effect operators but that does not have constructs for dealing with private regions or explicit concurrency. We also give an informal description of its semantics, and we present the *static semantics*, which determine whether or not a given *MFX-1* program is well-formed.

In Chapter 4 we present the dynamic semantics of *MFX-1*, and present our propositions regarding the soundness of the type and effect system.

In Chapter 5 we extend *MFX-1* with constructs for declaring and using private regions; the extended language is called *MFX-2*.

In Chapter 6 we extend *MFX-2* with constructs for introducing and managing explicit concurrency; the extended language is called *MFX-3*.

In Chapter 7 we present various possible extensions to *MFX*, including syntactic sugar and built-in data types; higher-order types; and recursion. The specific benefits of an effect system in relation to polymorphism are also discussed in this chapter.

In Chapter 8 we show how to use the effect information in an *MFX* program to find opportunities for concurrent evaluation. We present a compiler that compiles *MFX* programs into dataflow graphs. Topics covered include interference, graph minimization, and the construction of dataflow graphs. We also present simulation results.

In Chapter 9 we discuss the limitations of our approach, sketch some topics for future research, and summarize the results of our research.

1.5 A Note on Rigor

In the course of our research, we have come across far more opportunities than we have been able to investigate in detail. We have limited the scope of this thesis to a body of results that we believe is solid and fairly well-contained, but broad enough to suggest the opportunities that we have identified. Our primary objective has been to formulate and explain what we consider to be the major results of our research; while we outline the proofs of most of our propositions in some detail, we do not always provide rigorous proofs. We hope that the product of this approach will prove to be of interest to a broad audience ranging from programming language designers to programming language theorists.

♣

Chapter 2. Effect Specifications

2.1 Introduction

In a language with a type and effect system, the potential effect of each expression and the potential *latent* effect of each subroutine are described by effect specifications. We have adopted the view that it should be possible to perform effect analysis strictly on the basis of effect specifications. In the design of the effect specification language we adopted the following objectives: effect specifications should be

- useful to the programmer and to the compiler;
- applicable to user-defined as well as built-in constructs;
- suitable for static analysis;
- sufficiently fine-grain; and yet
- sufficiently abstract.

The purpose of this chapter is to describe and justify the form of effect specifications that has emerged from our research.

The rest of this chapter is organized as follows. We begin by reviewing the objective of utility in more detail, especially from the compiler's point of view. We then consider the limits of static analysis. Next, we turn to fine-grain effect specifications, and consider the advantages and disadvantages of a variety of possible approaches. We then focus on the chosen approach, and illustrate how it fits into a type and effect system. We conclude by comparing our results with previous work on fine-grain effect specifications.

2.2 Utility of Effect Descriptions

First and foremost, effect specifications must be useful, both to the programmer and to the compiler. From the programmer's point of view, effect specifications serve as documentation that enhances program readability and identifies effect invariants that may be important for performance. From the compiler's point of view, effect specifications provide information that help it identify optimization opportunities. In this section we focus on utility to the compiler; we will return to the programmer's point of view later in this chapter.

In determining the utility of effect specifications to the compiler, we have focused on two classes of optimizations:

- relaxations of evaluation order, such as concurrent evaluation, and
- common subexpression elimination and memoization.

2.2.1 Effect Information and Evaluation Order

Any sequential program can be transformed into a (possibly concurrent) program in which expressions are evaluated in an order different from that specified by the standard semantics. Such program transformations include *code motion*, or CM, and *eager*, *lazy* and *concurrent* evaluation.

In a functional language, expressions are *referentially transparent*: the value of every expression is completely determined by the values of its free variables, which are fixed throughout their lifetimes. Since the value of an expression is determined as soon as its free variables are bound, the value of the expression is the same regardless of when it is computed. Therefore, none of these program transformations changes the meaning of a functional program.

In an imperative language, on the other hand, expressions are not referentially transparent: an expression not only returns a value, but may also have side-effects, and both are determined not only by the values of the free variables, but also by the contents of the memory locations read by the expression. Since the contents of memory may change during the course of a computation, the value and the effects of an expression generally depend on when the expression is evaluated. Because of this, program transformations that change the evaluation order generally change the meaning of an imperative program. Nevertheless, there are situations in which the compiler can perform such program transformations without changing the meaning of the program.

In a sequential language, two expressions are said to *interfere* iff one expression may write to locations that the other expression may read or write. Given any pair of expressions, the absence of interference between two expressions is a sufficient, although not necessary, condition to guarantee that evaluating the expressions out of order or concurrently does not change the meaning of the program [Ber66]. For example, consider the expression

(BEGIN e_1 e_2)

which evaluates e_1 and then e_2. If neither expression writes to locations that the other expression may read or write, then the expressions do not interfere, and can be evaluated out of order or concurrently without changing the meaning of the program.

Given a set of expressions, a compiler can identify opportunities for CM and concurrent evaluation by determining whether or not the expressions may interfere with one another. To make this analysis possible, the effect specifications of an expression must indicate the locations that may be read and/or written by the expression.

Determining opportunities for eager and lazy evaluation is somewhat more difficult, since a program that would halt under the standard evaluation order

might diverge under eager evaluation, and conversely for lazy evaluation. Except for divergence, however, a compiler can identify opportunities for eager and lazy evaluation by determining whether or not a given expression may interfere with other expressions. Therefore, except for divergence, the analysis of eager and lazy evaluation requires the same information as CSE analysis.

2.2.2 Common Subexpression Elimination and Memoization

Any program that contains two identical instances of an expression in the same scope can be transformed so that the expression will be evaluated just once and its value communicated to the two places in the program that need it. This program transformation is known as *common subexpression elimination*, or CSE.

Similarly, any subroutine f can be replaced by a subroutine f' that keeps a table of the argument tuples and resulting values of past invocations, and that simply returns the value stored in the table whenever it is called with an argument tuple on which it has been called before. This program transformation is known as *memoization*. Memoization can be regarded as a form of dynamic common subexpression elimination.

In a functional language, identical expressions in the same scope return the same value, and every subroutine returns the same value each time it is called with the same arguments. Therefore, neither CSE nor memoization changes the meaning of a functional program.

In an imperative language, CSE and memoization generally do change the meaning of a program. Nevertheless, there are situations in which a compiler can perform these program transformations without changing the meaning of the program.

The process of CSE can be divided into two steps: given two identical instances of an expression in the same scope, the instances must be

1. *moved* to the same point in the program, and subsequently
2. *merged* into a single expression whose value is shared.

In order to identify opportunities for CSE in an imperative language, the compiler must determine whether either of these two steps (code motion and merging) may change the meaning of the program.

The first step, code motion, is possible iff the expression instances are not separated by an expression with which they may interfere. Once the expression instances have been moved to the same point in the program, the second step, merging them, is possible iff the expression is *idempotent*, *i.e.* if evaluating it repeatedly has the same effects, and returns the same value, as evaluating it just once.

An expression that only reads memory locations and does not write them is idempotent. Similarly, an expression that only writes memory locations and does not read them is also idempotent. However, such *blind writes*

seem to be rare in practice. In general, an expression that reads and writes overlapping sets of locations is not idempotent. Finally, an expression that *allocates* new (writable) memory locations is not generally idempotent, since two instances of a single location are in general distinguishable from two distinct locations that happen to contain the same value.

Given two identical expression instances in the same scope, a compiler can determine whether CSE is possible by verifying that the expression instances do not interfere with any of the expressions that separate them, and that the expression itself is idempotent. To make this analysis possible, the effect specifications of an expression must indicate the locations that may be allocated, read and/or written by the expression.

Memoization, like CSE, can be regarded as consisting of two steps: the first call with a given argument tuple is evaluated normally, and all subsequent calls with the same argument tuple are

1. *moved* (in time) to the time of the first call, and
2. *merged* into a single expression whose results are shared.

Since successive calls on the subroutine can be separated (in time) by arbitrary expressions, the first step is possible iff the subroutine has no latent effects that can interfere with other expressions, *i.e.* iff the subroutine cannot read and/or write any (global, writable) memory locations. As for the second step, successive calls can be merged iff the body of the subroutine is idempotent. Therefore, analysis of memoization opportunities requires the same information as CSE analysis.

2.2.3 Explicit Concurrency

In a language that permits explicit concurrency, evaluating two expressions out of order or concurrently can change the meaning of the program even if the expressions do not read and/or write any common locations:

- if two expressions e_1 and e_2 both read the contents of a location that is incremented concurrently by some third computation, then evaluating e_1 and e_2 out of order may cause them to observe inconsistent values;
- if e_1 and e_2 write to distinct locations that are read concurrently by some third computation, then evaluating e_1 and e_2 out of order may cause the third computation to observe inconsistent values.

In fact, the presence of explicit concurrency creates invisible, asynchronous channels of communication between otherwise unrelated locations. This tends to inhibit optimizations such as CM and concurrent evaluation.

In chapters 2 through 5 we deal exclusively with sequential languages, in which this difficulty does not arise. In chapter 6 we show how to use the concepts introduced in the previous chapters to design a language with explicit concurrency in which all interference between concurrent expressions is mediated by monitors and critical sections. This permits the compiler to identify opportunities for CM and concurrent evaluation using the same

analysis that is used for sequential programs. The interaction between concurrent computations is expressed using a new effect, MCALL, which stands for "monitor call". Interference in the presence of MCALL effects is discussed in detail in Chapter 8.

2.3 Suitability for Static Analysis

In general, most interesting properties of programs, such as the reachability of subexpressions, are undecidable. Therefore, one of the challenges in developing a method for static program analysis is to identify program attributes that are informative, and yet can be determined statically. As an illustration of the difficulties of static analysis, consider the expression

(IF e_1 e_2 e_3)

which evaluates e_1, and then evaluates either e_2 or e_3 depending on whether the value of e_1 was TRUE or FALSE. This expression is equivalent either to

(BEGIN e_1 e_2)

or to

(BEGIN e_1 e_3)

depending on the value of e_1. It is clear that unless e_2 and e_3 happen to have the same effect, the effect of the expression as a whole depends on the value of e_1. This value cannot in general be determined statically; moreover, it is not generally the same each time the expression is evaluated. In order to meet our goal of static effect analysis, we have decided not to allow effect specifications to depend on run-time values.

In the example above, this means that we can only conclude that the expression as a whole either has the effect of the expression (BEGIN e_1 e_2) or that of the expression (BEGIN e_1 e_3), depending on whether the value of e_1 is TRUE or FALSE. Since both of these cases may occur, we conclude that the effect of the expression as a whole is some combination of these two effects.

2.4 Fine-Grain Specifications

When analyzing expressions for interference, it matters not only *whether* a given expression may read and/or write: it also matters *which locations* the expression may read and/or write. We call this the *extent* of the effect. Consider, for example, the expression

(BEGIN e_1 e_2)

which evaluates e_1 and e_2, in that order, and returns the value of e_2. If both e_1 and e_2 may read and write an unspecified set of locations, then they might interfere, and evaluating them out of order or concurrently might change the meaning of the program. If, on the other hand, it is known that there are no locations that may be both written by e_1 and read or written by e_2, nor vice versa, then the expressions do not interfere, and may be evaluated out of order or concurrently. For example, in the expression below, the assignments to **x** and **y** may be performed concurrently, provided that **x** and **y** denote distinct locations.

```
(BEGIN
  (SET x TRUE)
  (SET y FALSE))
```

Similarly, when analyzing subroutines for memoizability, it matters not only *whether* each subroutine may allocate writable memory locations; it also matters what happens to those locations afterwards, *i.e.* whether they are strictly for local use or whether they are exported. Consider, for example, the subroutine

```
(LAMBDA (x:BOOL)
  (mkref x))
```

which allocates a new (writable) memory location, initializes it to the value of its argument **x**, and returns the location, and compare it with the subroutine

```
(LAMBDA (x:BOOL)
  (contents (mkref x)))
```

which allocates a new (writable) memory location, initializes it to the value of its argument **x**, and returns the contents of the location, *i.e.* the value of **x**. Clearly, the first subroutine can not generally be memoized without changing the semantics of the program, since two instances of a single writable memory location are in general distinguishable from to two distinct locations that happen to contain the same value. The second subroutine, on the other hand, can be memoized without changing the meaning of the program, even though it allocates a writable memory location and reads its contents, because the extent of these effects is confined to the subroutine body.

The observation that not only the sort of effect is important, but also its extent, leads to the notion of *fine-grain* effect specifications. The basic idea

is to use effect specifications of the form (*sort extent*), where *sort* indicates the sort of effect (which may be allocation, reading, writing and so forth), and *extent* indicates its extent. We now turn the question of how the extent of an effect should be expressed.

2.4.1 Locations

The most direct way to express the extent of effects is in terms of memory locations. Using this technique, the effect of each expression is expressed as a combination of zero or more effects of the form (ALLOC l) (short for ALLOCATE), (READ l) and (WRITE l), where l ranges over locations. For example, an expression that increments the contents of the location l_0 would have the effect $\{(\text{READ } l_0), (\text{WRITE } l_0)\}$. This notation is simple and unambiguous, and provides the compiler with precisely the information it needs. However, it is not particularly attractive from the programmer's point of view, since it forces the programmer to be aware of the details of memory management. In most contemporary high-level languages, these details are hidden from the programmer.

In fact, this method has a more serious drawback: it can be used only when the locations used by the program are known statically. In practice, this is not the case: most contemporary high-level languages support dynamic memory allocation (on a stack, in a heap, or both). In fact, even in languages in which memory allocation is static, the locations used by a relocatable program fragment are not known until it has been linked with the rest of the program. Clearly, we need to find some way to express the extent of an effect in *symbolic* form.

2.4.2 Variable Names

One way to express the extent of effects without direct reference to memory locations is to express it in terms of *variable names*. This method has several advantages: not only are the variables of a program known statically (in most languages), but they are also familiar to the programmer, and therefore more suitable for inclusion in program specifications. Using this technique, the effect of each expression is expressed as a combination of zero or more effects of the form (ALLOC x), (READ x) and (WRITE x), where x ranges over variable names. For example, an expression that increments the value of the variable x_0 would have the effect $\{(\text{READ } x_0), (\text{WRITE } x_0)\}$.

Although this approach is commonly used in conventional effect analysis, it is less appropriate as a basis of a type and effect system, for the following reasons.

First, a program may contain several variables with the same name. If so, the use of variable names in effect specifications may suggest that two expressions interfere, say through a variable named **x**, when they actually refer to different variables that happen to have the same name. In conventional effect analysis, this problem can be avoided by means of systematic renaming. However, in a language with a type and effect system, where the effect

specifications are visible to the programmer, such renaming is undesirable since it goes against the programmer's choice of variable names.

Even if no two variables have the same name, each variable may in general have many instances. For example, the local variables of a subroutine are instantiated each time the subroutine is called. In a language that supports either recursive subroutine calls or closures, there can be arbitrarily many instances of the same variable, and the compiler may conclude that two expressions interfere, say through a variable **x**, when they actually refer to different locations that happen to be instances of the same variable. This problem cannot be solved by statically renaming the variables in the source program.

Furthermore, if the language treats locations as first-class values, as is the case in most languages that support heap allocation, then there is no fixed correspondence between locations and variable names. For example, a location that is accessible through the variable **x** when it is allocated, can subsequently be assigned to another variable, say **y**. The lack of such a fixed correspondence also introduces the problem of aliasing.

Based on these considerations, we have decided not to rely on variable names to express the extent of an effect.

2.4.3 Types

To permit a meaningful characterization of the extent of effects on objects that are dynamically allocated in a heap, we considered expressing the extent of effects in terms of data types. For example, the operators for dealing with arrays could have effect specifications such as (ALLOC `array`), (READ `array`) and (WRITE `array`). This method can deal with heap-allocated objects, unlike the methods discussed above, because the types of these objects, unlike their locations and their names, are known statically and remain constant throughout their lifetime.

In view of the objectives listed at the beginning of this chapter, type names could be an attractive way of expressing the extent of effects, provided that (i) the programmer can easily instantiate a given type under different names, to indicate how the values of the resulting types are used, and provided that (ii) the method apply equally well to built-in and programmer-defined types.

We found that the second objective cannot be met using the conventional notion of abstract types. Consider, for example, a data abstraction for tables in which each table is internally represented as an array. We would like the operators of this abstract type to have effect specifications of the form (WRITE `table`) and so forth, even though they are actually implemented in terms of arrays. In order to verify that operations on tables do not interfere with operations on arrays and vice versa, it is necessary to distinguish between those arrays that are used as arrays and those that are used as tables. Unfortunately, no programming language that we know of can prevent the implementation of an abstract data type from coercing values of the representation type to the abstract type and vice versa. Such a coercion is said to

expose the representation type, since subsequent modifications of the value of the representation type may be visible in some value of the abstract data type and vice versa.

Since conventional type systems are concerned only with verifying that types are equivalent, and never that they are *not* equivalent, exposure of the representation of an abstract type does not constitute a type loophole in a conventional type system. If, on the other hand, the extent of effects is expressed in terms of types, the compiler needs to be able to verify that types are *disjoint*. In this case, exposure of the representation of an abstract type is not simply a programming practice to be discouraged, but actually constitutes a loophole that renders effect analysis based upon type names unsound.

2.4.4 Regions

Having found neither locations, nor variables, nor types to be a suitable vehicle for expressing the extent of an effect, we decided to introduce a new concept specifically for this purpose. In this section we show how to express the extent of an effect in terms of *regions*.

A region corresponds to a set of writable memory locations. Using regions, the effect of each expression is expressed as a combination of zero or more effects of the form (ALLOC ρ), (READ ρ) and (WRITE ρ), where ρ ('rho') ranges over regions. For example, an expression that increments the value of a location in the region ρ_0 would have the effect $\{(\text{READ } \rho_0), (\text{WRITE } \rho_0)\}$. It is our intention to leave it up to the programmer to specify in which region any given memory location should be allocated.

Regions are orthogonal to existing programming language features: they are intended exclusively for the purpose of describing the extent of effects, and are not tied to any existing programming language concepts such as variable names or types, nor to implementation considerations such as storage allocation. In particular, a programmer can use regions to group together a set of locations regardless of when, where, what for, or under what name they are allocated. Regions subsume all previous ways of expressing the extent of effects, and then some:

- coarse-grain effect specifications can be simulated by allocating all locations in the same region;
- effect specification in terms of locations can be simulated by allocating each location in a separate region;
- effect specification in terms of variable names can be simulated by defining a separate region for each variable;
- effect specification in terms of data types can be simulated by defining a separate region for each type;
- effect specification in terms of named fields of data structures [Steele78, p. 48] can be simulated by defining a separate region for each field; and

- effect specification in terms of named properties on property lists [Mar83, p. 82] can be simulated by defining a separate region for each property.

It is left up to the programmer to decide in which region each memory location should be allocated. For example, a programmer who does not wish to reveal the locations accessed by a given abstraction, the names of the local variables of a subroutine, or the representation of an abstract data type, can allocate all locations, variables, or instances of the abstract data type in a single global region. At the same time, a programmer who wishes to document the fact that computations involving certain data structures can safely be evaluated out of order or concurrently can allocate those data structures in distinct regions.

In a language with a type and effect system based on regions, a location has two attributes: the type of value that may be stored in the location, and the region to which the location belongs. This information is introduced, maintained, and utilized as follows:

- when allocating a new location, the programmer must specify both the desired type and the region to which the location should belong;
- both of these attributes, the region and the type, are reflected in the *type* of the resulting location;
- the compiler uses this information to determine the effect specification of any expression that subsequently reads or writes the location.

In a more general language, region information is not only found in the types of locations, but also in the types of more complicated mutable values such as arrays and lists, as well as in programmer-defined abstract types.

Much of the power of regions is due to the fact that they are orthogonal to existing programming language concepts. However, this very property also leads to their two main disadvantages.

- Since regions are not tied to variable names or data types, the burden of indicating which location belongs to which region appears to fall squarely on the programmer. Fortunately, it appears that a compiler can assume much of this burden, using techniques borrowed from type inference, possibly in combination with sensible defaults.
- Because region information must be communicated across subroutine interfaces, the types of locations, arrays, lists and other mutable data structures must carry region information. This complication of the type system is certainly the most visible aspect of a type and effect system based on regions, and probably the most controversial.

We believe that on balance, the extra expressive power of a type and effect system based on regions is worth the additional complexity.

2.5 Types, Effects, and Regions

As it turns out, types, effects and regions have a lot in common. All three are, in some sense, descriptions of computational entities: types describe expressions and the resulting values, effects describe expressions and the corresponding computations, and regions describe mutable values and the corresponding memory locations. Types, effects and regions are closely interrelated. In particular,

- the type of a location is expressed in terms of the type of its contents and the region to which the location belongs, and
- the type of a subroutine is expressed in terms of its argument type, its return type, and its latent effect.

Moreover, effects themselves are expressed in terms of the regions that indicate their extent.

The similarity between types, effects and regions is probably best illustrated by the fact that all three can be regarded as *base kinds* in a higher-order, *kinded* type system along the lines of the type system proposed by McCracken [McC79]. McCracken showed how to generalize the familiar concepts of *type* abstraction and *type* polymorphism [Rey74] to permit abstraction and polymorphism over a larger class of descriptions, including descriptions that map types to types.

We propose to regard types, effects and regions as base kinds. This makes it possible to abstract types and expressions over types, effects, and regions, and hence introduces the notions of *effect* and *region* abstraction and *effect* and *region* polymorphism. For example, the `mapcar` subroutine [Ree86] can be abstracted over the argument type, return type, and latent effect of the mapping subroutine, and even over the regions that contain the input and output lists, if these are mutable. Moreover, the type constructors for such types as locations, arrays and lists can be parameterized not only with respect to the type of its contents, but also with respect to the region to which the corresponding locations belong.

One unexpected advantage of the analogy with the higher-order lambda-calculus is that the standard rules for higher-order abstraction guarantee that any program fragment that can be abstracted over a given region has no knowledge of any locations in that region. This property forms the basis for a set of language constructs for dealing with private, anonymous regions, which are described in Chapter 5.

The connection to the higher-order lambda-calculus demonstrates that effect and region information, like type information, can be a analyzed statically and then discarded: there is no need to keep track of effect or region information at run-time.

2.6 Related Work

2.6.1 Conventional Effect Analysis

Effect analysis is a critical aspect of the design of optimizing compilers, and there is an extensive literature on the subject. Since the availability of effect specifications affects only *interprocedural* effect analysis, we focus on the literature on that subject specifically [Ban79] [Bar78] [Wei80].

Interprocedural effect analysis generally proceeds in two steps, which can be performed either sequentially or concurrently. The first step is to analyze each subroutine and compute a (possibly parameterized) summary of its effects; the second step is to combine these summaries with information about the calling pattern between the subroutines, and to compute the true effects of each expression while taking account of the effects of subroutine calls. The second step can be regarded either as a transitive closure problem or as a graph flow problem, and algorithms of both types have been proposed [Ban79] [Bar78]. In all cases, the calling pattern must be known in advance.

Our approach to interprocedural effect analysis is very different. Because we are dealing with languages in which subroutines first-class values, we cannot assume that the calling pattern is known in advance. Instead, we have recast the problem as a *verification* problem:

- wherever a subroutine is declared, the programmer must specify its latent effect as part of the declaration;
- the compiler uses these declarations to compute the effect description of every expression and program fragment;
- these computed effect descriptions are then used to verify the declarations given by the programmer.

Since the latent effect declarations include the effects of nested and recursive subroutine calls, there is no need for a global computation that requires information about the calling pattern.

A second important difference concerns the format of the effect summaries. In a conventional compiler, these effect summaries are for internal use only, and need not be human-readable. In a language with a type and effect system, on the other hand, the effect specifications must be human-readable, and hence must be expressed in terms that are reasonably high-level and related to the programmer's view of the program. This difference explains why we do not use, for example, information such as the depth at which individual subroutines are defined [Bar78].

2.6.2 Syntactic Control of Interference

The idea of imposing syntactic restrictions on programs to simplify the detection and analysis of interference has been discussed by Reynolds in the context of a language with higher-order procedures and call-by-name semantics [Rey78] (see also [Ten83]). Reynolds proposed to characterize every expression and procedure by its support set, which is the set of variables that the expression or procedure may access. By permitting an application only if the support sets of the procedure and its argument do not overlap, all aliasing is prevented. Moreover, by permitting concurrent evaluation of two expressions only when their support sets do not overlap, determinacy is assured. The restrictions are subsequently relaxed to some extent.

Because the investigation is conducted in a call-by-name context, there is no opportunity to distinguish between types and effects: any expression, even an ordinary variable, can have an effect. Likewise, there is no way to distinguish between the support set of an expression and the support set of the value returned by that expression. Finally, there is no way to distinguish between the support set of an expression and the extent of its potential effects. Nevertheless, Reynolds' paper can be regarded as a precursor of our present research.

Several of the unresolved questions posed by Reynolds are addressed in this thesis [Rey78, p. 44]. Because we are dealing with applicative evaluation order, we are able to distinguish between the support set of an expression and the extent of its potential effects. In particular, our type and effect system can identify situations in which certain expressions are passive and can never be executed; the effects of such expressions do not affect the effect specifications of surrounding expressions.

2.6.3 Abstract Interpretation

The idea of computing aliasing and support sets using the techniques of abstract interpretation has been discussed by Neirynck *et al.* [Nei86]. Their point of departure is similar to ours, namely that "the main difficulty is to determine if a subroutine application produces side-effects" [*id.*, p. 3]. The two approaches diverge quickly, however, since Neirynck *et al.* take the view that the effects of a subroutine application may depend on the arguments of the subroutine application, if these in turn are subroutines [*id.*, p. 3]. By contrast, in our framework such a dependence arises only if the subroutine being applied is polymorphic in the latent effect of its argument.

In order to make the analysis tractable, Neirynck *et al.* confine their attention to a language in which subroutines and references are not storable and references may not be returned out of the scope in which they were created. Within these restrictions, they define a pair of semantic functions, one to indicate the effects of an expression and one to indicate its value, and a corresponding pair of abstract interpretation functions: one to indicate

the support set of the expression, and one to indicate how to compute the support set of the resulting value in the event that it is a subroutine.

The strength of the abstract interpretation approach lies in the close correspondence between the abstract interpretations and the standard semantics of the language [*id.*, p. 20]. The approach presented in this thesis can be regarded as a variant of the abstract interpretation approach. In particular, the rules for computing the type and effect of an expression can be regarded as an abstract interpretation of the expression, and there is a close correspondence between this set of rules and the semantics of the language. Moreover, the instantiation of a polymorphic type corresponds to application of an abstract interpretation function, in that it specializes a description (which may incorporate type, effect and region descriptions) with respect to some other description (which may itself be a type, effect or region).

The distinguishing feature of our approach is our reliance on the type system. Since we take the view that effect analysis should be performed strictly on the basis of effect declarations, our analysis is not hindered even when procedures and references may be passed, returned and stored as first-class values. We believe that our approach provides a basis for the study of abstract interpretation of more powerful languages than has been attempted to date.

2.6.4 Euclid's Collections

The programming language Euclid features a notion of a *collection*, which denotes a set of locations that are grouped together to assist the verifier (a person or a computer) in reasoning about the extent of effects [Lam77] [Pop77]. Collections resemble regions in that "the programmer can partition his dynamic variables and pointers into separate collections to indicate some of his knowledge about how they will be used; the verifier is assured that pointers in different collections can never point to overlapping variables" [Pop77, p. 14].

Collections differ from regions in several ways. Some of these differences are minor: for example, a collection can contain only values of a single type. This appears to be an arbitrary restriction imposed by the designers of Euclid, and is not a fundamental limitation. The more fundamental difference is that Euclid treats collections as *values*, syntactically as well as semantically. This gives rise to a number of interrelations between collections and pointers that make the language as a whole appear far from orthogonal. For example, every subroutine that accesses the values in a collection must be passed the collection itself as an extra parameter, and yet this parameter is supplied implicitly by the compiler if it is omitted by the programmer. In reality, collections, like regions, are a static concept. Implementations of Euclid must therefore rely on an ad-hoc mechanism to eliminate the overhead that would result if these extra parameters were actually passed at run-time.

We believe that our decision to make regions *descriptions* rather than *values* clearly differentiates the two approaches, and we believe that our

approach is more consistent with the fact that regions are a static concept. Moreover, we believe that our approach is more general and more powerful because of effect and region polymorphism.

2.6.5 Relation to Previous Work

We have extended the results reported in an earlier paper [Gif86] in several respects:

- Effect descriptions have been generalized from a small, fixed set of effect classes to a countably infinite set;
- The introduction of regions has permitted the treatment of locations as first-class values;
- The notion of polymorphism has been generalized from type polymorphism to region and effect polymorphism;
- The introduction of private regions has permitted the masking of provably local effects.
- The use of monitors and critical sections has permitted the integration of implicit and explicit concurrency.

If we compare the current state of our research with the issues for future research cited in the earlier paper, we find that we have actually aggravated the burden of providing explicit declarations by our decision to introduce regions into the type and effect system. However, our research in this area has reinforced our belief that a compiler can assume much of this burden, using techniques borrowed from type inference. On the other hand, the generalization of polymorphism to effects and regions has alleviated the adverse impact on software reusability, and the introduction of effect masking has alleviated the need for effect loopholes.

♣

Chapter 3. The MFX Language

3.1 Introduction

In chapters 3 through 6, we present a demonstration language with a type and effect system. We have called this language *MFX*, for Mini-FX. (*FX* is a complete programming language with a type and effect system that is currently being developed at MIT by the Programming System Research Group [Gif87].) *MFX* is a lexically scoped, higher-order, imperative language with static typing and left-to-right, applicative order semantics. In an attempt to introduce the features of *MFX* gradually, we introduce the language in three stages. In this chapter we present *MFX-1* (Mini-FX level 1), a subset of *MFX* that has side-effect operators but no constructs for dealing with private regions or explicit concurrency.

The purpose of this chapter is to present the syntax and static semantics of *MFX-1*. The dynamic semantics of the language, along with the major propositions relating the static and dynamic semantics, are presented in the next chapter.

The rest of this chapter is organized as follows. We begin by giving a general overview of the *MFX* language, concentrating on the features that all three versions of *MFX* language have in common. Next, we present the syntax of *MFX-1*, and give an informal description of its semantics. We also introduce some notation related to free and bound variables. We then give a formal definition and semantic motivation of description conversion and inclusion, and we present the static semantics of *MFX-1*. We conclude with a discussion of aliasing.

3.2 Overview

MFX is a lexically scoped, higher-order, imperative language with static typing and left-to-right, applicative order semantics. The type system of the language is based on the higher-order lambda-calculus of McCracken [McC79], which is a generalization of the second-order lambda calculus of Reynolds [Rey74]; the dynamic semantics and the parameter passing conventions are based on languages such as Scheme and CLU [Ree86] [Lis79], although they are formalized in a way that closely parallels the operational semantics of the lambda-calculus. In order to make the language into an *imperative* language, we have added three operators called NEW, GET and SET that can be used to allocate (writable) locations in a *store* and to read and write their contents. *MFX* is expression-oriented: every expression returns a value, and every expression may have side-effects.

In *MFX*, every expression has a *type description*, which describes what sort of value the expression may return, and an *effect description*, which describes what sort of side-effects the expression may have. Type descriptions may incorporate effect descriptions; in particular, the type of every subroutine incorporates a description of the *latent effect* of the subroutine. This latent effect description indicates what sort of side-effects the subroutine may have when it is applied. Effect descriptions, in turn, may incorporate *region descriptions*; these region descriptions describe the extent of the effect in question.

Types, effects and regions are collectively called *descriptions*. Every well-formed description has a *kind*, which acts as the 'type' of the description. There are only three different kinds in *MFX*: TYPE, EFFECT and REGION.

MFX consists of three levels: expressions, descriptions and kinds. Within each level, all language constructs are orthogonal. For example, any expression, regardless of its type or effect, can be abstracted over any of its free ordinary variables, regardless of its type. Similarly, any expression can be abstracted over any of its free description variables, regardless of its kind. Thus, an expression can be polymorphic with respect to any number of description variables regardless of their kinds. Finally, all values are storable, including ordinary subroutines, polymorphic subroutines and locations.

Although McCracken's language supports type functions and recursive types, we have omitted these features from *MFX* in order to simplify the presentation. However, in order to make our treatment of types and effects more uniform, we have added type inclusion (also known as subtyping). Finally, we have added constructs for conditional and sequential evaluation in order to illustrate how such control flow constructs can be treated.

In order to keep the presentation simple, we have kept the language to a bare minimum. It should be understood from the outset that the type and effect system presented here is compatible with a wide variety of built-in types, control structures, and type constructors. In Chapter 7 we show how to integrate some of these features into the language.

3.3 Syntax

In this section we present the syntax of *MFX-1*. The language consists of three levels: kinds, descriptions and expressions. The names of syntactic classes are written in a *Slanted Typeface*; reserved words are written in SMALL CAPITALS. The meta-variable for each syntactic class is shown in parentheses. The superscripts * and $^+$ stand for *zero or more* and *one or more* repetitions respectively.

3.3.1 Kinds

Kinds act as the types of descriptions. There are three kind constants: REGION, EFFECT and TYPE.

Kind =		– kinds (κ)
	REGION	– the kind of regions
	EFFECT	– the kind of effects
	TYPE	– the kind of types

3.3.2 Descriptions

Descriptions describe the types and effects of expressions. The simplest sort of description is a description variable. There is an infinite set of descriptions variables:

Dvar =	$\{d_1, d_2, \ldots\}$	– description variables (d)

A description variable may have any kind.

We now present the three kinds of descriptions (regions, effects, and types) in turn.

Region descriptions correspond to countably infinite sets of locations (locations are defined in Chapter 4). There is an infinite set of region constants, which denote disjoint sets of locations:

Rconst =	$\{r_1, r_2, \ldots\}$	– region constants (r)

The grammar of region descriptions in general is given below. A region description is one of the following: a region constant, a region variable, or the union of one or more region descriptions.

Region =		– region descriptions (ρ)
	Rconst	– region constant
	Dvar	– region variable
	(UNION *Region*$^+$)	– union of one or more regions

Since a region description must correspond to an *infinite* set of locations, the union of zero region descriptions is not a valid region description.

Effect descriptions correspond to countable sets of *store operations*, which can be thought of as pairs of the form $\langle sort, loc \rangle$, where *sort* is an *effect constructor* (one of ALLOC, READ, and WRITE) and *loc* is a location. There is one effect constant, which corresponds to the *empty set*:

Econst =	PURE	– the constant denoting "no effect"

The grammar of effect descriptions in general is given below. An effect description is one of the following: an effect constant, an effect variable, a combination of an effect constructor and a region description, or a combination of zero or more effect descriptions.

Effect =		– effect descriptions (ϵ)
	Econst	– the effect constant PURE
	Dvar	– effect variable
	(ALLOC *Region*)	– effect of allocating in a given region
	(READ *Region*)	– effect of reading from a given region
	(WRITE *Region*)	– effect of writing to a given region
	(MAXEFF *Effect**)	– combination of zero or more effects

A combination of zero effect descriptions is itself a valid effect description; in fact, the effect constant PURE and the empty combination "(MAXEFF)" are synonymous. Note that a combination of effects is a set union; the reserved word MAXEFF is somewhat misleading.

Type descriptions correspond to countable sets of *values*, which we view as syntactic entities. There are two type constants, the meaning of which will be defined shortly:

Tconst = {UNIT, BOOL}	– type constants (t)

The grammar of type descriptions in general is given below. A type description is one of the following: a type constant, a type variable, an ordinary subroutine type, a polymorphic subroutine type, or a reference type.

Type =	– type descriptions (τ)
Tconst	– type constant
Dvar	– type variable
(SUBR (*Type*) *Effect Type*)	– types of ordinary subroutines
(POLY (*Dvar*:*Kind*) *Effect Type*)	– types of polymorphic subroutines
(REF *Region Type*)	– types of locations

The type description (SUBR (τ_1) ϵ τ_2) is a generalization of the type "$\tau_1 \rightarrow \tau_2$" in the typed lambda-calculus. The effect description ϵ represents the latent effect of the subroutine. The parentheses around the formal parameter type are present only to simplify a generalization to subroutines of multiple arguments.

The type description (POLY (d:κ) ϵ τ) is a generalization of the type "$\Delta t{:}\kappa.\tau$" in the higher-order lambda-calculus of McCracken [McC83], which in turn is a generalization of the type "$\Delta t.\tau$" in the second-order lambda-calculus of Reynolds [Rey74]. The effect description ϵ represents the latent effect.

The type description (REF ρ τ) is a generalization of the type "*ref* τ" in programming languages such as ML [Gor79a].

The grammar of descriptions in general is given below. A description is one of the following: a region description, an effect description, or a type description.

Desc =		– descriptions (δ)
	Region	– region descriptions
	Effect	– effect descriptions
	Type	– type descriptions

3.3.3 Expressions

Expressions correspond to programs. The simplest sort of expression is an ordinary variable. There is an infinite set of ordinary variables:

Var =	$\{x_1, x_2, \ldots\}$	– ordinary variables (x)

An ordinary variable may have any type.

There are three ordinary constants, all of which have specific types: NIL has type UNIT, and TRUE and FALSE have type BOOL.

Unit =	{NIL}	– the unit type
Bool =	{TRUE, FALSE}	– the Booleans (b)

Const =		– ordinary constants (c)
	Unit	– the unit type
	Bool	– the Booleans

The grammar for expressions in general is given below. There are three general classes of expressions: expressions that come from the higher-order lambda-calculus; expression that deal with evaluation order; and expressions that deal with side-effects. The first class consists of constants and variables, ordinary abstraction and application, and polymorphic abstraction and application. The second class consists of expressions for conditional and sequential evaluation. The third class consists of expressions for the allocating, reading, and writing of locations.

Exp =		– expressions (e)
	Const	– ordinary constant
	Var	– ordinary variable
	(LAMBDA (*Var*:*Type*) *Exp*)	– ordinary abstraction
	(*Exp Exp*)	– ordinary application
	(PLAMBDA (*Dvar*:*Kind*) *Exp*)	– polymorphic abstraction
	(PROJ *Exp Desc*)	– polymorphic application
	(IF *Exp Exp Exp*)	– conditional evaluation
	(BEGIN Exp^+)	– sequential evaluation

(NEW *Region Type Exp*)	– allocating a location
(GET *Exp*)	– reading a location
(SET *Exp Exp*)	– writing a location

Certain expressions, such as applications, represent *computations*; other expressions, such as constants, do not exhibit any computational behavior, and represent *values*.

Definition. An expression is a *value* iff it is a constant, a LAMBDA expression, or a PLAMBDA expression. We use *Val* to denote the set of values and v to denote individual values. In other words,

Val =	– values (v)
Const	– ordinary constant
(LAMBDA (*Var*:*Type*) *Exp*)	– ordinary abstraction
(PLAMBDA (*Dvar*:*Kind*) *Exp*)	– polymorphic abstraction

3.3.4 Informal Semantics

We have already informally described the semantics of regions, effects and types; we now sketch the semantics of the different expressions.

Informally, the semantics of ordinary abstraction and application are as follows. The expression (LAMBDA (x:τ) e) returns an ordinary subroutine with formal parameter x of type τ and with body e. The ordinary application (e_1 e_2) has the following semantics. First, the expressions e_1 and e_2 are evaluated, in that order. Next, provided that e_1 returns an ordinary subroutine, the body of the subroutine is evaluated with the formal parameter of the subroutine bound to the value of e_2, and its value returned.

For example, the expression

```
(LAMBDA (x:BOOL)
  (IF x FALSE TRUE))
```

denotes a subroutine that returns the negation of its argument, which must be of type BOOL. Since this subroutine has argument type BOOL, latent effect PURE and return type BOOL, the expression has type

```
(SUBR (BOOL) PURE BOOL)
```

Expressions may have free variables, provided that they are defined in the surrounding scope. For example, if `int` is declared in the surrounding scope as a type and `e` as an effect, and `f` is declared as a subroutine that takes an argument of type `int` and returns a value of type `int`, then the ordinary subroutine

```
(LAMBDA (x:int)
  (f (f x)))
```

is equivalent to the self-composition of the subroutine `f`.

Note that only the argument type of the subroutine, `int`, is manifest in this expression; the latent effect of the subroutine and its return type must be inferred from the body of the subroutine and the types of its free variables. In this case the return type is `int`, and if the latent effect of `f` is `e` then the latent effect of the expression is (MAXEFF `e e`), which is equivalent to `e`.

The semantics of polymorphic abstraction and application are as follows. The expression (PLAMBDA (d:κ) e) returns a polymorphic subroutine with formal parameter d of kind κ and with body e. The polymorphic application (PROJ e δ) has the following semantics. First, the expression e is evaluated. Next, provided that e returns a polymorphic subroutine, the body of the subroutine is evaluated with the formal parameter of the subroutine bound to δ, and its value returned.

For example, the expression

```
(PLAMBDA (t:TYPE)
  (LAMBDA (x:t)
    x))
```

denotes the polymorphic identity function; it has type

```
(POLY (t:TYPE) PURE
  (SUBR (t) PURE
    t))
```

Expressions may be polymorphic not only in types, but also in effects and regions. For example, the expression

```
(PLAMBDA (t:TYPE)
  (PLAMBDA (e:EFFECT)
    (LAMBDA (f:(SUBR (t) e t))
      (LAMBDA (x:t)
        (f (f x))))))
```

denotes the polymorphic self-composition functional, polymorphic in the argument type and the latent effect of its argument. This expression has the following type:

```
(POLY (t:TYPE) PURE
  (POLY (e:EFFECT) PURE
    (SUBR ((SUBR (t) e t)) PURE
      (SUBR (t) (MAXEFF e e)
        t))))
```

where the latent effect of the innermost subroutine, (MAXEFF `e e`), is equivalent to simply `e`.

The semantics of the conditional expression (IF e_1 e_2 e_3) are as follows. First, e_1 is evaluated. Provided that it returns TRUE or FALSE, either e_2 or e_3,

respectively, is evaluated and its value returned. This expression is non-strict in the branch that is not taken: for example, if `f` is bound in the surrounding scope to a subroutine that diverges on FALSE, then the expression

```
(IF TRUE
  NIL
  (f FALSE))
```

returns NIL.

The semantics of the sequencing expression (BEGIN $e_1 \ldots e_n$) are as follows: the expressions $e_1 \ldots e_n$ are evaluated sequentially, from left to right, and the value of e_n is returned. For example, the expression

```
(BEGIN
  (IF TRUE NIL (f FALSE))
  FALSE)
```

returns FALSE.

The semantics of the expression (NEW ρ τ e) are as follows. First, e is evaluated. Next, a new writable memory location capable of holding values of type τ is allocated in the region ρ and initialized to the value of e. Finally, this location, which has type (REF ρ τ), is returned. For example, the expression

```
(PLAMBDA (r:REGION)
  (PLAMBDA (t:TYPE)
    (LAMBDA (x:t)
      (NEW r t x))))
```

emulates the NEW expression, in that it takes a region `r`, a type `t`, and a value `x`, and returns a new location of type (REF `r` `t`) that has been initialized to `x`. This expression has type

```
(POLY (r:REGION) PURE
  (POLY (t:TYPE) PURE
    (SUBR (t) (ALLOC r)
      (REF r t))))
```

The semantics of the expression (GET e) are as follows. First, e is evaluated. Provided that e returns a location that contains a value, the expression as a whole returns that value. For example, the expression

```
((LAMBDA (x:(REF r1 BOOL))
   (GET x))
 (NEW r1 BOOL TRUE))
```

returns TRUE. This expression has type BOOL and effect (MAXEFF (ALLOC r1) (READ r1)).

Finally, the semantics of the expression (SET e_1 e_2) are as follows. First, e_1 and e_2 are evaluated, in that order. Next, provided that e_1 returns a location, the value of e_2 is written to this location. The expression as a whole returns the value NIL. For example, the expression

```
((LAMBDA (x:(REF r1 BOOL))
   (BEGIN
     (SET x FALSE))
     (GET x))
 (NEW r1 BOOL TRUE))
```

returns FALSE. This expression has type BOOL and effect (MAXEFF (ALLOC r1) (WRITE r1) (READ r1)).

3.3.5 Free Variables

To illustrate the scoping rules of the language, we present the rules for determining the free (ordinary and description) variables of an expression or description. Note that these rules are the same as in the higher-order typed lambda-calculus; in particular, LAMBDA binds ordinary variables, PLAMBDA and POLY bind description variables, and all other constructs simply pass on the free variables of their components. The scoping rules are formalized in the following definitions.

Definition. The free ordinary variables of an expression are given by the function $FV : Exp \to \mathcal{P}ow(Var)$ defined below.

$$\begin{aligned}
FV(c) &= \emptyset \\
FV(x) &= \{x\} \\
FV(\text{LAMBDA } (x{:}\tau)\ e) &= FV(e) - \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\text{PLAMBDA } (d{:}\kappa)\ e) &= FV(e) \\
FV(\text{PROJ } e\ \delta) &= FV(e) \\
FV(\text{IF } e_1\ e_2\ e_3) &= FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(\text{BEGIN } e_1 \ldots e_n) &= FV(e_1) \cup \ldots \cup FV(e_n) \\
FV(\text{NEW } \rho\ \tau\ e) &= FV(e) \\
FV(\text{GET } e) &= FV(e) \\
FV(\text{SET } e_1\ e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}$$

Definition. The free description variables of a description are given by

the function $FDV_{desc} : Desc \to \mathcal{P}ow(Dvar)$ defined below.

$$
\begin{aligned}
FDV_{desc}(d) &= \{d\} \\
FDV_{desc}(r) &= \emptyset \\
FDV_{desc}(\text{UNION } \rho_1 \ldots \rho_n) &= FDV_{desc}(\rho_1) \cup \ldots \cup FDV_{desc}(\rho_n) \\
FDV_{desc}(\text{ALLOC } \rho) &= FDV_{desc}(\rho) \\
FDV_{desc}(\text{READ } \rho) &= FDV_{desc}(\rho) \\
FDV_{desc}(\text{WRITE } \rho) &= FDV_{desc}(\rho) \\
FDV_{desc}(\text{MAXEFF } \epsilon_1 \ldots \epsilon_n) &= FDV_{desc}(\epsilon_1) \cup \ldots \cup FDV_{desc}(\epsilon_n) \\
FDV_{desc}(\text{SUBR } (\tau_1) \ \epsilon \ \tau_2) &= FDV_{desc}(\tau_1) \cup FDV_{desc}(\epsilon) \cup FDV_{desc}(\tau_2) \\
FDV_{desc}(\text{POLY } (d{:}\kappa) \ \epsilon \ \tau) &= FDV_{desc}(\epsilon) \cup FDV_{desc}(\tau) \ - \ \{d\} \\
FDV_{desc}(\text{REF } \rho \ \tau) &= FDV_{desc}(\rho) \cup FDV_{desc}(\tau)
\end{aligned}
$$

Definition. The free description variables of an expression are given by the function $FDV_{exp} : Exp \to \mathcal{P}ow(Dvar)$ defined below.

$$
\begin{aligned}
FDV_{exp}(c) &= \emptyset \\
FDV_{exp}(x) &= \emptyset \\
FDV_{exp}(\text{LAMBDA } (x{:}\tau) \ e) &= FDV_{desc}(\tau) \cup FDV_{exp}(e) \\
FDV_{exp}(e_1 \ e_2) &= FDV_{exp}(e_1) \cup FDV_{exp}(e_2) \\
FDV_{exp}(\text{PLAMBDA } (d{:}\kappa) \ e) &= FDV_{exp}(e) - \{d\} \\
FDV_{exp}(\text{PROJ } e \ \delta) &= FDV_{exp}(e) \cup FDV_{desc}(\delta) \\
FDV_{exp}(\text{IF } e_1 \ e_2 \ e_3) &= FDV_{exp}(e_1) \cup FDV_{exp}(e_2) \cup FDV_{exp}(e_3) \\
FDV_{exp}(\text{BEGIN } e_1 \ldots e_n) &= FDV_{exp}(e_1) \cup \ldots \cup FDV_{exp}(e_n) \\
FDV_{exp}(\text{NEW } \rho \ \tau \ e) &= FDV_{desc}(\rho) \cup FDV_{desc}(\tau) \cup FDV_{exp}(e) \\
FDV_{exp}(\text{GET } e) &= FDV_{exp}(e) \\
FDV_{exp}(\text{SET } e_1 \ e_2) &= FDV_{exp}(e_1) \cup FDV_{exp}(e_2)
\end{aligned}
$$

We adopt the usual conventions for alpha-renaming and beta-substitution. We use the notation $\delta[\delta'/d]$, $e[\delta'/d]$ and $e[e'/x]$ to indicate substitution, where bound variables are renamed as needed to avoid capture. We adopt the usual definition of closed descriptions and expressions:

Definition. A description δ is *closed* iff it has no free description variables, *i.e.* iff $FDV_{desc}(\delta) = \emptyset$.

Definition. An expression e is *closed* iff it has no free ordinary variables and no free description variables, *i.e.* iff $FV(e) = \emptyset \ \wedge \ FDV_{exp}(e) = \emptyset$.

3.3.6 Free Constants

It is convenient to define the free region constants of a description, $FRC_{desc}(\delta)$, and of an expression, $FRC_{exp}(e)$. Since region constants cannot be bound, this definition is trivial: all the region constants that occur in a description or expression are free. We give a more precise syntactic definition anyway, in order to highlight the differences between $FRC_{exp}(e)$ and $FDV_{exp}(e)$.

Definition. The free region constants of a description are given by the function $FRC_{desc} : Desc \rightarrow \mathcal{P}ow(Rconst)$ defined below.

$$\begin{aligned} FRC_{desc}(d) &= \emptyset \\ FRC_{desc}(r) &= \{r\} \\ FRC_{desc}(\text{POLY } (d{:}\kappa)\ \epsilon\ \tau) &= FRC_{desc}(\epsilon) \cup FRC_{desc}(\tau) \end{aligned}$$

The remaining clauses are analogous to the corresponding clauses in the definition of $FDV_{desc}(\delta)$.

Definition. The free region constants of an expression are given by the function $FRC_{exp} : Exp \rightarrow \mathcal{P}ow(Rconst)$ defined below.

$$\begin{aligned} FRC_{exp}(\text{NIL}) &= \emptyset \\ FRC_{exp}(b) &= \emptyset \\ FRC_{exp}(x) &= \emptyset \\ FRC_{exp}(\text{PLAMBDA } (d{:}\kappa)\ e) &= FRC_{exp}(e) \end{aligned}$$

The remaining clauses are analogous to the corresponding clauses in the definition of $FDV_{exp}(e)$.

3.4 Description Conversion and Inclusion

Every description corresponds to a set: a region description corresponds to a set of locations, an effect description corresponds to a set of store operations, and a type description corresponds to a set of values.

In general, two distinct descriptions may correspond to the same set: for example, the region descriptions

```
(UNION r1 r2)
```

and

```
(UNION r2 r1)
```

correspond to the same set of locations. Similarly, the effect descriptions

```
(WRITE (UNION r1 r2))
```

and

```
(MAXEFF (WRITE r1) (WRITE r2))
```

correspond to the same set of store operations. Finally, the type descriptions

```
(POLY (t:TYPE) PURE
  (SUBR (t) PURE
    (SUBR (t) PURE
      BOOL)))
```

and

```
(POLY (u:TYPE) PURE
  (SUBR (u) PURE
    (SUBR (u) PURE
      BOOL)))
```

correspond to the same set of values.

We would like to say that two descriptions δ_1 and δ_2 are *convertible* iff they correspond to the same set of locations, store operations, or values. Unfortunately, there are many type descriptions that correspond to the empty set, such as

```
(POLY (t:TYPE) PURE
  t)
```

and

```
(POLY (t:TYPE) PURE
  (SUBR ((SUBR (t) PURE t)) PURE
    t))
```

and it makes little sense to regard all such type descriptions as convertible. Aside from empty types, however, we say that two descriptions are convertible iff they correspond to the same set of locations, store operations, or values, and we write $\delta_1 \simeq \delta_2$. Empty types may be convertible to other empty types, but not to nonempty types. With the same proviso for empty types, we say that δ_1 is a *subregion*, *subeffect* or *subtype* of δ_2, depending on the kinds of δ_1 and δ_2 (which must be the same), iff the set corresponding to the description δ_1 is a subset of the set corresponding to the description δ_2, and we write $\delta_1 \sqsubseteq \delta_2$. Empty types may be subtypes of nonempty types, but not vice versa. The conversion relation $\simeq$ is an equivalence relation; the inclusion relation $\sqsubseteq$ is a partial order.

Below we present a set of description conversion and inclusion rules for regions, effects and types, in that order. These rules constitute a complete definition of the inclusion relation; additional conversions not explicitly shown can be obtained from the identity

$$\delta_1 \simeq \delta_2 \quad \Leftrightarrow \quad (\delta_1 \sqsubseteq \delta_2) \wedge (\delta_1 \sqsupseteq \delta_2)$$

There are no conversion or inclusion rules for descriptions of different kinds, since a description is included in another only if they have the same kind.

3.4.1 Region Descriptions

A region constant or variable corresponds to a countably infinite set of locations, and the UNION of one or more region descriptions corresponds to the union of the corresponding sets of locations. It follows that every region description is uniquely characterized by its set of free region constants and variables, regardless of how they are combined, and hence that region descriptions can be flattened, *i.e.*

$$(\text{UNION } (\text{UNION } \rho_1 \ldots \rho_n)\ \rho_{n+1} \ldots \rho_m) \simeq (\text{UNION } \rho_1 \ldots \rho_m)$$
$$(\text{UNION } \rho) \simeq \rho$$

that order does not matter, *i.e.*

$$\forall \pi_n \ .\ (\text{UNION } \rho_1 \ldots \rho_n) \simeq (\text{UNION } \rho_{\pi_n(1)} \cdots \rho_{\pi_n(n)})$$

(where π_n ranges over the permutations of the integers $1 \ldots n$), and that duplicates can be eliminated, *i.e.*

$$(\text{UNION } \rho_1\ \rho_1 \ldots \rho_n) \simeq (\text{UNION } \rho_1 \ldots \rho_n)$$

This leads to the definition of region inclusion given below.

Definition. The inclusion relation $\sqsubseteq$ on region descriptions is the partial order generated by the conversion rules given above and the single inference rule given below. When $\rho \sqsubseteq \rho'$, we say that ρ is a *subregion* of ρ'.

$$\frac{\forall i\ \exists j\ .\ \rho_i \simeq \rho'_j}{(\text{UNION } \rho_1 \ldots \rho_n) \sqsubseteq (\text{UNION } \rho'_1 \ldots \rho'_m)}$$

Using the conversion rules given above, one can convert any region description to a union of one or more region constants and variables. If we regard the arguments to UNION as a set, we find that two region descriptions that have been put in this form are convertible iff they correspond to the same set. Thus, the set of *closed* region descriptions modulo conversion is isomorphic to the set of all possible nonempty combinations of region constants, *i.e.* to the set

$$\mathcal{P}ow(Rconst) - \{\emptyset\}$$

When region variables are taken into account, the set of region descriptions modulo conversion is isomorphic to the set

$$\mathcal{P}ow(Rconst \cup Dvar) - \{\emptyset\}$$

The region descriptions modulo conversion form a powerset lattice with the bottom element removed. The region constructor UNION acts as the least upper bound operator on regions.

It would be possible to add a top element to the set of regions. We have decided not to do so, in part in order to keep *MFX* simple, and in part for reasons discussed at the end of Chapter 5.

3.4.2 Effect Descriptions

An effect constant or variable corresponds to a set of store operations, and a combination of zero or more effect descriptions corresponds to the union of the corresponding sets. It follows that the effect constructors ALLOC, READ and WRITE distribute over the least upper bound operator:

$$(\text{ALLOC } (\text{UNION } \rho_1 \ldots \rho_n)) \simeq (\text{MAXEFF } (\text{ALLOC } \rho_1) \ldots (\text{ALLOC } \rho_n))$$
$$(\text{READ } (\text{UNION } \rho_1 \ldots \rho_n)) \simeq (\text{MAXEFF } (\text{READ } \rho_1) \ldots (\text{READ } \rho_n))$$
$$(\text{WRITE } (\text{UNION } \rho_1 \ldots \rho_n)) \simeq (\text{MAXEFF } (\text{WRITE } \rho_1) \ldots (\text{WRITE } \rho_n))$$

Note that the least upper bound operator on regions is denoted by the symbol UNION, whereas the least upper bound operator on effects is denoted by the symbol MAXEFF.

It also follows that effect descriptions can be flattened, *i.e.*

$$(\text{MAXEFF } (\text{MAXEFF } \epsilon_1 \ldots \epsilon_n) \ \epsilon_{n+1} \ldots \epsilon_m) \simeq (\text{MAXEFF } \epsilon_1 \ldots \epsilon_m)$$
$$(\text{MAXEFF } \epsilon) \simeq \epsilon$$

that order does not matter, *i.e.*

$$\forall \pi_n \ . \ (\text{MAXEFF } \epsilon_1 \ldots \epsilon_n) \simeq (\text{MAXEFF } \epsilon_{\pi_n(1)} \ldots \epsilon_{\pi_n(n)})$$

(where π_n ranges over the permutations of the integers $1 \ldots n$), and that duplicates can be eliminated, *i.e.*

$$(\text{MAXEFF } \epsilon_1 \ \epsilon_1 \ldots \epsilon_n) \simeq (\text{MAXEFF } \epsilon_1 \ldots \epsilon_n)$$

This leads to the definition of effect inclusion given below.

Definition. The inclusion relation $\sqsubseteq$ on effect descriptions is the partial order generated by the conversion rules given above and the single inference rule given below. When $\epsilon \sqsubseteq \epsilon'$, we say that ϵ is a *subeffect* of ϵ'.

$$\frac{\forall i \ \exists j \ . \ \epsilon_i \simeq \epsilon'_j}{(\text{MAXEFF } \epsilon_1 \ldots \epsilon_n) \sqsubseteq (\text{MAXEFF } \epsilon'_1 \ldots \epsilon'_m)}$$

This definition gives rise to the following derived rule:

$$\frac{\rho \sqsubseteq \rho'}{\begin{array}{c}(\text{ALLOC } \rho) \sqsubseteq (\text{ALLOC } \rho') \\ (\text{READ } \rho) \sqsubseteq (\text{READ } \rho') \\ (\text{WRITE } \rho) \sqsubseteq (\text{WRITE } \rho')\end{array}}$$

Using the conversion rules given above, one can convert any effect description to a combination of zero or more effect variables and zero or more effect descriptions of the form (*sort extent*), where *sort* is a effect constructor

and *extent* is a region constant or variable. If we regard the arguments to MAXEFF as a set, we find that two effect descriptions that have been put in this form are convertible iff they correspond to the same set.

Thus, the set of *closed* effect descriptions modulo conversion is isomorphic to the set of all possible combinations of pairs of effect constructors and region constants, *i.e.* to the set

$$\mathcal{P}ow(\{\text{ALLOC}, \text{READ}, \text{WRITE}\} \times Rconst)$$

When region variables and effect variables are taken into account, the set of effect descriptions modulo conversion is isomorphic to the set

$$\mathcal{P}ow((\{\text{ALLOC}, \text{READ}, \text{WRITE}\} \times (Rconst \cup Dvar)) \cup Dvar)$$

The effect descriptions modulo conversion form a powerset lattice with PURE, the synonym for "(MAXEFF)", at the bottom. The effect constructor MAXEFF acts as the least upper bound operator on effects.

It would be possible to add a top element to the set of effects. We have decided not to do so, in part in order to keep *MFX* simple, and in part for reasons discussed at the end of Chapter 5.

3.4.3 Type Descriptions

A type corresponds to a countable set of values, which we regard as syntactic entities. In type descriptions that have bound variables, the choice of bound variables is immaterial. This is illustrated by the rule for alpha-conversion, where $[d'/d]$ indicate substitution of d' for all free instances of d:

$$(\text{POLY } (d{:}\kappa)\ \tau) \simeq (\text{POLY } (d'{:}\kappa)\ \tau[d'/d]) \quad (d' \notin FDV_{desc}(\tau))$$

Unlike the region and effect descriptions, the type descriptions do not have rules for reordering, flattening and duplicate elimination. Indeed, the type conversion and type inclusion rules for the various type constructors are not even symmetric, nor are they in all cases monotonic. The formal type inclusion rules are given below; an informal explanation with examples follows.

Definition. The inclusion relation $\sqsubseteq$ on type descriptions is the partial order generated by the conversion rule given above and the inference rules given below. When $\tau \sqsubseteq \tau'$, we say that τ is a subtype of τ'.

The type inclusion inference rule for the type constructor SUBR reflects the fact that SUBR is *monotonic* in its effect and return type components, but *anti-monotonic* in its parameter type component. This is consistent with its interpretation as corresponding to a set of subroutines [Mit84, p 272].

$$\frac{\tau_1 \sqsupseteq \tau_1' \quad \wedge \quad \epsilon \sqsubseteq \epsilon' \quad \wedge \quad \tau_2 \sqsubseteq \tau_2'}{(\text{SUBR } (\tau_1)\ \epsilon\ \tau_2) \sqsubseteq (\text{SUBR } (\tau_1')\ \epsilon'\ \tau_2')}$$

The rule for the type constructor POLY reflects the fact that POLY is monotonic in its effect and return type components. This is consistent with its interpretation as corresponding to a set of polymorphic expressions [*id.*].

$$\frac{\epsilon \sqsubseteq \epsilon' \quad \wedge \quad \tau \sqsubseteq \tau'}{(\text{POLY } (d{:}\kappa)\ \epsilon\ \tau) \sqsubseteq (\text{POLY } (d{:}\kappa)\ \epsilon'\ \tau')}$$

The rule for the type constructor REF reflects the fact that REF is monotonic in its region component but neither monotonic nor anti-monotonic in its type component. This is consistent with its interpretation as corresponding to a set of (writable) memory locations.

$$\frac{\rho \sqsubseteq \rho' \quad \wedge \quad \tau \simeq \tau'}{(\text{REF } \rho\ \tau) \sqsubseteq (\text{REF } \rho'\ \tau')}$$

To see why REF is monotonic in its region component but neither monotonic nor anti-monotonic in its type component, consider the following argument. Conceptually, a location is equivalent to a pair of subroutines, one for reading and one for writing. Thus, the type (REF ρ τ) is in some sense equivalent to a *pair* of types, (SUBR (UNIT) (READ ρ) τ) and (SUBR (τ) (WRITE ρ) UNIT). Since ρ appears as the latent effect in both cases, (REF ρ τ) is a subtype of (REF ρ' τ) whenever $\rho \sqsubseteq \rho'$. However, since τ appears as the return type in the first type and as the parameter type in the second type, (REF ρ τ) is a subtype of (REF ρ τ') only if $\tau \sqsubseteq \tau'$ *and* $\tau \sqsupseteq \tau'$, *i.e.* only if $\tau \simeq \tau'$. It follows that (REF ρ τ) is neither monotonic nor anti-monotonic in τ.

The type descriptions modulo conversion do not form a lattice, because the set is not closed under $\sqcup$ and $\sqcap$. For example, there is no type description τ such that UNIT $\sqsubseteq \tau$ and BOOL $\sqsubseteq \tau$; similarly, there is no type description τ' such that $\tau' \sqsubseteq$ UNIT and $\tau' \sqsubseteq$ BOOL. However, the partial order $\sqsubseteq$ on type descriptions does have the property that any two type descriptions that have an upper bound also have a least upper bound, and any two type descriptions that have a lower bound also have a greatest lower bound.

It would be possible to add top and bottom elements to the set of types, but we have decided not to do so in order to keep *MFX* simple.

We conclude this section with a few examples that illustrate the type inclusion rules. If `int` is a subtype of `real` and `r` is a region description, then

```
(SUBR (real) PURE
  int)
```

is a subtype of

```
(SUBR (int) (READ r)
  real)
```

Similarly, if r is a region description, then

```
(POLY (t:TYPE) PURE
  (SUBR ((REF r t)) PURE
    (REF r t)))
```

is a subtype of

```
(POLY (u:TYPE) (ALLOC r)
  (SUBR ((REF r u)) (READ r)
    (REF r u)))
```

Finally, the type inference rule for REF types is illustrated by the following examples: if int is a subtype of real and r1 and r2 are region descriptions, then the type description

```
(REF r1 int)
```

is a subtype of the type description

```
(REF (UNION r1 r2) int)
```

but *not* of the type description

```
(REF r1 real)
```

3.5 Static Semantics

The grammar of the language, which was presented in the first section of this chapter, defines the syntax of kinds, descriptions and expressions. However, not all syntactically correct descriptions or expressions make sense: for example, the following expressions are meaningless, since TRUE is not a subroutine, NIL is not a Boolean, and FALSE is not a location.

```
(TRUE NIL)
(IF NIL TRUE FALSE)
(GET FALSE)
```

In order to define what descriptions and expressions do make sense, we introduce the notion of *well-formed* descriptions and expressions. Informally, a description is well-formed iff it can be assigned a kind, and an expression is well-formed iff it can be assigned a type.

Below, we give a formal definition of what we mean by a well-formed description or expression. For descriptions, this definition is expressed in the form of a set of kind axioms and kind inference rules. For expressions, the definition is expressed in the form of a set of type axioms and type inference rules.

Every well-formed expression has not only a type, but also an effect. The effect of an expression is determined by a set of effect axioms and effect inference rules. In order to emphasize the interplay between types and effects, we present the effect inference rules together with the type inference rules.

3.5.1 Kind Inference

Because a description may have free variables, the kind of a description is defined in the context of a *kind assignment* , which is a partial function $B : Dvar \rightarrow Kind$ that maps description variables to their kinds.

Definition. A description δ *has kind* κ with respect to the kind assignment B iff the formula

$$B \vdash \delta : \kappa$$

can be derived using the axioms and inference rules given below.

We write $\delta : \kappa$ when $B \vdash \delta : \kappa$ for all B.

The kind axioms, below, give the kinds of the description constants and variables.

$$r : \text{REGION}$$
$$t : \text{TYPE}$$
$$B \vdash d : B(d)$$

The sole kind inference rule for region descriptions (shown below) states that the UNION of one or more descriptions of kind REGION also has kind REGION.

$$\frac{\forall i, 1 \leq i \leq n \;.\; B \vdash \rho_i : \text{REGION}}{B \vdash (\text{UNION } \rho_1 \ldots \rho_n) : \text{REGION}}$$

There are two kind inference rules for effect descriptions. The first rule states that if ρ has kind REGION, then the effect descriptions (ALLOC ρ), (READ ρ) and (WRITE ρ) all have kind EFFECT.

$$\frac{B \vdash \rho : \text{REGION}}{\begin{array}{c} B \vdash (\text{ALLOC } \rho) : \text{EFFECT} \\ B \vdash (\text{READ } \rho) : \text{EFFECT} \\ B \vdash (\text{WRITE } \rho) : \text{EFFECT} \end{array}}$$

The second rule states that the MAXEFF of zero or more descriptions of kind EFFECT also has kind EFFECT. In particular, the effect description (MAXEFF) or PURE has kind EFFECT.

$$\frac{\forall i, 1 \leq i \leq n \;.\; B \vdash \epsilon_i : \text{EFFECT}}{B \vdash (\text{MAXEFF } \epsilon_1 \ldots \epsilon_n) : \text{EFFECT}}$$

Finally, there are three kind inference rules for type descriptions. The first two rules are direct adaptations of the corresponding rules in the higher-order typed lambda-calculus, changed only to handle the effect component of the SUBR and POLY type descriptions. We write $B[d \leftarrow \kappa]$ to denote the kind assignment obtained by patching B at d so that $B[d \leftarrow \kappa](d) = \kappa$.

$$\frac{\begin{array}{c} B \vdash \tau_1 : \text{TYPE} \\ B \vdash \epsilon : \text{EFFECT} \\ B \vdash \tau_2 : \text{TYPE} \end{array}}{B \vdash (\text{SUBR } (\tau_1) \; \epsilon \; \tau_2) : \text{TYPE}}$$

$$\frac{B[d \leftarrow \kappa] \vdash \tau : \text{TYPE} \quad B[d \leftarrow \kappa] \vdash \epsilon : \text{EFFECT}}{B \vdash (\text{POLY } (d{:}\kappa)\ \epsilon\ \tau) : \text{TYPE}}$$

The rule for REF type descriptions simply states that if ρ has kind REGION and τ has kind TYPE, then (REF ρ τ) has kind TYPE.

$$\frac{B \vdash \rho : \text{REGION} \quad B \vdash \tau : \text{TYPE}}{B \vdash (\text{REF } \rho\ \tau) : \text{TYPE}}$$

Definition. A description δ is *well-formed* with respect to a kind assignment B iff it has a kind under that kind assignment, *i.e.* iff $B \vdash \delta : \kappa$ for some κ.

The kind inference rules have been constructed so that any well-formed description has a unique kind: if $B \vdash \delta : \kappa_1$ and $B \vdash \delta : \kappa_2$ then $\kappa_1 = \kappa_2$.

3.5.2 Type and Effect Inference

In this section we present the axioms and rules for determining the type and effect of an expression. The axioms and rules for determining the *type* of an expression are intended to be as routine as possible; they closely resemble the corresponding axioms and rules for the higher-order lambda-calculus.

The axioms and rules for determining the *effect* of an expression are new, and merit some advance explanation. The effect axioms for constants, LAMBDA expressions and PLAMBDA expressions (in other words, the axioms for *values*) are straightforward: since values do not exhibit any computational behavior (*i.e.* they cannot be reduced), all values have effect PURE. Likewise, all ordinary variables have effect PURE.

The effects of the remaining expressions are given by effect inference rules. In general, the effect of an expression consists of two components, namely its *inherited* effect and its *intrinsic* effect:

- the *inherited* effect of an expression consists of the effects of those of its subexpressions that may be evaluated in order to evaluate the expression. Specifically,
  - an (ordinary or polymorphic) abstraction inherits no effects, and has an inherited effect of PURE;
  - any other expression has an inherited effect equal to the least upper bound of the effects of its immediate subexpressions.
- the *intrinsic* effect of the expression is the effect that is introduced by the expression itself rather than by one of its subexpressions. Specifically,
  - an (ordinary or polymorphic) application has an intrinsic effect equal to the latent effect of the subroutine;
  - a NEW, GET or SET expression has an intrinsic effect consisting of a ALLOC, READ or WRITE effect, respectively, on the region to which the location in question belongs;

- any other expression has an intrinsic effect of PURE.

The cumulative effect of an expression is the least upper bound of its inherited and intrinsic effects, either or both of which may be PURE.

Because an expression may have free ordinary and description variables, its type and effect are defined in the context of both a kind assignment and a *type assignment.* A type assignment is a partial function $A : Var \rightarrow Type$ that maps ordinary variables to their types.

Definition. An expression e *has type* τ with respect to the type assignment A and the kind assignment B iff the formula

$$A, B \vdash e : \tau$$

can be derived using the axioms and inference rules given below. Similarly, the expression *has effect* ϵ iff the formula

$$A, B \vdash e \,!\, \epsilon$$

can be derived.

We write $e : \tau$ when $A, B \vdash e : \tau$ for all A and B, and $e \,!\, \epsilon$ when $A, B \vdash e \,!\, \epsilon$ for all A and B.

The type axioms, below, give the types of the ordinary constants and variables.

$$\text{NIL} : \text{UNIT}$$
$$b : \text{BOOL}$$
$$A, B \vdash x : A(x)$$

The effect axioms, below, give the effect of values and ordinary variables, which is always PURE.

$$v \,!\, \text{PURE}$$
$$x \,!\, \text{PURE}$$

The type inference rule for ordinary abstraction (shown below) is a generalization of the corresponding rule in the typed lambda-calculus. The main difference is that the *effect* of the body of the LAMBDA expression is incorporated into the *type* of the expression itself. This reflects the fact that the body is not evaluated when the LAMBDA expression is evaluated, but when the subroutine is applied. We write $A[x \leftarrow \tau]$ to denote the type assignment obtained by patching A at x so that $A[x \leftarrow \tau](x) = \tau$. Note that the *effect* of a LAMBDA expression is always PURE, since a LAMBDA expression is a value.

$$\frac{\begin{array}{c} B \vdash \tau : \text{TYPE} \\ A[x \leftarrow \tau], B \vdash e : \tau' \\ A[x \leftarrow \tau], B \vdash e \,!\, \epsilon \end{array}}{A, B \vdash (\text{LAMBDA } (x{:}\tau)\ e) : (\text{SUBR } (\tau)\ \epsilon\ \tau')}$$

The type and effect inference rule for ordinary application is a generalization of the corresponding rule in the typed lambda-calculus. The main difference is that the latent effect incorporated in the *type* of the subroutine becomes part of the *effect* of the application as a whole. This reflects the fact that the body of the subroutine is evaluated when the subroutine is applied. Note that the type of the actual parameter need not match that of the formal parameter exactly, but must be included in it.

$$\frac{\begin{array}{c} A,B \vdash e_1 : (\text{SUBR } (\tau_1)\ \epsilon\ \tau_2) \\ A,B \vdash e_2 : \tau \wedge \tau \sqsubseteq \tau_1 \\ A,B \vdash e_1 \,!\, \epsilon_1 \\ A,B \vdash e_2 \,!\, \epsilon_2 \end{array}}{\begin{array}{lll} A,B \vdash (e_1\ e_2) & : & \tau_2 \\ A,B \vdash (e_1\ e_2) & ! & (\text{MAXEFF } \epsilon_1\ \epsilon_2\ \epsilon) \end{array}}$$

The rule for polymorphic abstraction is a generalization of the corresponding rule in the higher-order typed lambda-calculus. Again, the main difference is that the *effect* of the body of the PLAMBDA expression is incorporated into the *type* of the expression itself, reflecting the fact that the body is not evaluated when the PLAMBDA expression is evaluated, but when the subroutine is applied. Note that the *effect* of a PLAMBDA expression is always PURE, since a PLAMBDA expression is a value.

$$\frac{\begin{array}{c} A,B[d \leftarrow \kappa] \vdash e : \tau \\ A,B[d \leftarrow \kappa] \vdash e \,!\, \epsilon \\ \forall x \in FV(e)\ .\ d \notin FDV_{desc}(A(x)) \end{array}}{A,B \vdash (\text{PLAMBDA } (d{:}\kappa)\ e) : (\text{POLY } (d{:}\kappa)\ \epsilon\ \tau)}$$

This rule ensures that the free description variables of the types of the free variables of the body are not captured by the bound description variable.

The rule for polymorphic application is a generalization of the corresponding rule in the higher-order typed lambda-calculus. Again, the latent effect incorporated in the *type* of the subroutine becomes part of the *effect* of the application as a whole. Note that $\delta'[\delta/d]$ denotes the result of substituting δ for all free instances of d in δ', renaming the bound variables in δ' as needed to avoid capture.

$$\frac{\begin{array}{c} A,B \vdash e : (\text{POLY } (d{:}\kappa)\ \epsilon'\ \tau') \\ A,B \vdash e \,!\, \epsilon \\ B \vdash \delta : \kappa \end{array}}{\begin{array}{lll} A,B \vdash (\text{PROJ } e\ \delta) & : & \tau'[\delta/d] \\ A,B \vdash (\text{PROJ } e\ \delta) & ! & (\text{MAXEFF } \epsilon\ \epsilon'[\delta/d]) \end{array}}$$

The rule for conditional expressions ensures that the first subexpression has type BOOL, and that the remaining two subexpressions have types whose

least upper bound exists. When this is the case, the type of the IF expression is this least upper bound, and its effect is the least upper bound of the effects of its subexpressions.

$$\frac{\begin{array}{cc} A,B \vdash e_1 : \textsc{bool} & A,B \vdash e_1 \,!\, \epsilon_1 \\ A,B \vdash e_2 : \tau_2 & A,B \vdash e_2 \,!\, \epsilon_2 \\ A,B \vdash e_3 : \tau_3 & A,B \vdash e_3 \,!\, \epsilon_3 \\ \tau_2 \sqcup \tau_3 = \tau & \end{array}}{\begin{array}{lcl} A,B \vdash (\textsc{if}\ e_1\ e_2\ e_3) & : & \tau \\ A,B \vdash (\textsc{if}\ e_1\ e_2\ e_3) & ! & (\textsc{maxeff}\ \epsilon_1\ \epsilon_2\ \epsilon_3) \end{array}}$$

This inference rule is rather conservative: it is written as if both branches of a conditional are always evaluated. It might be possible to express the fact that the effect of the expression depends on the value of e_1, for example by using dependent effects of the form (IF $e\ \epsilon_1\ \epsilon_2$), which correspond closely to dependent conditionals [Car86, p. 13]. However, we are not currently exploring this line of research.

The rule for sequencing expressions is straightforward: provided that each subexpression is well-formed, the type of a BEGIN expression is the same as that of the last subexpression, and the effect is the least upper bound of the effects of the subexpressions.

$$\frac{\begin{array}{c} \forall i, 1 \le i \le n \,.\, A,B \vdash e_i : \tau_i \\ \forall i, 1 \le i \le n \,.\, A,B \vdash e_i \,!\, \epsilon_i \end{array}}{\begin{array}{lcl} A,B \vdash (\textsc{begin}\ e_1 \ldots e_n) & : & \tau_n \\ A,B \vdash (\textsc{begin}\ e_1 \ldots e_n) & ! & (\textsc{maxeff}\ \epsilon_1 \ldots \epsilon_n) \end{array}}$$

The remaining three rules deal with the operators for allocating, reading, and writing locations in the store. The rule for the NEW expression can be read as follows: provided that ρ has kind REGION, τ has kind TYPE, and e is well-formed, and provided that the type of e is a subtype of τ, the type of the expression as a whole is (REF $\rho\ \tau$), and the effect of the expression is the least upper bound of the effect of e and the effect (ALLOC ρ).

$$\frac{\begin{array}{c} B \vdash \rho : \textsc{region} \\ B \vdash \tau : \textsc{type} \\ A,B \vdash e : \tau' \quad \wedge \quad \tau' \sqsubseteq \tau \\ A,B \vdash e \,!\, \epsilon \end{array}}{\begin{array}{lcl} A,B \vdash (\textsc{new}\ \rho\ \tau\ e) & : & (\textsc{ref}\ \rho\ \tau) \\ A,B \vdash (\textsc{new}\ \rho\ \tau\ e) & ! & (\textsc{maxeff}\ \epsilon\ (\textsc{alloc}\ \rho)) \end{array}}$$

The rule for the GET expression can be read as follows: if the type of e is (REF $\rho\ \tau$), then the type of the expression as a whole is τ, and its effect is the least upper bound of the effect of e and the effect (READ ρ).

$$\frac{\begin{array}{c} A,B \vdash e : (\textsc{ref}\ \rho\ \tau) \\ A,B \vdash e \,!\, \epsilon \end{array}}{\begin{array}{lcl} A,B \vdash (\textsc{get}\ e) & : & \tau \\ A,B \vdash (\textsc{get}\ e) & ! & (\textsc{maxeff}\ \epsilon\ (\textsc{read}\ \rho)) \end{array}}$$

The rule for the SET expression can be read as follows: provided that the type of e_1 is (REF ρ τ) and the type of e_2 is a subtype of τ, the type of the expression as a whole is UNIT, and its effect is the least upper bound of the effects of e_1 and e_2 and the effect (WRITE ρ).

$$\frac{\begin{array}{c} A,B \vdash e_1 : (\text{REF } \rho\ \tau) \\ A,B \vdash e_2 : \tau' \quad \wedge \quad \tau' \sqsubseteq \tau \\ A,B \vdash e_1 \,!\, \epsilon_1 \\ A,B \vdash e_2 \,!\, \epsilon_2 \end{array}}{\begin{array}{ll} A,B \vdash (\text{SET } e_1\ e_2) & : \quad \text{UNIT} \\ A,B \vdash (\text{SET } e_1\ e_2) & ! \quad (\text{MAXEFF } \epsilon_1\ \epsilon_2\ (\text{WRITE } \rho)) \end{array}}$$

3.5.3 Properties of the Static Semantics

Although the type and effect inference rules appear to be interleaved, the rules are structured so that any expression that has a type also has an effect. We call such an expression *well-formed.*

Definition. An expression e is *well-formed* with respect to a type assignment A and a kind assignment B iff it has a type under A and B, *i.e.* iff $A,B \vdash e : \tau$ for some τ. If e is well-formed with respect to the empty type and kind assignment we write $\mathcal{WF}_{exp}(e)$.

The type and effect inference rules have been constructed so that the type and effect of a well-formed expression are themselves well-formed and of kind TYPE and EFFECT respectively: if $A,B \vdash e : \tau$ and $A,B \vdash e \,!\, \epsilon$ and $B \vdash A(x) :$ TYPE for each $x \in FV(e)$, then $B \vdash \tau :$ TYPE and $B \vdash \epsilon :$ EFFECT.

The type and effect inference rules have been constructed so that any well-formed expression has a unique type and effect: if $A,B \vdash e : \tau_1$ and $A,B \vdash e : \tau_2$ then $\tau_1 = \tau_2$, and likewise for effects.

Finally, two properties of type and effect inference are of particular significance: a well-formed ordinary application yields a well-formed result with the expected type and effect, and a well-formed polymorphic application yields a well-formed result with the expected type and effect in which the actual parameter has been substituted for the bound description variable of the operator.

- If an expression e has type τ and effect ϵ under a type assignment A and a kind assignment B such that $A(x) = \tau'$, and e' is a closed expression whose type is a subtype of τ', then $e[e'/x]$ is well-formed under A and B, and has a type that is a subtype of τ and an effect that is a subeffect of ϵ.
- If an expression e has type τ and effect ϵ under a type assignment A and a kind assignment B such that $B(d) = \kappa$, and δ is a closed description of kind κ, then $e[\delta/d]$ has type $\tau[\delta/d]$ and effect $\epsilon[\delta/d]$ under A and B.

These two properties are used in the next chapter to prove the all-important type and effect preservation proposition.

3.6 Aliasing

The notion of *aliasing* usually refers to a situation in which two distinct identifiers (*i.e.* constants and/or variables) refer to overlapping sets of locations. Since we take the view that effect analysis should be performed strictly on the basis of effect specifications, we are interested only in aliasing between *description* identifiers.

Most programming languages offer two ways to bind ordinary variables: as formal parameters or as local variables. The actual value of the variable is unknown in the former case, but known in the latter. Similarly, a description variable can be either an *abstract* type, effect or region, or a *synonym* for a specific type, effect or region description. The actual description corresponding to the variable is unknown in the former case, but known in the latter. We regard description synonyms as mere syntactic sugar, and we have not included them in *MFX*; in Chapter 7 we show how they can be added. In this section we discuss the issue of aliasing as it relates to description variables that represent abstract types, effects, and regions respectively.

Since we take the view that effect analysis should be performed strictly on the basis of effect specifications, the *MFX* type and effect inference rules use only the description information that has been supplied in the form of declarations, even when the corresponding actual parameter is known.

3.6.1 Abstract Types

For type variables, the only property of interest is *convertibility*. Since distinct type variables are treated as if they were unrelated type constants, a type variable is not convertible with any other type description. This guarantees representation independence.

3.6.2 Abstract Effects

For effect variables, there are two properties of interest: convertibility and *interference*. As for convertibility, distinct effect variables are treated as if they were unrelated effect constants, and an effect variable is not convertible with any other effect description. As for interference, no additional assumptions are made: thus, any conservative syntactic effect analysis algorithm has to assume that an effect variable may interfere with any effect, including itself. For a detailed discussion of interference, see Chapter 8.

We have considered augmenting *MFX* with some form of *bounded quantification* over effects; this would permit the programmer to specify, as part of the declaration of a formal effect parameter, an upper bound (such as a read-only effect description) that the corresponding actual parameter must satisfy. However, a detailed investigation of bounded quantification is beyond the scope of this thesis.

3.6.3 Abstract Regions

For region variables, there are also two properties of interest: convertibility and *overlap*. As for convertibility, distinct region variables are treated as if they were unrelated region constants, and a region variable is not convertible with any other region description. As for overlap, the treatment of regions differs somewhat from the treatment of types and effects. If no additional assumptions were made about region parameters, any conservative syntactic effect analysis algorithm would have to assume that any two region variables may correspond to overlapping regions. This would discourage the use of procedural abstraction, which is contrary to our philosophy.

We have considered augmenting *MFX* with *constrained quantification* over regions; this would permit the programmer to specify, as part of the declaration of a formal region parameter, a set of constraints (such as disjointness from certain other region descriptions) that the corresponding actual parameter must satisfy. However, a detailed investigation of constrained quantification is beyond the scope of this thesis.

As a compromise, we have adopted a simple language restriction that ensures that there is no aliasing between region identifiers. The main advantage of this restriction is that it permits a conservative effect analysis algorithm to assume that distinct region identifiers correspond to disjoint regions. Of course, the main drawback of the restriction is that, due to its conservative nature, it rules out many programs that do not actually introduce aliasing at all.

The *MFX* anti-aliasing rule is modeled after the rule employed by Euclid [Lam77] [Pop77]. However, since the rule concerns aliasing between region descriptions, it does not affect ordinary applications. Only polymorphic applications are affected, and then only when the operand is a region description.

The objective of the anti-aliasing rule is to ensure that for any polymorphic application (PROJ e ρ), the region description ρ be disjoint from all region constants and variables that are used by the value of e, which must be a polymorphic subroutine. Fortunately, the region constants and variables that are used by a value all appear free in its type. This leads us to the following restriction:

Restriction. In a polymorphic application (PROJ e ρ), where e has type τ, no region constant or variable may appear free in both τ and ρ; in other words, $FRC_{desc}(\tau) \cap FRC_{desc}(\rho)$ and $FDV_{desc}(\tau) \cap FDV_{desc}(\rho)$ must be empty.

Since the region constants and variables that are used by a value all appear free in its type, the above restriction is *sufficient* to maintain the invariant that effects on distinct region constants and/or variables in any scope do not interfere. The restriction is rather conservative: it is possible that more precise anti-aliasing rules can be devised. But, to paraphrase Reynolds,

where to stop is ultimately a question of taste: the anti-aliasing rule should permit expressions that *obviously* do not interfere [Rey78, p. 41, emphasis in original]. We regard the current anti-aliasing rule as no more than a reasonable compromise.

Chapter 4. Dynamic Semantics

4.1 Introduction

In this chapter we continue our presentation of *MFX-1* by defining its semantics and presenting our propositions regarding the soundness of the type and effect system. In particular, we present our claims regarding type and effect preservation, type soundness, static typing, effect soundness, location invariance, and typeless semantics. We give proof sketches for all these propositions.

The purpose of this chapter is to define the semantics of the language formally, and to demonstrate the soundness of the type and effect system.

The rest of this chapter is organized as follows. We begin by defining the standard semantics of the language, which is expressed in terms of a set of rewriting axioms and rewriting inference rules. Next, we present some properties of these rewriting rules that illustrate the structure of the standard semantics. We then present the type soundness and static typing propositions, followed by the effect soundness proposition. We conclude by discussing some additional important properties of the semantics, namely location invariance and typeless semantics.

4.2 Overview

The standard semantics of the language is based on the standard rewriting rules for the second-order typed lambda-calculus [Bar84]. In particular, we have expressed the semantics of application in terms of beta-substitution. Consequently, there is no environment that maps free variables to their values, and expressions ought not to have free variables.

This way of treating free variables appears to be rather unusual: the most popular ways of specifying the semantics of an imperative language, such as denotational semantics [Gor79b] [Sco82] [Sto77], so-called structured operational semantics [Plo81], and the meta-circular evaluator approach [Abe85] all use environments to represent the relation between variables and their values. We have adopted the current approach primarily in order to simplify our proofs.

In our standard semantics, effects are modeled using a *store* that maps locations to values. To avoid the complications that arise when a computation runs out of unused locations, we define a store to be a *finite* function from locations to values. Since the number of locations is infinite and every finite computation allocates only a finite number of locations, this definition ensures that a computation never runs out of unused locations.

The *state* of a computation consists of an expression and a store. Computation proceeds by repeatedly rewriting the state until a terminal state is reached. The rewriting inference rules ensure left-to-right applicative order evaluation by designating, for each state, the unique innermost expression (if any) that can be rewritten next. Since the rewriting rules are uni-directional, we use the term *reduction* rather than *rewriting*, and we refer to the rules as *reduction rules.*

To avoid over-specification, we have defined the standard semantics so that new locations, when needed, are chosen nondeterministically. We show that the course of computation is not affected by the choice of locations, and that the result of a computation is unique modulo the choice of locations. This gives the language implementation a great deal of flexibility, which is essential for optimization.

Although there is type and effect information available during the reduction process, we show that the standard semantics does not make use of this information. The compiler described in Chapter 8 takes advantage of this fact.

The main propositions presented in this chapter are the type and effect soundness propositions.

- The type soundness proposition states that if a well-formed expression has a certain type description, then the type of the value returned by the expression is a subtype of that type description.
- The effect soundness proposition states that if a well-formed expression has a certain effect description, then the actual side-effects of the expression are a subeffect of that effect description.

The effect soundness property forms the basis for the use of syntactic effect specifications to identify optimization opportunities.

4.3 Standard Semantics

4.3.1 Locations

Before we can describe the semantics formally, we must define what we mean by locations. Formally, locations are a countably infinite set of constants:

$Loc =$	$\{l_1, l_2, \dots\}$	– locations (l)
$Const =$	$\dots$	– ordinary constants
	Loc	– locations

A location can be *tagged* with a region description and a type description. The region tag of a location indicates to what region the location belongs, and the type tag of a location indicates what types of values the location may contain. Specifically, a location tagged with a region description ρ

belongs to the region ρ, and a location tagged with a type description τ may contain values whose type is a subtype of τ. The tags of a location ought to be *closed*; tags that contain free description variables are meaningless. We write $R(l)$ for the region tag of the location l and $T(l)$ for its type tag. Moreover, we write $l_{\rho,\tau}$ to indicate that $R(l_{\rho,\tau}) = \rho$ and $T(l_{\rho,\tau}) = \tau$.

Every closed region description ρ corresponds to a nonempty set of region constants, namely $FRC_{desc}(\rho)$. If ρ is a region constant, then the location $l_{\rho,\tau}$ belongs to the region corresponding to that region constant. If ρ is a UNION of several region constants, then the location $l_{\rho,\tau}$ belongs to the union of the corresponding regions. This situation reflects either *uncertainty* or *indifference* about the region constant to which the location actually belongs.

Definition. A location can be *reached* through a region ρ, $l \in Reach(\rho)$, iff the region tag of l overlaps with ρ, *i.e.* iff $FRC_{desc}(R(l)) \cap FRC_{desc}(\rho) \neq \emptyset$.

It is convenient to define the free locations of an expression, $FL_{exp}(e)$. Since locations are constants, the definition of FL_{exp} is trivial: all the locations that occur in an expression are free. We give a more precise syntactic definition anyway, in order to highlight the differences between $FL_{exp}(e)$ and $FV(e)$.

Definition. The free locations of an expression are given by the function $FL_{exp} : Exp \rightarrow \mathcal{P}ow(Loc)$ defined below.

$$\begin{aligned} FL_{exp}(\text{NIL}) &= \emptyset \\ FL_{exp}(b) &= \emptyset \\ FL_{exp}(l) &= \{l\} \\ FL_{exp}(x) &= \emptyset \\ FL_{exp}(\text{LAMBDA } (x{:}\tau)\ e) &= FL_{exp}(e) \end{aligned}$$

The remaining clauses of the definition of $FL_{exp}(e)$ are identical to the corresponding clauses of the definition of $FV(e)$.

Since locations are constants and therefore expressions, we must define their free ordinary and description variables, their free region constants, their types, and their effects. The first few are easy: since locations are constants, they have neither free ordinary variables nor free description variables. However, because of its region and type tag, a location may have free region constants:

$$FRC_{exp}(l_{\rho,\tau}) = FRC_{desc}(\rho) \cup FRC_{desc}(\tau)$$

Because locations are constants, their effect is PURE. Finally, the type of a location is a REF type whose region and type parameters are equal to the region and type tags of the location:

$$l_{\rho,\tau} : (\text{REF } \rho\ \tau)$$

4.3.2 Stores and States

The state of a computation consists of two components: an expression, which represents the computation that remains to be performed, and a *store*, which maps locations to values. Stores and states are formally defined below.

Definition. A *store* is a finite function $\sigma : Loc \rightarrow Val$ that maps locations to values. We use *Store* to denote the set of stores and σ to denote individual stores.

Definition. A *state* is a tuple $\langle e, \sigma \rangle$ of the form $(Exp \times Store)$. We use *State* to denote the set of states and θ to denote individual states.

Definition. A state θ is called a *terminal* state iff its expression component is a *value*, *i.e.* iff $\theta = \langle v, \sigma \rangle$ for some v and σ.

4.3.3 Reduction

Computation proceeds by repeatedly reducing the current state until a terminal state is reached. The reduction relation $\stackrel{\text{red}}{\Longrightarrow}$ on $State \times State$ is defined by a set of reduction axioms and a set of reduction inference rules. The reduction axioms show how to reduce an expression when certain of its (immediate) subexpressions have already been reduced to values; the reduction inference rules show how to reduce an expression to which none of the reduction axioms applies by reducing one of its (immediate) subexpressions.

A terminal state cannot be reduced; in other words, there are no v, σ and θ such that $\langle v, \sigma \rangle \stackrel{\text{red}}{\Longrightarrow} \theta$. The reduction axioms and inference rules make extensive use of this fact in order to ensure the correct evaluation order. For example, the reduction axiom for ordinary application (shown below) is applicable only when the operator is a LAMBDA expression, which is a value, and the operand is a value as well. This technique is used throughout to keep the reduction axioms and inference rules from being invoked prematurely.

The relation 'reduces to' or '$\stackrel{\text{red}}{\Longrightarrow}$' on $(State \times State)$ gives the states, if any, to which a given state can be reduced.

Definition. The relation 'reduces to' is the relation generated by the axioms and inference rules given below.

The first two axioms, which deal with ordinary and polymorphic application, are adapted directly from the second-order typed lambda calculus. As before, the notation $e[v/x]$ and $e[\delta/d]$ indicates beta-substitution, where bound variables are renamed as needed to avoid capture. Note that the store is not involved in these reductions.

$$\langle((\text{LAMBDA } (x{:}\tau)\ e)\ v), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e[v/x], \sigma\rangle$$

$$\langle((\text{PLAMBDA } (d{:}\kappa)\ e)\ \delta), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e[\delta/d], \sigma\rangle$$

Note that the first of these axioms may duplicate the actual parameter v. This does not cause any problems, despite the possibility of side-effects,

because the actual parameter is a value, which cannot be reduced and may therefore be duplicated freely.

The next set of axioms, which deal with conditional and sequential evaluation, should be more or less self-explanatory. Note, once again, that the store is not involved in these reductions.

$$\langle(\text{IF TRUE } e_2\ e_3), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e_2, \sigma\rangle$$
$$\langle(\text{IF FALSE } e_2\ e_3), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e_3, \sigma\rangle$$
$$\langle(\text{BEGIN } v), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle v, \sigma\rangle$$
$$\langle(\text{BEGIN } v\ e_1 \ldots e_n), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle(\text{BEGIN } e_1 \ldots e_n), \sigma\rangle \quad (n > 0)$$

The remaining axioms deal with the allocating, reading, and writing of locations.

The axiom for the NEW expression can be read as follows. To reduce the expression (NEW $\rho\ \tau\ v$), choose any location l that is not bound in the store, and tag it with the descriptions ρ and τ. Then bind l to v in the store, and replace the expression by the value l.

$$\langle(\text{NEW } \rho\ \tau\ v), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle l_{\rho,\tau}, \sigma[l_{\rho,\tau} \leftarrow v]\rangle \quad (l \textit{ not bound in } \sigma)$$

This axiom represents a non-deterministic reduction: unlike the other axioms, this axiom permits a state to be reduced in one step to any of a countably infinite number of states, differing only in their choice of the new location. We show that the choice of new locations does not affect the course of a computation.

To reduce the expression (GET l), where l is bound to v in the store, simply replace the expression by the value v. The tags of the location are immaterial.

$$\langle(\text{GET } l), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle\sigma(l), \sigma\rangle$$

Note that this reduction duplicates the value v. This does not cause any problems, despite the possibility of side-effects, because a value cannot be reduced and may therefore be duplicated freely.

To reduce the expression (SET $l\ v$), where l is bound in the store, simply bind l to v in the store and replace the expression by the value NIL. In this case, too, the tags of the location are immaterial.

$$\langle(\text{SET } l\ v), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle\text{NIL}, \sigma[l \leftarrow v]\rangle$$

This concludes the set of reduction axioms. Note that these axioms can be invoked only when the outermost expression of the state matches the pattern of an axiom.

The reduction inference rules show how to reduce a state θ that does not match any reduction axiom by reducing a designated subexpression of the expression component of θ. In order to describe the reduction inference rules we need to define the notion of a *context*.

Definition. A *context* C is an expression containing a single "hole" in which an expression can be placed, *i.e.* such that $C[e]$ is an expression for any expression e. For example, if C is (GET (e [ ])) then $C[e']$ is (GET (e e')).

The reduction inference rules are all of the following form:

$$\frac{\langle e, \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma'\rangle}{\langle C[e], \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle C[e'], \sigma'\rangle}$$

Each rule can be represented by an appropriate context C: for example, the reduction inference rules for application,

$$\frac{\langle e_1, \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e_1', \sigma'\rangle}{\langle (e_1\ e_2), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle (e_1'\ e_2), \sigma'\rangle}$$

and

$$\frac{\langle e_2, \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e_2', \sigma'\rangle}{\langle (v_1\ e_2), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle (v_1\ e_2'), \sigma'\rangle}$$

can be represented by the contexts ([ ] e_2) and (v_1 [ ]). The reduction inference rules of *MFX* as a whole are represented by the following contexts:

- ordinary constants and variables: none
- ordinary abstraction: none
- ordinary application: ([ ] e_2) and (v_1 [ ])
- polymorphic abstraction: none
- polymorphic application: (PROJ [ ] δ)
- conditional evaluation: (IF [ ] e_2 e_3)
- sequential evaluation: (BEGIN [ ] $e_2 \ldots e_n$)
- allocating a location: (NEW ρ τ [ ])
- reading a location: (GET [ ])
- writing a location: (SET [ ] e_2) and (SET v_1 [ ])

Together with the reduction axioms, these rules guarantee left-to-right, applicative order evaluation.

The *meaning* of an expression is a map from stores to terminal states (where $\stackrel{\text{red}}{\Longrightarrow}{}^*$ is the transitive closure of $\stackrel{\text{red}}{\Longrightarrow}$):

Definition. The *meaning* of an expression e is the map $M[\![\ e\ ]\!] : \mathit{Store} \to \mathit{State}$ that maps every store σ to the set of terminal states that can result from reducing the state $\langle e, \sigma\rangle$, *i.e.*

$$M[\![\ e\ ]\!]\sigma = \{\langle v, \sigma'\rangle \mid \langle e, \sigma\rangle \stackrel{\text{red}}{\Longrightarrow}{}^* \langle v, \sigma'\rangle\}$$

This map is many-to-many: for example, the expression (NEW r BOOL TRUE) maps the store σ to the set of states $\langle l, \sigma[l \leftarrow \text{TRUE}]\rangle$ for all $l \notin Dom(\sigma)$, and the expressions TRUE, (BEGIN TRUE), (BEGIN (BEGIN TRUE)), and so forth all map any store σ to the state $\langle \text{TRUE}, \sigma\rangle$.

In a subsequent section we show that for any expression e and store σ, the terminal states in the set $M[\![\ e\ ]\!]\sigma$ are all equal up to the choice of locations.

4.3.4 Properties of the Standard Semantics

The reduction axioms and inference rules have been designed so that every expression matches at most one reduction axiom or one reduction inference rule.

For example, consider the expression

```
((PROJ (PLAMBDA (t:TYPE)
         (LAMBDA (x:t)
           x))
       BOOL)
 (GET 1))
```

Although this expression is an application, it does not match the application reduction axiom, since neither the operator nor the operand are values. However, it does match the first of the two reduction inference rules for application, since the operator is a polymorphic application that can be reduced by the corresponding reduction axiom. This reduction changes the expression to

```
((LAMBDA (x:BOOL)
   x)
 (GET 1))
```

without accessing or changing the store.

The resulting expression still does not match the application reduction axiom, since the operand is not a value, as required by the axiom. It also does not match the first application reduction inference rule, since there is no way to reduce the operator. It does, however, match the second reduction inference rule, since the operator is a LAMBDA expression, which is a value. If the store maps 1 to TRUE, this reduction changes the expression to

```
((LAMBDA (x:BOOL)
   x)
 TRUE)
```

while reading the store to obtain the contents of 1.

Since the operator and the operand of this expression are both values, this expression can be reduced by the application reduction axiom. This reduction changes the expression to

```
TRUE
```

without accessing or changing the store. Note that since every expression matches at most one reduction axiom or one reduction inference rule, the order in which the reduction axioms and inference rules are listed is immaterial.

By induction on the number of inference rules employed in a reduction step, it can be shown that every state can be reduced in at most one way, except for the choice of free locations. The expressions that can participate in the next reduction step of a state are called the *active* expressions of that state.

Definition. The *active* expressions of an expression are given by the following inductive definition:

1. any expression is an active expression of itself, provided that it is not a value;
2. if $C[e]$ is an active expression of e_0, and there exists a reduction inference rule that states that $\langle C[e], \sigma \rangle$ can be reduced to $\langle C[e'], \sigma' \rangle$ if $\langle e, \sigma \rangle$ can be reduced to $\langle e', \sigma' \rangle$, then e is an active expression of e_0, provided that it is not a value.

The active expressions and contexts of a context are defined analogously.

Definition. The active expressions of a state are the active expressions of its expression component.

The active expressions of an expression form a *chain*: the chain begins at the expression itself, and the active (immediate) subexpression of each active expression forms the next link of the chain. The last expression on this chain is called the *active redex* of the expression. For example, the expression

```
((LAMBDA (x:BOOL) x)
 (BEGIN
   (SET 1 FALSE)
   (GET 1)))
```

has the following chain of active expressions:

```
((LAMBDA (x:BOOL) x)
 (BEGIN
   (SET 1 FALSE)
   (GET 1)))
    |
(BEGIN
  (SET 1 FALSE)
  (GET 1))
    |
(SET 1 FALSE)
```

The active redex of the expression is (SET 1 FALSE).

The expression (LAMBDA (x:BOOL) x) in the above expression is not active because it is already a value; the expression (GET 1) is not active because the reduction inference rule for BEGIN reduces only the first subexpression.

Since every state has at most one active redex, it follows that every state can be reduced in at most one way, namely by invoking the reduction axiom, if any, that matches the active redex. Since all the reduction axioms are deterministic except for the choice of free locations it follows that reduction in general is deterministic except for the choice of free locations.

4.3.5 Stuck States

Definition. A nonterminal state $\langle e, \sigma \rangle$ (where $e \notin Val$) is *stuck* iff it cannot be reduced, *i.e.* iff there is no θ such that $\langle e, \sigma \rangle \stackrel{\text{red}}{\Longrightarrow} \theta$.

In a stuck state, the active redex is one of the following:

1. a variable
2. $(v_1\ v_2)$ where v_1 is not an ordinary subroutine
3. (PROJ $v\ \delta$) where v is not a polymorphic subroutine
4. (IF $v\ e_2\ e_3$) where v is not a Boolean
5. (GET v) where v is not a location
6. (GET l) where l is not bound in the store
7. (SET $v_1\ v_2$) where v_1 is not a location

These expressions can be divided into various categories:

- case 1 represents an attempt to use an undeclared variable,
- case 6 represents an attempt to use an uninitialized location, and
- cases 2-5 and 7 represent type errors.

Proposition. If the state $\langle e, \sigma \rangle$ is stuck, then either e is ill-formed, or it contains some location whose contents is undefined.

Proof. A state $\langle e, \sigma \rangle$ is stuck iff its active redex matches one of the cases 1-7 listed above. Let e' be this active redex. If e' contains some location whose contents is undefined, then so does e, and if e' is ill-formed, then so is e (by the monotonicity of the type inference rules). Therefore, it remains to verify that if e' matches one of the cases 1-7 listed above, then it is either ill-formed or contains some location whose contents is undefined. We discuss the first few cases.

1. If e' is a variable, then e' is not closed, and therefore ill-formed.
2. If e' is of the form $(v_1\ v_2)$ where v_1 is not an ordinary subroutine, then e' is not well-formed because no value other than an ordinary subroutine can have a type of the form (SUBR $(\tau_1)\ \epsilon\ \tau_2$).

The remaining cases are similar. □

In the next section we show that reduction of a well-formed state never gets stuck. In practical terms, this means that static type checking prevents all run-time type errors and all attempts to use undeclared variables or uninitialized locations.

4.4 Type and Effect Preservation

In this section we show that reduction of a well-formed state yields another well-formed state whose type and effect are at most those of the original state. Since a well-formed state is not stuck, it follows as a corollary that reduction of a well-formed state never gets stuck.

A state is *consistent* iff all the locations that occur in the state are bound in its store component, and have the same tags everywhere. Before we can formalize this definition, we must define what it means for a location to *occur* in a store or state.

Definition. A location l *occurs* in a state $\langle e, \sigma \rangle$ iff it occurs in the expression component e or in the store component σ. The locations that occur in a state are given by the function FL_{sta}, which is formally defined below.

$$FL_{sto}(\sigma) = Dom(\sigma) \cup \bigcup_{l \in Dom(\sigma)} FL_{exp}(\sigma(l))$$

$$FL_{sta}(\langle e, \sigma, \rangle) = FL_{exp}(e) \cup FL_{sto}(\sigma)$$

Definition. A state is *consistent*, $\mathcal{C}(\langle e, \sigma \rangle)$, iff every location that occurs in the state is bound in the store and has the same tags everywhere, *i.e.* iff $FL_{sta}(\langle e, \sigma \rangle) \subseteq Dom(\sigma)$, and $l_{\rho,\tau} \in FL_{sta}(\langle e, \sigma \rangle)$ and $l_{\rho',\tau'} \in FL_{sta}(\langle e, \sigma \rangle)$ implies that $\rho = \rho'$ and $\tau = \tau'$.

Definition. A store is *well-formed*, $\mathcal{WF}_{store}(\sigma)$, iff every value in the store is well-formed and has a type description that is a subtype of the type tag of its location. In other words,

$$\mathcal{WF}_{store}(\sigma) \quad \Leftrightarrow \quad l \in Dom(\sigma) \Rightarrow (\sigma(l) : \tau \wedge \tau \sqsubseteq T(l))$$

We can now define what constitutes a well-formed *state*.

Definition. A state $\langle e, \sigma \rangle$ is *well-formed*, $\mathcal{WF}_{state}(\langle e, \sigma \rangle)$, iff it is consistent and e and σ are both well-formed. In other words,

$$\mathcal{WF}_{state}(\langle e, \sigma \rangle) \quad \Leftrightarrow \quad \mathcal{C}(\langle e, \sigma \rangle) \wedge \mathcal{WF}_{exp}(e) \wedge \mathcal{WF}_{store}(\sigma)$$

If $\langle e, \sigma \rangle$ is well-formed and $e : \tau$, we write $\langle e, \sigma \rangle : \tau$ and say that $\langle e, \sigma \rangle$ has type τ; similarly, if $\langle e, \sigma \rangle$ is well-formed and $e \,!\, \epsilon$, we write $\langle e, \sigma \rangle \,!\, \epsilon$ and say that $\langle e, \sigma \rangle$ has effect ϵ. Note that every terminal state has effect PURE.

We can now express the type and effect preservation proposition. This proposition is a generalization of the type preservation theorem of the second-order typed lambda-calculus, which states that reduction of a well-typed expression yields another well-typed expression of the same type.

The proposition presented here is more general than the type preservation theorem of the lambda-calculus in three respects: side-effects, type inclusion, and effect descriptions.

- In order to deal with side-effects, we have generalized our proposition from expressions to states.
- In order to deal with type inclusion, we have relaxed our proposition so that the reduction of a state of type τ may yield a state of any type $\tau' \sqsubseteq \tau$.
- Finally, in order to deal with effect descriptions, we have added the proposition that the reduction of a state with effect ϵ yields a state with some effect $\epsilon' \sqsubseteq \epsilon$.

Proposition. (Type and Effect Preservation) Reduction of a well-formed state preserves or decreases the type and effect descriptions of the state.

$$\begin{array}{c} \mathcal{WF}_{state}(\langle e, \sigma \rangle) \\ e : \tau \\ e \,!\, \epsilon \\ \langle e, \sigma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma' \rangle \end{array} \quad \Rightarrow \quad \begin{array}{c} \mathcal{WF}_{state}(\langle e', \sigma' \rangle) \\ e' : \tau' \text{ where } \tau' \sqsubseteq \tau \\ e' \,!\, \epsilon' \text{ where } \epsilon' \sqsubseteq \epsilon \end{array}$$

Proof. By induction on the number of inference rules employed in a reduction step, with a case analysis for the reduction axiom employed. □

During the course of a computation, both the type and the effect of the state may decrease. As is customary, the decrease in the type reflects the decreasing uncertainty regarding the type of the eventual result of the computation, and the same is true of the decrease in the effect. However, the decrease in the effect also reflects the fact that during the course of reduction, certain effects actually take place, during which process the expressions that actually cause the effects are replaced by values, which cannot cause any further effects.

This difference is most apparent when the computation reaches a terminal state, by which time the effect of the state must have decreased to PURE. The type of the state, on the other hand, is not constrained *a priori* to converge to a certain type.

Corollary 1. (Type Soundness) If reduction of a well-formed state $\langle e, \sigma \rangle$ terminates in a state $\langle v, \sigma' \rangle$, then the type of v is a subtype of the type of e.

Corollary 2. (Static Typing) Reduction of a well-formed state never gets stuck; in particular, reduction of a well-formed state never encounters an undefined variable, an uninitialized location, or a type error.

In practical terms, this means that an implementation of the language does not need to check for these conditions at run-time.

4.5 Effect Soundness

In this section we show that the effects of reduction of any well-formed state are a subeffect of the effect description of that state. This property forms the basis for syntactic effect analysis.

Lemma. In a well-formed state, every active expression is closed.

Proof. Since every well-formed state is closed, all expressions containing free variables must appear inside of LAMBDA and PLAMBDA expressions. By definition, these expressions are values, which have no active subexpressions. □

Since every active expression in a well-formed state is closed, we can refer to the effect of an active expression without specifying a type or kind assignment.

Lemma. (Effect Propagation) In a well-formed state, the effect of each active expression is a subeffect of the effect of its parent expression. In particular, the effect of the active redex is a subeffect of the effect of the outermost expression.

Proof. There is a direct correspondence between the reduction inference rules, which determine which of the subexpressions of an expression can be active, and the effect inference rules: the effect inference rules are deliberately structured so that the effect of every expression is at least the least upper bound of the effects of its (immediate) subexpressions that can be active. It follows that the effect of each active expression is a subeffect of the effect of its parent expression. □

In order to state our effect soundness proposition, we need to be able to refer to the locations that are involved in effects in a reduction step.

Definition. For all θ and θ' such that $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$, let

- $\mathcal{A}(\theta, \theta')$ denote the set of locations allocated in the reduction step $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$
- $\mathcal{R}(\theta, \theta')$ denote the set of locations read in the reduction step $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$
- $\mathcal{W}(\theta, \theta')$ denote the set of locations written in the reduction step $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$

Proposition. (Effect Soundness) Reduction of a well-formed state allocates, reads, and writes only locations that can be reached through the regions specified by its effect. In other words, if $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$ and $\theta \,!\, \epsilon$ where

$$\epsilon \simeq (\text{MAXEFF } (\text{ALLOC } \rho_A)\ (\text{READ } \rho_R)\ (\text{WRITE } \rho_W))$$

then

$$\begin{aligned} \mathcal{A}(\theta, \theta') &\subseteq \mathit{Reach}(\rho_A) \\ \mathcal{R}(\theta, \theta') &\subseteq \mathit{Reach}(\rho_R) \\ \mathcal{W}(\theta, \theta') &\subseteq \mathit{Reach}(\rho_W) \end{aligned}$$

Proof. By the effect propagation lemma and the fact that every NEW, GET or SET expression has a corresponding ALLOC, READ or WRITE effect. □

Since reduction preserves or reduces the effect of a state, this proposition generalizes immediately to $\theta \stackrel{red}{\Longrightarrow}{}^* \theta'$.

In practical terms, the effect soundness proposition implies that the effect description of an expression is a conservative approximation of the actual side-effects that the expression may have. This effect information, which can be inferred and verified statically, can be used to identify a variety of optimization opportunities, including concurrent evaluation and memoization. This means that it is possible to integrate functional program fragments into imperative programs while retaining the benefits of functional programming within those program fragments.

4.6 Location Invariance

In this section we show that the meaning of an expression is independent of the choice of locations that are allocated during the computation. In practical terms, this means that the semantics of the language are independent of the storage allocation policy employed by the implementation.

The proof of this property is fairly routine. We begin by defining an equivalence relation that relates stores that are equal up to the choice of locations, and likewise for states. We then show that if a state can be reduced in one step to multiple states, then these resulting states are all equivalent, and we show that subsequent reduction preserves this equivalence. We finally consider the equivalence classes of stores and states modulo the choice of locations, and we show that the meaning of an expression corresponds to a function from equivalence classes of stores to equivalence classes of terminal states.

Definition. A *location permutation* is a bijection $\mu : Loc \rightarrow Loc$. A permutation μ *fixes* a set of locations L iff $\mu(l) = l$ for all $l \in L$.

We extend permutations to stores and states as follows: $\mu(\sigma)$ and $\mu(\theta)$ denote the store and state, respectively, obtained by simultaneously replacing every location occurrence l in σ or θ (as defined by $FL_{sto}(\sigma)$ and $FL_{sta}(\theta)$) by $\mu(l)$.

When two states are equal up to a location permutation, we consider them to be *equivalent*. If this permutation fixes a set of locations L, the states are equivalent with respect to L.

Definition. Two states θ and θ' are *equivalent* with respect to a set of locations L, $\theta \doteq_L \theta'$, iff $\theta' = \mu(\theta)$ for some μ that fixes L.

This relation is an equivalence relation: it is reflexive (since the identity permutation fixes every set L), it is symmetric (since the inverse of a permutation that fixes L also fixes L) and it is transitive (since the composition of two permutations that fix L also fixes L).

When two states are equivalent with respect to $FL_{exp}(e)$ or $FL_{sta}(\theta)$, we say that they are equivalent with respect to e or θ, respectively.

Since a reducible state can be reduced in precisely one way except for the choice of locations that are allocated during the reduction, the states to which a given state can be reduced are all equivalent with respect to the locations that occur in the original state.

Proposition. If a state reduces to more than one state, then the resulting states are equivalent with respect to the locations that occur in the original state

$$\theta \stackrel{\text{red}}{\Longrightarrow} \theta_1' \ \wedge \ \theta \stackrel{\text{red}}{\Longrightarrow} \theta_2' \quad \text{implies} \quad \theta_1' \doteq_{FL_{sta}(\theta)} \theta_2'$$

Proof. If the reduction does not allocate a location, reduction is deterministic and $\theta_1' = \theta_2'$. Otherwise, let l_1 and l_2 be the new locations allocated in θ_1' and θ_2' respectively. Choose μ so that $\theta = \mu(\theta)$ and $l_2 = \mu(l_1)$. Such a μ is guaranteed to exist because neither l_1 nor l_2 is in $FL_{sta}(\theta)$. We then have $\theta_2' = \mu(\theta_1')$. □

In fact, a stronger result holds: any state that is equivalent to a reducible state with respect to some set of locations L is itself reducible, and the results of these two reductions are also equivalent with respect to L.

Proposition. (Location Invariance) Equivalent states reduce to equivalent results.

$$\theta_1 \doteq_L \theta_2 \ \wedge \ \theta_1 \stackrel{\text{red}}{\Longrightarrow} \theta_1' \quad \Rightarrow \quad \begin{cases} \exists \theta_2' \ . \ \theta_2 \stackrel{\text{red}}{\Longrightarrow} \theta_2' \\ \forall \theta_2' \ . \ \theta_2 \stackrel{\text{red}}{\Longrightarrow} \theta_2' \ \Rightarrow \ \theta_1' \doteq_L \theta_2' \end{cases}$$

Proof. Omitted. □

This proposition subsumes the previous proposition, as can be shown by taking $L = FL_{sta}(\theta_1)$.

The location invariance proposition generalizes immediately to reduction sequences of any length. Moreover, using the fact that any state that is equivalent to some terminal state is itself a terminal state, it follows that either all reduction sequences of a particular state diverge, or all reach a terminal state after the same number of steps, in which case these terminal states are all equivalent with respect to the original state. This leads to the following corollary:

Corollary. The meaning of an expression, $M[\![\ e \ ]\!]$, maps stores that are equivalent with respect to e to terminal states that are equivalent with respect to e.

In fact, for all e, the union of the sets $M[\![\ e \ ]\!]\sigma$ for any set of stores that are equivalent with respect to e is *closed* under $\doteq_{FL_{exp}(e)}$. It follows that the meaning of an expression, $M[\![\ e \ ]\!]$, corresponds to a *function* from the equivalence classes of stores σ modulo the permutations that fix $FL_{exp}(e)$ to the equivalence classes of terminal states modulo the permutations that fix $FL_{exp}(e)$.

In practical terms, this means that the semantics of the language are independent of the storage allocation policy employed by the implementation.

4.7 Typeless Semantics

In this section we show that the meaning of an expression does not depend on the type, effect or region information present in the expression. In practical terms, this means that the implementation does not need to perform any type checking, effect checking, or other description computation at run-time.

The proof of this property is fairly routine. We begin by defining a non-standard, *typeless* semantics that parallels the standard semantics exactly. We then show that θ reduces to θ' iff the typeless state that results from erasing all type information from θ reduces to the typeless counterpart of θ'.

The definition of the non-standard semantics consists of two parts: type erasure and typeless reduction. The type erasure function *Erase* is defined as follows:

$$
\begin{aligned}
\mathit{Erase}(c) &= c \\
\mathit{Erase}(x) &= x \\
\mathit{Erase}(\text{LAMBDA } (x{:}\tau)\ e) &= (\text{LAMBDA } x\ \mathit{Erase}(e)) \\
\mathit{Erase}(e_1\ e_2) &= (\mathit{Erase}(e_1)\ \mathit{Erase}(e_2)) \\
\mathit{Erase}(\text{PLAMBDA } (d{:}\kappa)\ e) &= (\text{PLAMBDA } \mathit{Erase}(e)) \\
\mathit{Erase}(\text{PROJ } e\ \delta) &= (\text{PROJ } \mathit{Erase}(e)) \\
\mathit{Erase}(\text{IF } e_1\ e_2\ e_3) &= (\text{IF } \mathit{Erase}(e_1)\ \mathit{Erase}(e_2)\ \mathit{Erase}(e_3)) \\
\mathit{Erase}(\text{BEGIN } e_1 \ldots e_n) &= (\text{BEGIN } \mathit{Erase}(e_1) \ldots \mathit{Erase}(e_n)) \\
\mathit{Erase}(\text{NEW } \rho\ \tau\ e) &= (\text{NEW } \mathit{Erase}(e)) \\
\mathit{Erase}(\text{GET } e) &= (\text{GET } \mathit{Erase}(e)) \\
\mathit{Erase}(\text{SET } e_1\ e_2) &= (\text{SET } \mathit{Erase}(e_1)\ \mathit{Erase}(e_2))
\end{aligned}
$$

We generalize the type erasure function to stores and to states as follows:

$$
\begin{aligned}
\mathit{Erase}(\emptyset) &= \emptyset \\
\mathit{Erase}(\sigma[l \leftarrow v]) &= \mathit{Erase}(\sigma)[l \leftarrow \mathit{Erase}(v)] \\
\mathit{Erase}(\langle e, \sigma \rangle) &= \langle \mathit{Erase}(e), \mathit{Erase}(\sigma) \rangle
\end{aligned}
$$

The typeless reduction relation $\overset{\text{erased}}{\Longrightarrow}$ parallels the standard reduction relation $\overset{\text{red}}{\Longrightarrow}$, and is defined in the same way, by a set of reduction axioms and a set of reduction inference rules. In fact, the axioms and inference rules for $\overset{\text{erased}}{\Longrightarrow}$ are textually the same as for $\overset{\text{red}}{\Longrightarrow}$, except for the three axioms and two inference rules in which description information appears. The erased counterparts of the axioms are:

$$
\begin{aligned}
\langle((\text{LAMBDA } x\ e)\ v), \sigma\rangle &\overset{\text{erased}}{\Longrightarrow} \langle e[v/x], \sigma\rangle \\
\langle(\text{PROJ } (\text{PLAMBDA } e)), \sigma\rangle &\overset{\text{erased}}{\Longrightarrow} \langle e, \sigma\rangle \\
\langle(\text{NEW } e), \sigma\rangle &\overset{\text{erased}}{\Longrightarrow} \langle l, \sigma[l \leftarrow v]\rangle \quad (l \text{ not bound in } \sigma)
\end{aligned}
$$

The erased counterparts of the inference rules are represented by the following contexts:

- polymorphic application: (PROJ [ ])
- allocating a location: (NEW [ ])

Despite the absence of type, effect and region information, typeless reduction proceeds exactly in the same way as typed reduction. Thus, for every reduction sequence in the standard semantics there is a corresponding reduction sequence in the typeless semantics, and vice versa:

Proposition. (Typeless Semantics) Reduction does not make use of type or effect information.

$$\theta \stackrel{\text{red}}{\Longrightarrow} \theta' \qquad \text{iff} \qquad \mathit{Erase}(\theta) \stackrel{\text{erased}}{\Longrightarrow} \mathit{Erase}(\theta')$$

Proof. By inspection of the reduction axioms and inference rules. □

It follows that $M[\![\ e\ ]\!]\sigma$ can be defined as

$$\{\langle v, \sigma' \rangle \mid \mathit{Erase}(\langle e, \sigma \rangle) \stackrel{\text{erased}}{\Longrightarrow} {}^{*}\ \mathit{Erase}(\langle v, \sigma' \rangle)\}$$

instead of

$$\{\langle v, \sigma' \rangle \mid \langle e, \sigma \rangle \stackrel{\text{red}}{\Longrightarrow}{}^{*} \langle v, \sigma' \rangle\}$$

In practical terms, this means that the typeless semantics can serve as the basis for an implementation: after static type checking, which prevents all run-time type errors and all attempts to use undeclared variables or uninitialized locations, there is no more need to keep track of type, effect or region information.

Chapter 5. Private Regions

5.1 Introduction

The language defined in the preceding chapters, *MFX-1*, permits functional program fragments to be incorporated in an imperative program while retaining the benefits of functional programming within those program fragments.

In this chapter we extend *MFX-1* with constructs for declaring and using *private* regions. We call the resulting language *MFX-2*, for Mini-FX level 2. Private regions can be used to integrate imperative program fragments into functional programs while retaining the benefits of functional programming in the surrounding program.

The purpose of this chapter is to explain the notion of private regions and to demonstrate the soundness of the type and effect system in the presence of constructs for declaring and using private regions. We focus on two cases in particular:

- we show how private regions can be used to prove that certain imperative program fragments that have a functional specification but that are implemented imperatively, typically for performance reasons – for example, a program fragment that performs an in-place sort – are functional;
- we show how private regions can be used to prove that certain program fragments that have a functional specification except for allocation but that are implemented imperatively, typically for reasons of expressive power – for example, a program fragment that constructs and returns a circular data structure – are functional except for allocation.

The rest of this chapter is organized as follows. We begin by giving an overview of the characteristics of *MFX-2*. We then present the syntax, informal semantics, and static semantics. Next, we present the dynamic semantics, and we consider the impact of the new language features on type and effect soundness. We conclude with a discussion of the use of region and effect information for storage reclamation.

5.2 Overview

If a region is declared for use by a certain computation, used, and then abandoned, then effects on that region are not visible outside of its scope. Such a region is called a *private* region. In this chapter we introduce two new expressions that take advantage of this principle: the PRIVATE expression and the EXTEND expression.

The PRIVATE expression declares a private, anonymous region for local use that becomes inaccessible when the expression returns. Effects on this region cannot be observed outside of the expression, and need not be reported; we say that such effects can be *masked*. Moreover, the locations belonging to the region can be deallocated when the expression returns. The PRIVATE expression allows functional program fragments to be implemented using imperative constructs.

The EXTEND expression also declares a private, anonymous region for local use, but when it returns it merges this anonymous region with a target region specified by the programmer. Effects on the anonymous region cannot be observed outside of the expression, and can be masked; merging the two regions is tantamount to allocating and initializing zero or more locations in the target region, and therefore has an ALLOC effect on the target region. The EXTEND expression allows functional program fragments to create and return circular data structures without resorting to general recursion equations.

The introduction of private regions invalidates the effect soundness proposition of the previous chapter: reduction of an expression that employs a private region may allocate, read and write locations in that region, even though this is not specified in the effect description of the expression. In this chapter we refine the definition of effect soundness to take account of effects on private regions.

5.3 New Language Features

5.3.1 Syntax

The grammar clauses for the PRIVATE and EXTEND expressions are given below.

Exp =	...	– expressions
	(PRIVATE *Dvar Exp*)	– declaration of a private region
	(EXTEND *Region Dvar Exp*)	– extension of an existing region

5.3.2 Informal Semantics

The semantics of the expression (PRIVATE d e) are as follows: the body e is evaluated with d bound to a fresh region constant, and its value is returned. From the programmer's point of view, this is equivalent to the polymorphic subroutine application

```
(PROJ (PLAMBDA (d:REGION)
        e)
      r)
```

where r is a fresh region constant. The bound region variable d must not occur free in the type of e; this ensures that the private region becomes inaccessible when the expression returns. Since effects on the private region cannot be observed outside of the expression, they can be masked.

The semantics of the expression (EXTEND ρ d e) are as follows. First, the body e is evaluated with d bound to a fresh region constant. Next, the region associated with d is merged with the target region ρ. Finally, the value of e is returned. Since the use of the private region cannot be observed, this is equivalent from the programmer's point of view to the polymorphic subroutine application

```
(PROJ (PLAMBDA (d:REGION)
        e)
      ρ)
```

The bound region variable d may occur free in the type of e; this allows the expression to return values that can refer to locations in the temporary region. After the two regions are merged, these values refer to locations in the target region. Since effects on the region denoted by d cannot be observed outside of the expression, they can be masked.

The following two program fragments should illustrate the use of PRIVATE and EXTEND, and highlight the differences between them. The first expression declares a local region r, allocates a location in r with initial value FALSE, writes TRUE in it, and returns its contents. This expression has type BOOL, and it has effect PURE because the effects on the private region r can be masked.

```
(PRIVATE r
  ((LAMBDA (x:(REF r BOOL))
     (BEGIN
       (SET x TRUE)
       (GET x)))
   (NEW r BOOL FALSE)))
```

The second expression extends the existing region r with an extension called r′, allocates a location in r′ with initial value FALSE, writes TRUE in it, and returns it. The temporary region r′ is private to the expression, so the effects on it can be masked; however, the overall effect of allocating in r must be reported. This expression has type (REF r BOOL), and it has effect (ALLOC r) because the effects on the private region r′ can be masked.

```
(EXTEND r r'
  ((LAMBDA (x:(REF r' BOOL))
     (BEGIN
       (SET x TRUE)
       x))
   (NEW r' BOOL FALSE)))
```

Note that this expression could not have been written using PRIVATE, since the bound region variable r′ appears free in the type of the body.

5.3.3 Free and Bound Variables

The free variables of PRIVATE and EXTEND expressions are the same as the free variables of the corresponding polymorphic subroutine applications:

$$FV(\text{PRIVATE } d\ e) = FV(e)$$
$$FV(\text{EXTEND } \rho\ d\ e) = FV(e)$$

$$FDV_{exp}(\text{PRIVATE } d\ e) = FDV_{exp}(e) - \{d\}$$
$$FDV_{exp}(\text{EXTEND } \rho\ d\ e) = FDV_{desc}(\rho) \cup (FDV_{exp}(e) - \{d\})$$

5.3.4 The Empty Pseudo-region

In order to mask the effects on a private region, we introduce the pseudo-region ψ. Conceptually, ψ corresponds to an empty region, *i.e.* a region to which no locations belong. Thus, (UNION ψ ρ) is convertible with ρ (for all ρ), and the effects (ALLOC ψ), (READ ψ) and (WRITE ψ) are convertible with PURE. The pseudo-region ψ can be regarded as the bottom of the region lattice.

Since the pseudo-region ψ corresponds to the empty set, it is not a region, since a region must correspond to an *infinite* set of locations. The following example should make clear why ψ cannot be treated as a well-formed region description. Consider the following expression:

```
((LAMBDA (x:(REF ψ BOOL))
   (BEGIN
     (SET x TRUE)
     (SET x FALSE)
     (GET x)))
 (NEW ψ BOOL FALSE))
```

The two assignments to x have effect (WRITE ψ), which is convertible with PURE. As a result, it appears as if the expressions do not interfere. However, evaluating the expressions concurrently or out of order may result in a final value of TRUE, which does not agree with the value of FALSE predicted by the standard semantics.

The problem with this example is that the expression (NEW ψ BOOL FALSE) violates the assumption that ψ corresponds to the empty set by allocating a location in ψ. This is prevented by declaring that ψ is *not* a well-formed region description.

When ψ is substituted for a region variable in an *effect* description, the result is always convertible to an effect description that contains no instances of ψ. This is because a region variable in an effect description can appear only as part of an effect description, and any effect on ψ is convertible to PURE. We use this fact as follows: in order to mask the effects on a description variable d in an effect ϵ, we substitute ψ for d in ϵ. For example,

$$\begin{aligned}
(\text{MAXEFF } (\text{READ } d_1)\ (\text{WRITE } d_1))\ [\psi/d_1] &\simeq \text{PURE} \\
(\text{MAXEFF } (\text{READ } d_1)\ (\text{WRITE } d_2))\ [\psi/d_1] &\simeq (\text{WRITE } d_2) \\
(\text{ALLOC } (\text{UNION } d_1\ d_2\ d_3))\ [\psi/d_1] &\simeq (\text{ALLOC } (\text{UNION } d_2\ d_3))
\end{aligned}$$

By contrast, there is no way to eliminate ψ from the type (REF ψ BOOL).

5.3.5 Static Semantics

The type and effect inference rule for the PRIVATE expression is given below. This rule can be read as follows. First, determine the type and the effect of e under a kind assignment in which d denotes a region. Next, verify that d is bindable in e. Finally, verify that d does not appear free in the type of e. When all goes well, the type and effect of the expression are the same as the type and effect of e, except that all effects on d are masked.

$$\frac{\begin{array}{c} A, B[d \leftarrow \text{REGION}] \vdash e : \tau \\ A, B[d \leftarrow \text{REGION}] \vdash e\,!\,\epsilon \\ \forall x \in FV(e)\ .\ d \notin FDV_{desc}(A(x)) \\ d \notin FDV_{desc}(\tau) \end{array}}{\begin{array}{lll} A, B \vdash (\text{PRIVATE } d\ e) & : & \tau \\ A, B \vdash (\text{PRIVATE } d\ e) & ! & \epsilon[\psi/d] \end{array}}$$

As might be expected, this rule resembles a composition of the rules for polymorphic abstraction and polymorphic application, as applied to the polymorphic subroutine application

```
(PROJ (PLAMBDA (d:REGION)
        e)
      ψ)
```

In fact, the only difference is the additional premise that requires that d not be free in the type of e. This ensures that the private region becomes inaccessible when the expression returns.

Note that the type of the expression, τ, is equal to $\tau[\psi/d]$ because d does not appear free in τ. Thus, the type and the effect of the expression are constructed in the same way. The restriction $d \notin FDV_{desc}(\tau)$ is necessary and sufficient to ensure that $\tau = \tau[\psi/d]$.

We now turn to the type and effect inference rule for the EXTEND expression, which is given below. The rule can be read as follows. First, verify that ρ has kind REGION, and determine the type and the effect of e under a kind assignment in which d denotes a region. Next, verify that d is bindable in e. When all goes well, the type and effect of the expression are the same as the type and effect of e, except that all occurrences of d in the type are replaced by ρ, all effects on d are masked, and an ALLOC effect on ρ is added.

$$\frac{\begin{array}{c} B \vdash \rho : \text{REGION} \\ A, B[d \leftarrow \text{REGION}] \vdash e : \tau \\ A, B[d \leftarrow \text{REGION}] \vdash e \,!\, \epsilon \\ \forall x \in FV(e) \,.\, d \notin FDV_{desc}(A(x)) \end{array}}{\begin{array}{lll} A, B \vdash (\text{EXTEND } \rho\ d\ e) & : & \tau[\rho/d] \\ A, B \vdash (\text{EXTEND } \rho\ d\ e) & ! & (\text{MAXEFF } \epsilon[\psi/d]\ (\text{ALLOC } \rho)) \end{array}}$$

As might be expected, this rule resembles a composition of the rules for polymorphic abstraction and polymorphic application, as applied to the polymorphic subroutine application

```
(PROJ (PLAMBDA (d:REGION)
        e)
      ρ)
```

In fact, the only difference is that the effects on ρ, whatever they may be, are masked and replaced by an ALLOC effect.

5.3.6 Aliasing

Although a PRIVATE expression is equivalent to a polymorphic application to a region, no aliasing is introduced because the region in question is fresh, and disjoint from all other regions past, present or future.

In the case of the EXTEND expression, however, there is a possibility of aliasing because the bound region variable becomes an alias for the target region when the expression returns. In the current *MFX* language, aliasing between the bound variable and the target region is prevented by a syntax restriction that forces the programmer to use a single description variable to denote both the bound region variable and the target region. This makes the target region inaccessible in the body.

5.4 Dynamic Semantics

5.4.1 Auxiliary Expressions

The simplest way to define the semantics of the new expressions would be with the following very simple reduction axioms, where r denotes a fresh region constant:

$$\langle(\text{PRIVATE } d\ e), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e[r/d], \sigma\rangle$$
$$\langle(\text{EXTEND } \rho\ d\ e), \sigma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e[\rho/d], \sigma\rangle$$

Unfortunately, these reduction axioms would invalidate the type preservation proposition by removing the expression that masks the effects on the private region. Since type preservation is the foundation of our type and effect soundness propositions, we have developed a technique for reducing expressions such as PRIVATE and EXTEND while maintaining the type preservation property.

The semantics of the PRIVATE and EXTEND expressions are defined in terms of *auxiliary expressions*. The basic technique is this: instead of being reduced directly using the beta-substitutions given above, a PRIVATE and EXTEND expression is reduced to a corresponding *auxiliary expression*. During this initial reduction step, a fresh region constant is chosen and embedded in the auxiliary expression. While the body of the auxiliary expression is reduced recursively, the auxiliary expression serves as a reminder that the chosen region constant is private to the expression, and that any effects on it can therefore be masked. When the body has been reduced to a value, there are no more effects to be masked and the auxiliary expression can be reduced to its body.

The auxiliary expressions for PRIVATE and EXTEND are *PRIVATE* and *EXTEND* respectively. Their syntax is given below.

(*PRIVATE* *Rconst Exp*) – PRIVATE in progress
(*EXTEND* *Region Rconst Exp*) – EXTEND in progress

In each expression, the region constant represents the private, anonymous region. In the *EXTEND* expression, the region description identifies the target region.

The *PRIVATE* and *EXTEND* expressions are *binding* expressions: in each case, the region constant that represents the private region takes the place of the bound variable in the corresponding PRIVATE or EXTEND expression. To facilitate a generalization later on, we use the term *auxiliary binding expressions* to refer to *PRIVATE* and *EXTEND* expressions.

The type and effect inference rule for the *PRIVATE* expression is given below. It is derived directly from the rule for the PRIVATE expression. The

rule requires that the body be well-formed, and that the private region not be free in the type of the body. The effects on the private region are masked.

$$\frac{\begin{array}{c} A, B \vdash e : \tau \\ A, B \vdash e \,!\, \epsilon \\ r \notin FRC_{desc}(\tau) \end{array}}{\begin{array}{lcl} A, B \vdash (\text{*PRIVATE*}\ r\ e) & : & \tau \\ A, B \vdash (\text{*PRIVATE*}\ r\ e) & ! & \epsilon[\psi/r] \end{array}}$$

The type and effect inference rule for the *EXTEND* expression is given below. This rule is derived directly from the rule for the EXTEND expression. To obtain the type and effect of the expression, all occurrences of r in the return type are replaced by ρ, any effects on r are masked, and an ALLOC effect on ρ is introduced.

$$\frac{\begin{array}{c} B \vdash \rho : \text{REGION} \\ A, B \vdash e : \tau \\ A, B \vdash e \,!\, \epsilon \end{array}}{\begin{array}{lcl} A, B \vdash (\text{*EXTEND*}\ \rho\ r\ e) & : & \tau[\rho/r] \\ A, B \vdash (\text{*EXTEND*}\ \rho\ r\ e) & ! & (\text{MAXEFF}\ \epsilon[\psi/r]\ (\text{ALLOC}\ \rho)) \end{array}}$$

5.4.2 States

The reduction axioms for the PRIVATE and EXTEND expressions must pick fresh region constants that can be used to identify the private, anonymous regions. To ensure that a recursive reduction does not pick a region constant that is in use in the surrounding context, we augment the state of a computation with a list of the region constants that are in use. To facilitate a generalization later on, this list is represented as a finite partial function from region constants to the singleton set {USED}.

Definition. A *region map* γ is a finite partial function with signature $Rconst \rightarrow \{\text{USED}\}$.

Definition (revised). A *state* θ is a tuple $\langle e, \sigma, \gamma \rangle$ consisting of an expression, a store, and a region map.

The existing reduction axioms and inference rules do not require any substantive change to accommodate this revised definition of states, since they neither use nor alter the region map. To accommodate the new definition of states, each reduction axiom of the form

$$\langle e, \sigma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma' \rangle$$

must be replaced by a corresponding axiom of the form

$$\langle e, \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma', \gamma \rangle$$

Note that γ appears on both sides of the axiom: the region map is not involved in these reductions.

As for the reduction inference rules, each rule of the form

$$\frac{\langle e, \sigma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma' \rangle}{\langle C[e], \sigma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle C[e'], \sigma' \rangle}$$

must be replaced by a corresponding inference rule of the form

$$\frac{\langle e, \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma', \gamma' \rangle}{\langle C[e], \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle C[e'], \sigma', \gamma' \rangle}$$

5.4.3 Reduction Axioms and Inference Rules

The semantics of the new expressions are defined by four new reduction axioms, one for each expression, and two new reduction inference rules, one for each auxiliary expression.

The reduction axioms for PRIVATE and EXTEND, shown below, are quite simple: each rule simply chooses an unused region constant, marks it as USED, and replaces the expression by the corresponding auxiliary expression after substituting this region constant for the bound region variable throughout its scope.

$$\langle (\text{PRIVATE } d\ e), \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle (\text{*PRIVATE*}\ r\ e[r/d]), \sigma, \gamma[r \leftarrow \text{USED}] \rangle$$
$$\langle (\text{EXTEND } \rho\ d\ e), \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle (\text{*EXTEND*}\ \rho\ r\ e[r/d]), \sigma, \gamma[r \leftarrow \text{USED}] \rangle$$
(r not bound in γ)

When the body of a *PRIVATE* expression has been reduced to a value (by means of the reduction inference rules given below), the *PRIVATE* expression, which serves to mask the effects on the private region, is no longer needed and can be reduced to a value.

$$\langle (\text{*PRIVATE*}\ r\ v), \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle v, \sigma, \gamma \rangle$$

The private region constant r is not made unused because references to this region may still exist, both in the value v and in the store σ. However, if the state is well-formed, any remaining references to the region can never occur as the first argument to an active GET or SET expression.

When the body of an *EXTEND* expression has been reduced to a value, the expression can be reduced to that value while, simultaneously, the region description ρ is substituted for all instances of the private region constant r throughout v and σ. Since every trace of the region constant r is erased, r could be made unused in the region map; we will not make use of this.

$$\langle (\text{*EXTEND*}\ \rho\ r\ v), \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle v[\rho/r], \sigma[\rho/r], \gamma \rangle$$

The new reduction inference rules, which show how to reduce the body of a *PRIVATE* or *EXTEND* expression to a value, are represented by the following contexts:

- PRIVATE in progress: (*PRIVATE* r [ ])
- EXTEND in progress: (*EXTEND* $\rho\ r$ [ ])

5.5 Type Soundness

Because of the introduction of effect masking, the proof of type soundness no longer hold as originally formulated; in particular, the effect propagation lemma must be modified to deal with effect masking. In what follows, we refine the definitions and propositions that are affected by the introduction of private regions.

We begin be refining the notion of a *consistent* state.

Definition. A region constant r *occurs* in a state $\langle e, \sigma, \gamma \rangle$ iff it occurs in the expression component e, the store component σ, or the region map γ. The region constants that occur in a state are given by the function FRC_{state}, which is formally defined below.

$$FRC_{store}(\sigma) = \bigcup_{l \in Dom(\sigma)} FRC_{exp}(\sigma(l))$$

$$FRC_{state}(\langle e, \sigma, \gamma \rangle) = FRC_{exp}(e) \cup FRC_{store}(\sigma) \cup Dom(\gamma)$$

Definition (revised). A state is *consistent* , $C(\langle e, \sigma, \gamma \rangle)$, iff every location that occurs in the state is bound in the store and has the same tags everywhere, and every region constant that occurs in the state is bound in the region map, *i.e.* iff $FL_{sta}(\langle e, \sigma, \gamma \rangle) \subseteq Dom(\sigma)$, $FRC_{state}(\langle e, \sigma, \gamma \rangle) \subseteq Dom(\gamma)$, and $l_{\rho,\tau} \in FL_{sta}(\langle e, \sigma \rangle)$ and $l_{\rho',\tau'} \in FL_{sta}(\langle e, \sigma \rangle)$ implies that $\rho = \rho'$ and $\tau = \tau'$.

In a well-formed state, the private, anonymous region of an auxiliary binding expression must be accessible only within said expression. The following definition helps define this more precisely.

Definition. A region constant r is *accessible* in a context C, $r \in Acc(C)$, iff it appears in the effect of any active expression in that context, *i.e.* iff some active expression in C has an effect ϵ such that $r \in FRC_{desc}(\epsilon)$.

In a well-formed state, a region constant that is accessible only within a given expression is inaccessible after that expression is reduced to a value. This is true because the expression itself has effect PURE after it has been reduced to a value, and the region constant cannot become accessible anywhere else because the effects of all active expressions propagate to the outermost expression, whose effect is always preserved or decreased.

A second invariant of well-formed states is that an auxiliary binding expression must occur only as an active expression, *i.e.* an expression whose reduction is in progress. This ensures that such an expression is never duplicated (as a result of ordinary application or of reading a location).

As a summary of the invariants relating to auxiliary binding expressions, we define the notion of a *legal* state.

Definition. A state $\theta = \langle e, \sigma, \gamma \rangle$ is *legal* , $\mathcal{L}(\theta)$, iff

1. every auxiliary expression in θ is active;
2. no two auxiliary binding expressions in θ have the same private region constant; and
3. the private region constant of an auxiliary binding expression is inaccessible in the context that surrounds the expression, *e.g.* if the expression e is of the form $C[(\text{*PRIVATE*}\ r\ e')]$ then $r \notin Acc(C)$.

We can now restore the validity of the type soundness and static typing propositions in the presence of private, anonymous regions by revising the definition of well-formed states as follows.

Definition (revised). A state $\theta = \langle e, \sigma, \gamma \rangle$ is *well-formed*, $\mathcal{WF}_{state}(\theta)$, iff it is consistent and legal and e and σ are both well-formed. In other words,

$$\mathcal{WF}_{state}(\theta) \quad \Leftrightarrow \quad \mathcal{C}(\theta) \wedge \mathcal{L}(\theta) \wedge \mathcal{WF}_{exp}(e) \wedge \mathcal{WF}_{store}(\sigma)$$

Lemma (revised). (Effect Propagation) In a well-formed state, the effect of each active expression is a subeffect of the effect of its parent expression unless this parent expression is an auxiliary binding expression, in which case the effects on its private region constant are masked.

Proof. The proof proceeds as before, using the correspondence between the new reduction inference rules and the new effect inference rules. In particular, the effect of a *PRIVATE* or *EXTEND* expression is obtained from the effect of its body by masking the effects on its private region constant. □

Proposition (revised). (Type and Effect Preservation) Reduction of a well-formed state preserves or decreases the type and effect descriptions of the state.

$$\begin{array}{c} \mathcal{WF}_{state}(\langle e, \sigma, \gamma \rangle) \\ e : \tau \\ e \,!\, \epsilon \\ \langle e, \sigma, \gamma \rangle \stackrel{\text{red}}{\Longrightarrow} \langle e', \sigma', \gamma' \rangle \end{array} \quad \Rightarrow \quad \begin{array}{c} \mathcal{WF}_{state}(\langle e', \sigma', \gamma' \rangle) \\ e' : \tau' \text{ where } \tau' \sqsubseteq \tau \\ e' \,!\, \epsilon' \text{ where } \epsilon' \sqsubseteq \epsilon \end{array}$$

Proof. The proof proceeds as before, but there are several new cases to be considered, as well as some new properties whose preservation must be verified.

It is easy to see that reduction preserves the new, stronger notion consistency: new region constants are introduced only by the reduction axioms for PRIVATE and EXTEND, each of which binds the new region constant in the region map. Since a region constant bound in the region map never becomes unbound, reduction preserves consistency.

To show that reduction preserves legality, we consider each property of legal states in turn.

- The first property, activeness, is preserved because (i) every auxiliary expression is active when it is created, and (ii) an active expression remains active until it becomes a value.

- The second property, uniqueness, is preserved because (i) every auxiliary expression has a unique private region constant when it is created, and (ii) every auxiliary expression is active, and therefore cannot be duplicated.
- The third property, locality, is preserved because (i) the private region constant chosen when an auxiliary expression is initially only accessible within the expression, and (ii) this remains true by virtue of effect preservation and the revised effect propagation lemma. □

5.6 Effect Soundness

We are left with the task of restoring the effect soundness proposition, which has been invalidated by the introduction of private regions. For example, if a state θ ! ϵ contains an active expression of the form (*PRIVATE* r e), then reduction of θ may have effects on the region r, even though r does not appear in ϵ.

Below, we present a revised effect soundness proposition that relates the syntactic effect of each expression to the actual effects on the regions that are accessible in the context surrounding the expression. Although this proposition is weaker than the original proposition, its power to identify opportunities for concurrent evaluation and memoization has not diminished because effects on private regions do not inhibit these optimizations.

Rather than considering the effects of a reduction as a whole, the revised effect soundness proposition considers the individual active expressions that participate in a reduction, and focuses on the actual effect of the reduction on only those regions that are accessible in the context surrounding each active expression.

Proposition (revised). (Effect Soundness) Reduction of an expression in a well-formed state allocates, reads, and writes only locations that can be reached through the regions specified by its effect and/or through regions that are accessible only within the expression. In other words, if $\theta = \langle C[e], \sigma, \gamma \rangle$, $\theta' = \langle C[e'], \sigma', \gamma' \rangle$ and $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$, and e ! ϵ where

$$\epsilon \simeq (\text{MAXEFF } (\text{ALLOC } \rho_A) \ (\text{READ } \rho_R) \ (\text{WRITE } \rho_W))$$

then

$$\begin{aligned} \mathcal{A}(\theta, \theta') \cap \mathit{Reach}(\mathit{Acc}(C)) &\subseteq \mathit{Reach}(\rho_A) \\ \mathcal{R}(\theta, \theta') \cap \mathit{Reach}(\mathit{Acc}(C)) &\subseteq \mathit{Reach}(\rho_R) \\ \mathcal{W}(\theta, \theta') \cap \mathit{Reach}(\mathit{Acc}(C)) &\subseteq \mathit{Reach}(\rho_W) \end{aligned}$$

Proof. As before, using the revised effect propagation lemma. □

This proposition generalizes immediately to $\theta \stackrel{\text{red}}{\Longrightarrow}{}^* \theta'$.

In practical terms, the revised effect soundness proposition implies that it is possible to integrate imperative program fragments into functional programs, while retaining the benefits of functional programming in the surrounding program, provided that the *interface* of the imperative program fragments is entirely functional.

It would be possible to extend *MFX* with a region constant, such as UNIVERSE, of which every region is a subregion, and with an effect constant, such as PROCEDURE, of which every effect is a subeffect. This would change the language in certain predictable respects: for example, an effect such as (WRITE UNIVERSE) would interfere with any effect. However, the effect masking constructs PRIVATE and EXTEND depend critically on the fact that an identifier or expression can (be used to) access the locations in a region r only if r appears free in the type or the effect of the identifier or expression. If it were possible to coerce a reference of type (REF r BOOL) to type (REF UNIVERSE BOOL), or to coerce a subroutine of type (SUBR (BOOL) (WRITE r) BOOL) to type (SUBR (BOOL) (WRITE UNIVERSE) BOOL) or even (SUBR (BOOL) PROCEDURE BOOL), then the import restriction of EXTEND and the export restriction of PRIVATE would have to be modified to prohibit the import and export of any identifier or value in the type of which either UNIVERSE or PROCEDURE appears free. This would make these two constants much less useful. We have therefore decided not to include them in *MFX*.

5.7 Storage Reclamation

The practice of optimizing a program by identifying opportunities for storage reclamation statically is known as compile-time garbage collection or stack-consing [Hud86]. In this section we describe how effect and region information can be used to assist in compile-time garbage collection.

In a well-formed program, the private region of a PRIVATE expression becomes inaccessible after the expression returns. In practice, this means that the locations in the private region can be reclaimed. For example, consider the expression

```
(PRIVATE r
  ((LAMBDA (x:(REF r BOOL))
     (GET x))
   (NEW r BOOL TRUE)))
```

which chooses a new region constant, allocates a location initialized to TRUE in the corresponding region, and returns the contents of the location (which, of course, is TRUE). Since the location is inaccessible after the PRIVATE expression returns, it can in theory be reclaimed.

Although there have been earlier proposals for compile-time garbage collection, none of the standard methods are effective when procedures and locations are first-class values. This may discourage the use of procedural abstraction, which is contrary to our philosophy.

In *MFX-2*, the type and effect system provides information that makes compile-time garbage collection feasible even when procedures and locations are first-class values. Compile-time garbage collection based on effect and

region information is more powerful than existing methods, in that it permits the reclamation of inaccessible locations even when references to those locations continue to exist. This is illustrated below by a rather trivial example; we ask the reader to imagine more realistic examples, perhaps involving aggregate data structures. Consider the expression

```
(PRIVATE r
  ((LAMBDA (x:(REF r BOOL))
     (LAMBDA (y:BOOL)
       (BEGIN
         x
         y)))
   (NEW r BOOL TRUE)))
```

which chooses a new region constant, allocates a location initialized to TRUE in the corresponding region, and returns a subroutine of one argument that imports the location but discards it and returns its argument. This expression reduces to a value of the form

```
(LAMBDA (y:BOOL)
  (BEGIN
    l
    y))
```

where l is a location that belongs to the private region, which we will call r. Both the original expression and the resulting value have type

```
(SUBR (BOOL) PURE
  BOOL)
```

and effect PURE. Although the final value does contain a reference to l, it is clear that this reference is never used to access the location. In fact, this is true regardless of how complex the final value is, since l belongs to the region corresponding to r and the type and effect inference rule for the PRIVATE expression ensures that r does not appear in the type, and hence not in the latent effect, of the final value. Thus, the location l, and in general all locations that belong to the region corresponding to r, can safely be reclaimed when the PRIVATE expression returns, even when references to those locations continue to exist.

When this scheme is used together with a conventional garbage collector, it may be necessary to modify the latter to ensure that it does not trace or copy locations, such as l above, that are inaccessible and have been reclaimed. It appears that this can be done by maintaining a list or map of the locations that have been reclaimed, and erasing all references to these locations as they are encountered during the sweep or copy phase of the conventional garbage collector. When all references to the locations have been erased, the locations can be reused.

Chapter 6. Explicit Concurrency

6.1 Introduction

The languages defined in the preceding chapters, *MFX-1* and *MFX-2*, are *sequential* languages: according to the standard semantics, expressions are evaluated in left-to-right, applicative order. This ensures that the language semantics specify a unique result for every computation, but it limits the amount of concurrency.

In this chapter we extend *MFX-2* with constructs for introducing and managing *explicit* concurrency. We call the resulting language *MFX-3*, for Mini-FX level 3. Explicit concurrency can be used to increase the concurrency in a program at the expense of determinacy, as has been pointed out by Brinch-Hansen and others [Bri72]. As is customary for a concurrent language, the semantics of *MFX-3* do not specify precisely how the evaluations of concurrent expressions are interleaved. Consequently, the language semantics do not specify a unique result for every computation.

The purpose of this chapter is to demonstrate that a type and effect system can support explicit concurrency in several ways:

- we show how regions can be used to ensure that the data associated with a monitor can only be accessed within a *critical section* for that monitor, even if pointers to the data can be passed freely outside of the monitor;
- we show how the type and effect system can be used to ensure that all interactions between concurrent expressions are mediated by monitors and critical sections; and
- we show how program fragments that use explicit concurrency can be integrated into functional programs while retaining the benefits of functional programming in the surrounding program.

The rest of this chapter is organized as follows. We begin by giving an overview of the characteristics of *MFX-3*. We then present the syntax, informal semantics, and static semantics. Next, we present the dynamic semantics, and we conclude by considering the impact of the new language features on type and effect soundness.

6.2 Overview

In this chapter we introduce a new expression, COBEGIN, that allows the programmer to indicate that two or more expressions are to be evaluated concurrently. Semantically, concurrent evaluation of two or more expressions means that the computations of the expressions may be interleaved in arbitrary order.

The result of a concurrent computation may in general depend on how the computations are interleaved: for example, if two concurrent expressions write distinct values to the same location, then the final value depends on which expression was the last to write. To give the programmer some control over the way in which computations are interleaved, we introduce an expression, MONITORED, for declaring *monitored regions*, as well as an expression, EXCLUSIVE, for evaluating an expression as a *critical section* for a given monitored region. The language definition guarantees mutual exclusion between all evaluations of the critical sections for a given monitored region.

We have adopted the view that "intelligible programming requires all interactions between parallel processes to be mediated by some mechanism such as a critical section or monitor" [Rey78, p. 40]. Such a discipline helps not only the programmer, but also the compiler since uncontrolled interference between concurrent computations tends to inhibit optimizations such as CM and concurrent evaluation.

We have devised a set of effect restrictions that *enforce* the above discipline. These restrictions, which are part of the type and effect inference rules for the COBEGIN, MONITORED and EXCLUSIVE expressions, ensure that two or more expressions are evaluated concurrently only if all interactions between them are mediated by monitors. Because of these restrictions, the evaluation of a well-formed program never encounters a race condition between two attempts to read or write a certain location; race conditions occur only between attempts to enter critical sections. We believe that this contributes greatly to "intelligible programming", however elusive a concept that may be.

6.3 New Language Features

6.3.1 Syntax

In order to extend the language to deal with concurrency, we introduce one new effect constructor and one new type constructor.

Effect =	...	– effect descriptions
	(MCALL *Region*)	– effect of accessing a given region in exclusive mode
Type =	...	– type descriptions
	(MONITOR *Region*)	– types of monitors

The type description (MONITOR ρ) corresponds to the monitor for the region ρ. In a well-formed program, there is at most one monitor for any region.

The grammar clauses for the COBEGIN, MONITORED and EXCLUSIVE expressions are given below.

Exp =	...	– expressions
	(COBEGIN *Exp**)	– concurrent evaluation
	(MONITORED *Dvar Var Exp*)	– declaration of a monitored region
	(EXCLUSIVE *Exp Exp*)	– critical section

6.3.2 Informal Semantics

The semantics of the expression (COBEGIN $e_1 \ldots e_n$) are as follows: the expressions $e_1 \ldots e_n$ are evaluated concurrently, and NIL is returned when they have all terminated. The expressions must have effects that do not interfere with one another except through MCALL effects. This restriction, which is formally specified in the type and effect inference rule given below, ensures that all interaction between concurrent expressions is serialized by means of critical sections. For a more detailed discussion of interference in the presence of MCALL effects, see Chapter 8.

The semantics of the expression (MONITORED d x e) are as follows: the body e is evaluated with d bound to a fresh region constant and x bound to the corresponding monitor, and its value is returned. From the programmer's point of view, this is equivalent to the expression

```
((PROJ (PLAMBDA (d:REGION)
         (LAMBDA (x:(MONITOR d))
           e))
       r)
 m)
```

where r is a fresh region constant and m the corresponding monitor. The bound region variable d must not occur free in the type of e; this ensures that neither the monitored region nor the monitor itself are accessible in the surrounding scope. Moreover, all READ and WRITE effects on d in e must be encapsulated within EXCLUSIVE expressions; this ensures that the monitored region is accessed only within the critical sections for that region. This restriction could be relaxed at the expense of complicating the effect restriction on COBEGIN expressions. Since effects on the monitored region cannot be observed outside of the expression, they need not be reported.

Since the MONITORED construct introduces a region and the corresponding monitor simultaneously, there can never be more than one monitor for any given region. This ensures that any expression that acquires the monitor for a region has exclusive access to that region until it releases the monitor.

The semantics of the expression (EXCLUSIVE e_1 e_2) are as follows. First, e_1 is evaluated. Provided that e_1 evaluates to a monitor, the monitor is acquired, e_2 is evaluated, the monitor is released, and the value of e_2 is returned. In other words, e_2 is evaluated as a critical section for the monitored region corresponding to the value of e_1. Note that the acquisition of the monitor is delayed if the monitor is not idle.

In order to keep the semantics simple, we have decided not to guarantee fair scheduling. Fair scheduling traditionally means that any active expression that is not stuck is guaranteed to make progress eventually. The semantics of *MFX-3*, by contrast, guarantees only that as long as there is at least one active expression that can make progress, some active expression will make progress. From our point of view, this has two advantages over fair scheduling: its semantics are easier to express, and it permits a non-preemptive uniprocessor implementation.

The following program fragments illustrate the use of the new expressions. The first example illustrates the use of the COBEGIN expression. It declares a local monitored region r with monitor m, allocates a location in r initialized to FALSE, writes the values TRUE and FALSE to the location in unspecified order, and returns the contents of the location. The expression has type BOOL and effect PURE: the monitored region r is private to the expression, so the effects on it need not be reported.

```
(MONITORED r m
  ((LAMBDA (x:(REF r BOOL))
     (BEGIN
       (COBEGIN
         (EXCLUSIVE m
           (SET x TRUE))
         (EXCLUSIVE m
           (SET x FALSE)))
       (EXCLUSIVE m
```

```
        (GET x))))
    (NEW r BOOL FALSE)))
```

Even though this expression may return different values if it is evaluated more than once, the compiler can safely memoize it because it is not *guaranteed* to return different values. In fact, it is very well possible that this expression always returns the same value in a given concurrent implementation.

The next example shows how to use the EXCLUSIVE expression to implement atomic operations. For this example we will assume that `int` represents the type of integers, that the literals `0`, `1` and `2` represent the corresponding integers, and that `+` represents a subroutine that performs integer addition. The expression declares a local monitored region `r` with monitor `m`, allocates a location in `r` with initial value `0`, increments the contents of this location twice, and returns its contents. This expression has type `int` and effect PURE.

```
(MONITORED r m
  ((LAMBDA (x:(REF r int))
     (BEGIN
       (COBEGIN
         (EXCLUSIVE m
           (SET x (+ (GET x) 1)))
         (EXCLUSIVE m
           (SET x (+ (GET x) 1))))
       (EXCLUSIVE m
         (GET x))))
   (NEW r int 0)))
```

This example also shows how to use critical sections and associative operators (such as addition) to implement deterministic programs using nondeterministic language constructs: the above expression always returns the value 2.

In the above expression, each increment operation accesses the contents of the region `r` twice: once to read the contents of `x` and once to write the new value. If these operations were interleaved without any synchronization, one of the updates could be lost. However, the EXCLUSIVE expression, which acquires the monitor and holds it while it evaluates its body, ensures mutual exclusion and guarantees that each increment operation is evaluated atomically. If, for some reason, a non-atomic increment operation is desired, the subexpression

```
(EXCLUSIVE m
  (SET x (+ (GET x) 1)))
```

in the expression above can be replaced by

```
((LAMBDA (old-value:int)
   (EXCLUSIVE m
     (SET x (+ old-value 1))))
 (EXCLUSIVE m
   (GET x)))
```

Finally, note that the new expressions make the PRIVATE construct superfluous: from the programmer's point of view, the expression (PRIVATE x e) is equivalent to the expression

```
(MONITORED x m
  (EXCLUSIVE m
    e))
```

where m is any variable not free in e. It would be possible to define a monitored version of EXTEND as well, but we decided not to do so in order to keep the language simple.

6.3.3 Free and Bound Variables

The free and bound variables of the new expressions and descriptions are defined as usual; the following definitions are supplied for completeness only.

The free ordinary variables of the new expressions are defined below.

$$FV(\text{COBEGIN } e_1 \ldots e_n) = FV(e_1) \cup \ldots \cup FV(e_n)$$
$$FV(\text{MONITORED } d\ x\ e) = FV(e) - \{x\}$$
$$FV(\text{EXCLUSIVE } e_1\ e_2) = FV(e_1) \cup FV(e_2)$$

The free description variables of the new descriptions are defined below.

$$FDV_{desc}(\text{MCALL } \rho) = FDV_{desc}(\rho)$$
$$FDV_{desc}(\text{MONITOR } \rho) = FDV_{desc}(\rho)$$

The free description variables of the new expressions are defined below.

$$FDV_{exp}(\text{COBEGIN } e_1 \ldots e_n) = FDV_{exp}(e_1) \cup \ldots \cup FDV_{exp}(e_n)$$
$$FDV_{exp}(\text{MONITORED } d\ x\ e) = FDV_{exp}(e) - \{d\}$$
$$FDV_{exp}(\text{EXCLUSIVE } e_1\ e_2) = FDV_{exp}(e_1) \cup FDV_{exp}(e_2)$$

The free region constants of the new descriptions and expressions are determined in the same way as the free description variables.

6.3.4 Description Inclusion and Kind Inference

The new effect constructor MCALL has the same distributive properties as ALLOC, READ and WRITE:

$$(\text{MCALL } (\text{UNION } \rho_1 \ldots \rho_n)) \simeq (\text{MAXEFF } (\text{MCALL } \rho_1) \ldots (\text{MCALL } \rho_n))$$

This gives rise to the following derived rule:

$$\frac{\rho \sqsubseteq \rho'}{(\text{MCALL } \rho) \sqsubseteq (\text{MCALL } \rho')}$$

There are no description inclusion rules for MONITOR types, and only a single description conversion rule:

$$\frac{\rho \simeq \rho'}{(\text{MONITOR } \rho) \simeq (\text{MONITOR } \rho')}$$

To see why MONITOR is not monotonic in its region component, consider the following example. If the type (MONITOR `r1`) were a subtype of (MONITOR (UNION `r1 r2`)), then the monitor for the region `r1` could be mistaken for a monitor for the region (UNION `r1 r2`). This would create the false impression that acquiring the monitor would ensure exclusive access to the region (UNION `r1 r2`), which is not true. In particular, if the monitor for the region `r2` were similarly mistaken for a monitor for the region (UNION `r1 r2`), then there would be two monitors claiming to grant exclusive access to the same region, which is impossible. Hence (MONITOR ρ) is not monotonic in ρ.

The kind inference rules for the new descriptions are routine, and are given for completeness only. The first rule states that if ρ has kind REGION, then the description (MCALL ρ) has kind EFFECT.

$$\frac{B \vdash \rho : \text{REGION}}{B \vdash (\text{MCALL } \rho) : \text{EFFECT}}$$

Similarly, the rule for MONITOR type descriptions states that if ρ has kind REGION, then the description (MONITOR ρ) has kind TYPE.

$$\frac{B \vdash \rho : \text{REGION}}{B \vdash (\text{MONITOR } \rho) : \text{TYPE}}$$

The MONITORED expression introduces monitors with types of the form (MONITOR d), where d is a description variable. Since these types cannot be coerced to types of the form (MONITOR (UNION ...)), it follows that any such type is empty. It would be possible to define the language so that these types were considered syntactically invalid or ill-formed; however, this would create complications since kind-respecting beta-substitution would no longer preserve syntactic validity and well-formedness. We therefore opt for the current approach, in which certain MONITOR types are simply empty.

6.3.5 Static Semantics

There are three new type and effect inference rules, one for each new expression. The type and effect inference rule for the COBEGIN expression can be read as follows: provided that each subexpression is well-formed, and provided that the effect descriptions of the subexpressions do not contain interfering READ and WRITE effects (either manifestly or hidden in effect variables), the type description of a COBEGIN expression is UNIT, and its effect description is the least upper bound of the effect descriptions of the subexpressions.

$$\frac{\begin{array}{c}\forall i, 1 \le i \le n \;.\; A, B \vdash e_i : \tau_i \\ \forall i, 1 \le i \le n \;.\; A, B \vdash e_i \,!\, \epsilon_i \\ \forall i \ne j, \rho, d, d' \;.\; \left\{\begin{array}{l} d \sqsubseteq \epsilon_i \\ \text{or} \\ (\textsc{write}\ \rho) \sqsubseteq \epsilon_i \end{array}\right. \Rightarrow \left\{\begin{array}{l} d' \not\sqsubseteq \epsilon_j \\ (\textsc{read}\ \rho) \not\sqsubseteq \epsilon_j \\ (\textsc{write}\ \rho) \not\sqsubseteq \epsilon_j \end{array}\right.\end{array}}{\begin{array}{lll} A, B \vdash (\textsc{cobegin}\ e_1 \ldots e_n) & : & \textsc{unit} \\ A, B \vdash (\textsc{cobegin}\ e_1 \ldots e_n) & ! & (\textsc{maxeff}\ \epsilon_1 \ldots \epsilon_n) \end{array}}$$

Note that interfering MCALL effects are allowed; in fact, they are the only way in which the branches of a COBEGIN expression can interact. For a derivation of the interference restriction, see Chapter 8.

The type and effect inference rule for the MONITORED expression is given below. This rule is identical to the rule for the PRIVATE expression except in two respects: (i) the type of the body is determined in a type assignment in which the extra variable of the MONITORED expression is bound to an appropriate monitor type; and (ii) the body must not have any READ or WRITE effect on the monitored region. This effect restriction ensures that the monitored region is accessed only from within critical sections.

$$\frac{\begin{array}{c} A[x \leftarrow (\textsc{monitor}\ d)], B[d \leftarrow \textsc{region}] \vdash e : \tau \\ A[x \leftarrow (\textsc{monitor}\ d)], B[d \leftarrow \textsc{region}] \vdash e \,!\, \epsilon \\ \forall x' \in (FV(e) - \{x\}) \;.\; d \notin FDV_{desc}(A(x')) \\ d \notin FDV_{desc}(\tau) \\ (\textsc{read}\ d) \not\sqsubseteq \epsilon \;\wedge\; (\textsc{write}\ d) \not\sqsubseteq \epsilon \end{array}}{\begin{array}{lll} A, B \vdash (\textsc{monitored}\ d\ x\ e) & : & \tau \\ A, B \vdash (\textsc{monitored}\ d\ x\ e) & ! & \epsilon[\psi/d] \end{array}}$$

Naturally, MCALL effects on the monitored region are allowed. ALLOC effects on the monitored region are also allowed: since the allocation and initialization of new locations cannot interfere with other concurrent computations, program fragments that perform allocation need not be encapsulated within a critical section. In fact, it is possible to create circular data structures in the monitored region without acquiring the monitor, using an EXTEND expression.

Finally, the type and effect inference rule for EXCLUSIVE expressions ensures that the first subexpression has type (MONITOR ρ) for some region constant or variable ρ, and that the second subexpression, which constitutes the body of the EXCLUSIVE expression, is well-formed and does not have any MCALL effects on the region ρ. When this is the case, the type of the expression is simply the type of the body, and the effect of the expression is obtained by combining the effect of the monitor expression, the effect of the body (after masking any effects on the region denoted by ρ), and the effect (MCALL ρ).

$$\frac{\begin{array}{cc} A,B \vdash e_1 : (\text{MONITOR } \rho) & A,B \vdash e_1 \,!\, \epsilon_1 \\ A,B \vdash e_2 : \tau & A,B \vdash e_2 \,!\, \epsilon_2 \\ \rho \in (Rconst \cup Dvar) & (\text{MCALL } \rho) \not\sqsubseteq \epsilon_2 \end{array}}{\begin{array}{l} A,B \vdash (\text{EXCLUSIVE } e_1\ e_2) \ : \ \tau \\ A,B \vdash (\text{EXCLUSIVE } e_1\ e_2) \ ! \ (\text{MAXEFF } \epsilon_1\ \epsilon_2[\psi/\rho]\ (\text{MCALL } \rho)) \end{array}}$$

Note that the restriction on the effect of the body prevents deadlock due to recursive invocation of the monitor. We will not make use of this property.

6.4 Dynamic Semantics

From the programmer's point of view, the introduction of explicit concurrency into the language does not change the semantics of programs and program fragments that do not make use of the new expressions. In particular, any program or program fragment that is free of COBEGIN expressions is evaluated in left-to-right, applicative order and is deterministic.

The subexpressions of a COBEGIN expression, however, are evaluated concurrently: their evaluations proceed independently and may be interleaved subject only to the constraints imposed by critical sections. The semantics of the EXCLUSIVE expression ensure that the evaluations of two critical sections for the same region are never interleaved.

The standard semantics of *MFX-3* does not guarantee fair scheduling; programs that rely on interleaving may not work as intended.

6.4.1 Monitors

Before we can describe the semantics formally, we must define what we mean by monitors. Conceptually, a monitor is an object with one bit of state that indicates whether or not some process currently has access to the corresponding monitored region. Since a monitor is a shared resource with state that can change, monitors resemble locations.

Formally, monitors are a countably infinite set of constants:

$Mon =$	$\{m_1, m_2, \ldots\}$	– monitors (m)
$Const =$	$\ldots$	– ordinary constants
	Mon	– monitors

There is an implicit one-to-one correspondence between monitors and region constants with the same subscript: we regard each monitor m_i as the monitor for the region constant r_i.

In order to represent the state of the monitor for each monitored region, we augment the range of the region map with the symbols IDLE and BUSY.

Definition (revised). A *region map* γ is a finite partial function with signature $Rconst \rightarrow \{\text{USED}, \text{IDLE}, \text{BUSY}\}$.

We have adopted the following convention regarding the use of the symbols USED, IDLE and BUSY:

- if $\gamma(r) = \text{USED}$, then r is an ordinary region;
- if $\gamma(r) = \text{IDLE}$, then r is a monitored region to which no process currently has exclusive access; and
- if $\gamma(r) = \text{BUSY}$, then r is a monitored region to which some process currently has exclusive access.

Although two symbols would be sufficient, we decided to use three distinct symbols so that it is always clear whether a given region is an ordinary region or a monitored region. We do not actually make use of this property.

Since monitors are constants, they are also expressions. This means that we must define their free ordinary and description variables, their free locations and region constants, their types, and their effects. The first few are easy: since monitors are constants, they have neither free ordinary variables nor free description variables. However, because of the correspondence between monitors and region constants, we must treat a monitor constant as if it refers to the corresponding region constant:

$$FRC_{exp}(m_i) = \{r_i\}$$

The type of a monitor is a MONITOR type whose region parameter is the corresponding region constant:

$$m_i : (\text{MONITOR } r_i)$$

Finally, all monitors have effect PURE because they are constants.

6.4.2 Auxiliary Expressions

The semantics of the MONITORED and EXCLUSIVE expressions are defined in terms of auxiliary expressions. The basic technique is the same as for PRIVATE and EXTEND: each expression is reduced to a corresponding auxiliary expression, the body of which can then be reduced recursively. In the case of MONITORED, a fresh region constant is chosen during this initial reduction step, and embedded in the auxiliary expression. In the case of EXCLUSIVE, the initial reduction step makes the corresponding monitor BUSY, and the final reduction step makes it IDLE. As always, the auxiliary expression serves to express the applicable effect masking rule while the body of the auxiliary expression is reduced recursively.

The auxiliary expressions for MONITORED and EXCLUSIVE are *MONITORED* and *EXCLUSIVE* respectively. Their syntax is given below.

Exp =	...	– expressions
	(*MONITORED* *Rconst Exp*)	– MONITORED in progress
	(*EXCLUSIVE* *Mon Exp*)	– EXCLUSIVE in progress

In the *MONITORED* expression, the region constant identifies the private, anonymous monitored region. In the *EXCLUSIVE* expression, the monitor constant identifies the monitor, and hence the region, to which the expression has exclusive access.

The *MONITORED* expression is an auxiliary binding expression: the region constant that represents the private region takes the place of the bound variable in the corresponding MONITORED expression. The *EXCLUSIVE* expression is not a binding expression, since an EXCLUSIVE expression does not bind any variables.

The type and effect inference rule for the *MONITORED* expression is given below. It is derived directly from the rule for the MONITORED expression. It requires that the body be well-formed, that the private region not be free in the type of the body, and that the body not have any READ or WRITE effect on the monitored region. The effects on the private region are masked.

$$\frac{\begin{array}{c} A, B \vdash e : \tau \\ A, B \vdash e \,!\, \epsilon \\ r \notin FRC_{desc}(\tau) \\ (\text{READ } r) \not\sqsubseteq \epsilon \;\wedge\; (\text{WRITE } r) \not\sqsubseteq \epsilon \end{array}}{\begin{array}{lll} A, B \vdash (\text{*MONITORED*}\ r\ e) & : & \tau \\ A, B \vdash (\text{*MONITORED*}\ r\ e) & ! & \epsilon[\psi/r] \end{array}}$$

The type and effect inference rule for the *EXCLUSIVE* expression is given below. This rule is derived directly from the rule for the EXCLUSIVE expression. It ensures that the body of the expression is well-formed and does not

have any MCALL effects on the region r_i.

$$\frac{A,B \vdash e:\tau \qquad A,B \vdash e\,!\,\epsilon \qquad (\text{MCALL } r_i) \not\sqsubseteq \epsilon}{\begin{array}{lll} A,B \vdash (\text{*EXCLUSIVE* } m_i\ e) & : & \tau \\ A,B \vdash (\text{*EXCLUSIVE* } m_i\ e) & ! & (\text{MAXEFF } \epsilon[\psi/r_i]\ (\text{MCALL } r_i)) \end{array}}$$

6.4.3 Reduction Axioms and Inference Rules

The reduction axiom for COBEGIN is very simple: when all the subexpression of a COBEGIN expression are values, the expression reduces to NIL.

$$\langle(\text{COBEGIN } v_1 \ldots v_n), \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle \text{NIL}, \sigma, \gamma\rangle$$

Subexpressions of a COBEGIN expression that are not values can be reduced recursively; the recursive reduction inference rules will be presented shortly.

The reduction axiom for MONITORED expressions, which is given below, is straightforward: the rule simply chooses an unused region constant r_i, marks it as IDLE, and replaces the expression by the corresponding auxiliary expression after substituting r_i for the bound description variable d and m_i for the bound ordinary variable x.

$$\langle(\text{MONITORED } d\ x\ e), \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle(\text{*MONITORED* } r_i\ e[r_i/d][m_i/x]), \sigma, \gamma[r_i \leftarrow \text{IDLE}]\rangle \quad (r_i \textit{ not bound in } \gamma)$$

When the body of a *MONITORED* expression has been reduced to a value (by means of the reduction inference rules), the *MONITORED* expression, which serves to mask the effects on the private region, is no longer needed and can be reduced to a value.

$$\langle(\text{*MONITORED* } r\ v), \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle v, \sigma, \gamma\rangle$$

Note that the private region constant r cannot be made unused when the expression returns, because references to this region may still exist (as was the case for the PRIVATE expression). However, if the state is well-formed, any remaining references to the region can never occur as the first argument to an active GET or SET expression, and any remaining instances of the monitor can never occur as the first argument to an active EXCLUSIVE expression.

The reduction axiom for EXCLUSIVE expressions can be read as follows: an EXCLUSIVE expression whose first subexpression has been reduced to a monitor represents an attempt to acquire exclusive access to the corresponding region. Such an expression can acquire the monitor whenever the region corresponding to the monitor is IDLE. Acquiring the monitor simply means

making the region BUSY and replacing the expression by the corresponding auxiliary expression.

$$\langle(\text{EXCLUSIVE } m_i\ e), \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle(\text{*EXCLUSIVE*}\ m_i\ e), \sigma, \gamma[r_i \leftarrow \text{BUSY}]\rangle \quad (\gamma(r_i) = \text{IDLE})$$

An *EXCLUSIVE* expression represents a critical section whose evaluation is in progress. When the body of the critical section has been reduced to a value, the monitor can be released and the value of the body returned.

$$\langle(\text{*EXCLUSIVE*}\ m_i\ v), \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle v, \sigma, \gamma[r_i \leftarrow \text{IDLE}]\rangle$$

The new reduction inference rules are presented below. The reduction inference rule for COBEGIN is different from the others: it is the *only* rule that permits a given expression to have more than one active subexpression.

$$\frac{\langle e_i, \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle e_i', \sigma', \gamma'\rangle}{\langle(\text{COBEGIN} \ldots e_i \ldots), \sigma, \gamma\rangle \stackrel{\text{red}}{\Longrightarrow} \langle(\text{COBEGIN} \ldots e_i' \ldots), \sigma', \gamma'\rangle}$$

The remaining rules are routine, and can be represented by the following contexts:

- MONITORED in progress: (*MONITORED* r [ ])
- critical section: (EXCLUSIVE [ ] e_2)
- EXCLUSIVE in progress: (*EXCLUSIVE* m [ ])

One technicality remains to be resolved: due to the introduction of explicit concurrency, a state may have more than one active redex, which means that the context of an active expression can change while the expression is being reduced. This introduces a technical problem in the reduction axiom for the *EXTEND* expression given in Chapter 5: the scope of the substitution of the target region for the private region constant must be widened to encompass the entire expression component of the state, rather than just the body of the *EXTEND* expression, in addition to the store. This can be done by merging the definitions of contexts and active expressions [Fel87, p. 317]; we omit the details.

6.4.4 Properties of the Standard Semantics

Since an active COBEGIN expression may have more than one active subexpression, states can now in general be reduced in more than one way. Accordingly, we find that the active expressions of a state no longer form a chain but a *tree*: the root of the tree is the expression itself, and the active (immediate) subexpressions of each active expression are its children in the tree. Every leaf of the tree is an active redex. For example, the tree of active expressions of the expression

```
(COBEGIN
  (EXCLUSIVE m
    (SET l TRUE))
  (EXCLUSIVE m
    (SET l FALSE)))
```

has two branches, one for each subexpression of the COBEGIN expression.

Although the individual reduction axioms are still deterministic, except for the choice of free locations, reduction is no longer deterministic overall, and the value returned by an expression is no longer unique up to the choice of locations. Formally, the meaning of an expression, $M[\![e]\!]$, no longer maps equivalent stores to equivalent terminal states. However, it is still the case that the semantics of the language are independent of the storage allocation policy: the meaning of an expression, $M[\![e]\!]$, corresponds to a *one-to-many* mapping from the equivalence classes of stores σ modulo the permutations that fix $FL_{exp}(e)$ to the equivalence classes of terminal states modulo the permutations that fix $FL_{sta}(\langle e, \sigma \rangle)$.

6.4.5 Suspended Expressions

Definition. An EXCLUSIVE expression in a state $\langle e, \sigma, \gamma \rangle$ is *suspended* if it cannot be reduced because it is waiting to acquire a monitor that is BUSY, *i.e.* if it is of the form (EXCLUSIVE m_i e) for some i such that $\gamma(r_i) =$ BUSY.

More generally, an active expression e in a state $\langle C[e], \sigma, \gamma \rangle$ is suspended iff all its active subexpressions are suspended, *i.e.* if every active redex in e is of the form (EXCLUSIVE m_i e') for some i (not necessarily the same for each expression) such that $\gamma(r_i) =$ BUSY.

A suspended expression is not stuck, since it becomes reducible when the corresponding monitor becomes IDLE. However, since suspended expressions have no control over when their suspension ends, a variety of deadlocks can result.

- If an expression acquires the monitor m and then calls a subroutine that attempts to acquire m recursively, then a deadlock results. In a well-formed expression, deadlock due to recursive monitor invocation cannot occur.
- A more intricate example is the so-called *deadly embrace* [Lam80]. If one expression attempts to acquire the monitors m_1 and m_2, in that order, while another expression attempts to acquire the same monitors in the opposite order, then a deadly embrace results whenever m_1 is granted to the former expression and m_2 to the latter. There is a standard technique for avoiding deadly embrace, namely to insist that the set of resources that can be accessed by any given process must be totally ordered, and must always be acquired in a sequence consistent with this ordering [Bri75]. Unfortunately, this technique cannot be employed in a language with first-class monitors unless the declarations of monitors and monitored regions

are augmented with information about this ordering, which would require some form of constrained quantification over regions. We have not pursued this option.

- Finally, there are a variety of so-called dynamic deadlocks, in which one process is forever prevented from making progress by one or more other processes that share some of the same resources. A detailed investigation of dynamic deadlocks is beyond the scope of this thesis.

As long as a state contains *any* active expression that is not suspended, there is some way to reduce the state unless it is stuck. This justifies our earlier claim that as long as there is at least one active expression that can make progress, some active expression is guaranteed to make progress.

6.5 Type Soundness

Because of the introduction of concurrency, the proof of type soundness no longer holds as originally formulated: in particular, the effect propagation lemma must be extended with two new cases. In what follows, we refine the definitions and propositions that are affected by the introduction of concurrency.

We begin be refining the notion of a *legal* state.

Definition. A state $\theta = \langle e, \sigma, \gamma \rangle$ is *legal*, $\mathcal{L}(\theta)$, iff

1. (as before) every auxiliary expression in θ is active;
2. (as before) no two auxiliary binding expressions in θ have the same private region constant;
3. (as before) the private region constant of an auxiliary binding expression is inaccessible in the context that surrounds the expression;
4. (new) no two *EXCLUSIVE* expressions in θ have the same monitor; and
5. (new) a monitor m_i is BUSY, $\gamma(r_i) =$ BUSY, iff θ contains an *EXCLUSIVE* expression of the form (*EXCLUSIVE* m_i e').

Lemma (revised). (Effect Propagation) In a well-formed state, the effect of each active expression is a subeffect of the effect of its parent expression, unless

1. (as before) the parent is an auxiliary binding expression, in which case the effects on its private region constant are masked; or
2. (new) the parent is an *EXCLUSIVE* expression, in which case the effects on its monitored region are masked and replaced by an MCALL effect on that region.

Proof. The proof proceeds as before, using the correspondence between the new reduction inference rules and the new effect inference rules. In particular, the effect of a *MONITORED* expression is obtained from the effect of its body by masking the effects on its private region constant, and the effect of an *EXCLUSIVE* expression is obtained from the effect of its body

by replacing the effects on its monitored region by an MCALL effect on that region. □

As before, using the fact that the new effect inference rules parallel the structure of the new reduction inference rules. □

Proposition (revised). (Type and Effect Preservation) Reduction of a well-formed state preserves or decreases the type and effect descriptions of the state.

Proof. The proof proceeds as before, but there are some new cases to be considered.

It is easy to see that reduction still preserves consistency: new region constants are introduced only by the reduction axioms for PRIVATE, EXTEND and MONITORED, each of which binds the new region constant in the region map. Since a region constant bound in the region map never becomes unbound (although the state of a monitored region may alternate between IDLE and BUSY), reduction preserves consistency.

To show that reduction preserves legality, we consider each property of legal states in turn. The first three properties are the same as before, so we consider only the two new properties.

- The *fourth property*, mutual exclusion, is preserved because (i) an *EXCLUSIVE* expression can be created only when the region in question is IDLE, and this makes the region BUSY until the expression is reduced to a value; and (ii) every *EXCLUSIVE* expression is active, and therefore cannot be duplicated.
- The *fifth property*, monitor consistency, is preserved because (i) when a monitored region constant is created, it is IDLE and there are no *EXCLUSIVE* expressions for it; and (ii) a region can be made BUSY only by creating a corresponding *EXCLUSIVE* expression, and it can be made IDLE only by reducing that *EXCLUSIVE* expression to a value. □

6.6 Effect Soundness

We are left with the task of extending our effect soundness proposition, which has been invalidated by the introduction of MCALL effects. For example, if a state θ contains an active expression of the form (*EXCLUSIVE* m_i e) with effect ϵ, then a reduction of that active expression may access locations in the region r_i even though ϵ does not incorporate any ALLOC, READ or WRITE effects on r_i. This is reflected in the following revised effect soundness proposition:

Proposition (revised). (Effect Soundness) Reduction of an expression in a well-formed state allocates, reads, and writes only locations that can be reached through the regions specified by its effect and/or through regions that are accessible only within the expression. In other words, if $\theta = \langle C[e], \sigma, \gamma \rangle$, $\theta' = \langle C[e'], \sigma', \gamma' \rangle$ and $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$, and $e \,!\, \epsilon$ where

$$\epsilon \simeq (\text{MAXEFF } (\text{ALLOC } \rho_A)\ (\text{READ } \rho_R)\ (\text{WRITE } \rho_W)\ (\text{MCALL } \rho_M))$$

then

$$\begin{aligned} \mathcal{A}(\theta,\theta') \cap \mathit{Reach}(\mathit{Acc}(C)) &\subseteq \mathit{Reach}(\rho_A) \cup \mathit{Reach}(\rho_M) \\ \mathcal{R}(\theta,\theta') \cap \mathit{Reach}(\mathit{Acc}(C)) &\subseteq \mathit{Reach}(\rho_R) \cup \mathit{Reach}(\rho_M) \\ \mathcal{W}(\theta,\theta') \cap \mathit{Reach}(\mathit{Acc}(C)) &\subseteq \mathit{Reach}(\rho_W) \cup \mathit{Reach}(\rho_M) \end{aligned}$$

Proof. As before, using the revised effect propagation lemma. □

This proposition generalizes immediately to $\theta \stackrel{\text{red}}{\Longrightarrow}{}^* \theta'$.

Although this proposition is weaker than the previous effect soundness proposition, its power to identify opportunities for concurrent evaluation and memoization has not diminished because the locations that can be reached through a monitored region can be read and/or written only *in exclusive mode.*

Definition. A location $l_{\rho,\tau}$ read and/or written in the reduction step $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$ is read and/or written *in exclusive mode* iff the active redex reduced in the reduction step is embedded in a context of the form (*EXCLUSIVE* m_i [ ]) for each monitored region constant $r_i \in \mathit{FRC}_{desc}(\rho)$.

Proposition. (Monitor Encapsulation) In a well-formed state, a location that can be reached through a monitored region is read and/or written only in exclusive mode.

Proof. By the effect propagation lemma, and the fact that the body of a *MONITORED* expression has no READ or WRITE effects, a READ or WRITE effect on a monitored region can be masked only by an *EXCLUSIVE* expression. It follows that in every active redex that reads or writes a location that can be reached through a monitored region is embedded in an *EXCLUSIVE* expression for the corresponding region. □

Therefore, although an expression that has an MCALL effect on a monitored region may allocate, read, and write locations that are accessible through

that region, it must acquire exclusive access to the region in question in order to read or write.

In practical terms, the revised effect soundness and monitor encapsulation propositions show that is possible to state and enforce a set of language restrictions that ensure that all interference between concurrent tasks is mediated by monitors and critical sections. This makes it possible to integrate explicit concurrency into a programming language designed to promote implicit concurrency while preserving all existing opportunities for concurrent evaluation and memoization.

Chapter 7. Language Extensions

The *MFX* language is far from being a complete programming language. In this chapter we present several language extensions that make the language more practical, such as parameterized and recursive types, and we discuss their interaction with the type and effect system. The purpose of this chapter is to demonstrate the general applicability of the type and effect system.

We conclude the chapter with a discussion of polymorphic abstraction and its performance implications, and we propose an alternative definition of polymorphism that permits an implementation in which polymorphic application has no run-time cost.

7.1 Trivial Extensions

In this section we introduce syntactic sugar for subroutines of multiple arguments, for local variables, and for local description synonyms, and we show how to define pre-declared descriptions and constants, such as the type of integers and the usual integer literals and operators.

7.1.1 Subroutines of Multiple Arguments

Thus far, all the constructs for abstraction and application bind only one parameter at a time. As a result, subroutines of multiple arguments must be curried, and there is no way to define subroutines that take no arguments. We now remove this restriction: from now on, all constructs for abstraction and application, including the derived constructs, will accept any number of parameters. To improve readability, all parameter declarations should be enclosed in parentheses. All the variables bound by an expression must be distinct.

This generalization is fairly routine, but we will illustrate it by means of a few examples. For these examples we will assume that `int` is the type of integers, that the literals `0`, `1` and `2` represent the corresponding integers, and that `+` denotes a subroutine that takes two integers and returns their sum.

The following expression takes three integers and returns their sum:

```
(LAMBDA (x:int y:int z:int)
  (+ (+ x y) z))
```

This expression has the following type:

```
(SUBR (int int int) PURE
  int)
```

The following expression emulates application for subroutines of a single argument:

```
(PLAMBDA (t1:TYPE e:EFFECT t2:TYPE)
  (LAMBDA (x:t1 f:(SUBR (t1) e t2))
    (f x)))
```

This expression has the following type:

```
(POLY (t1:TYPE e:EFFECT t2:TYPE) PURE
  (SUBR (t1 (SUBR (t1) e t2)) e
    t2))
```

The necessary changes to the grammar, description inclusion rules, type and effect inference rules, and semantics are straightforward, and will be omitted.

7.1.2 Local Variables

MFX can easily be extended with the usual LET construct, the semantics of which can be expressed as a source-to-source transformation that generates an application of an ordinary subroutine. The syntax of the LET construct is given by the following grammar clause:

Exp =	...	– expressions
	(LET ((*Var Exp*)*) *Exp*)	– let-expression

The expression (LET $((x_1\ e_1)\ldots(x_n\ e_n))\ e$) has the following semantics. First, the e_i are evaluated, in order. Next, the resulting values v_i are substituted for the corresponding variables x_i throughout the body e. Finally, the body, thus modified, is evaluated and its value returned. In other words, this expression is equivalent to the expression

$$((\text{LAMBDA}\ (x_1{:}\tau_1 \ldots x_n{:}\tau_n)\ e)\ e_1 \ldots e_n)$$

where the τ_i are the types of the expressions e_i.

Because of the possibility of side-effects, this expression is *not* equivalent to the expression obtained by simultaneously substituting the expressions e_i for the corresponding variables x_i throughout the body e, *i.e.*

$$e\ [e_1/x_1 \ldots e_n/x_n]$$

For example, the expression

```
(LET ((x (NEW r1 BOOL TRUE)))
  (BEGIN
    (SET x FALSE))
    (GET x))
```

is equivalent to the expression

```
((LAMBDA (x:(REF r1 BOOL))
   (BEGIN
     (SET x FALSE))
     (GET x))
 (NEW r1 BOOL TRUE))
```

and *not* to the expression

```
(BEGIN
  (SET (NEW r1 BOOL TRUE) FALSE)
  (GET (NEW r1 BOOL TRUE)))
```

The type and effect inference rule for LET expressions is given below. This rule, which is a composition of the rules for ordinary abstraction and application generalized to n arguments, can be read as follows: provided that the e_i are all well-formed, the type of the expression as a whole is equal to the type of the body in a type assignment in which each x_i has the type of the corresponding e_i. The effect of the expression is the least upper bound of the effects of the e_i and the effect of the body.

$$\frac{\begin{array}{c}\forall i, 1 \leq i \leq n \;.\; A, B \vdash e_i : \tau_i \\ \forall i, 1 \leq i \leq n \;.\; A, B \vdash e_i \,!\, \epsilon_i \\ A[x_1 \leftarrow \tau_1] \ldots [x_n \leftarrow \tau_n], B \vdash e : \tau \\ A[x_1 \leftarrow \tau_1] \ldots [x_n \leftarrow \tau_n], B \vdash e \,!\, \epsilon\end{array}}{\begin{array}{lll} A, B \vdash (\text{LET } ((x_1\ e_1) \ldots (x_n\ e_n))\ e) & : & \tau \\ A, B \vdash (\text{LET } ((x_1\ e_1) \ldots (x_n\ e_n))\ e) & ! & (\text{MAXEFF } \epsilon_1 \ldots \epsilon_n\ \epsilon)\end{array}}$$

7.1.3 Description Synonyms

MFX can easily be extended with a construct for defining transparent description synonyms. The semantics of this construct can be expressed in terms of a source-to-source transformation. The syntax of the DLET construct (for Description LET) is given by the following grammar clause:

Exp =	...	– expressions
	(DLET ((*Dvar Desc*)*) *Exp*)	– dlet-expression

The expression (DLET $((d_1\ \delta_1) \ldots (d_n\ \delta_n))\ e$) has the following semantics: the descriptions δ_i are substituted for the corresponding description variables d_i throughout the body e, and the body, thus modified, is evaluated and its value returned. In other words, this expression is equivalent to the expression

$$e\ [\delta_1/d_1 \ldots \delta_n/d_n]$$

Because of the static type checking algorithm, this is *not* equivalent to the expression obtained by abstracting the body over the description variables $d_1 \ldots d_n$ and applying the result to the corresponding description, *i.e.*

$$(\text{PROJ } (\text{PLAMBDA } (d_1{:}\kappa_1 \ldots d_n{:}\kappa_n)\ e)\ \delta_1 \ldots \delta_n)$$

For example, the expression

```
(DLET ((apples int)
       (oranges int))
  ((LAMBDA (n:apples m:oranges)
     (+ n m))
   2 3))
```

is equivalent to the expression

```
((LAMBDA (n:int m:int)
   (+ n m))
 2 3)
```

and *not* to the expression

```
(PROJ (PLAMBDA (apples:TYPE oranges:TYPE)
        ((LAMBDA (n:apples m:oranges)
           (+ n m))
         2 3))
      int int)
```

In fact, the latter expression is ill-typed in two respects:

- the application `(+ n m)` is ill-typed since `+` expects two values of type `int` but `n` and `m` have type `apples` and `oranges` respectively;
- the application of the LAMBDA expression is ill-typed since it expects values of type `apples` and `oranges` respectively but `2` and `3` both have type `int`.

The type and effect inference rule for DLET expressions is given below. It can be read as follows: provided that the δ_i are all well-formed, the type and effect of the expression are equal to the type and effect of the body after substituting the descriptions δ_i for the corresponding description variables d_i.

$$\frac{\begin{array}{c}\forall i, 1 \leq i \leq n \;.\; B \vdash \delta_i : \kappa_i \\ A, B \vdash e[\delta_1/d_1 \ldots \delta_n/d_n] : \tau \\ A, B \vdash e[\delta_1/d_1 \ldots \delta_n/d_n] \,!\, \epsilon\end{array}}{\begin{array}{lll} A, B \vdash (\text{DLET } ((d_1\ \delta_1) \ldots (d_n\ \delta_n))\ e) & : & \tau \\ A, B \vdash (\text{DLET } ((d_1\ \delta_1) \ldots (d_n\ \delta_n))\ e) & ! & \epsilon \end{array}}$$

7.1.4 Built-in Types

MFX can easily be extended with a variety of pre-declared variables, denoting types, effects, regions, subroutines, literals and so forth. The existing abstraction constructs provide a formalism for describing such language extensions. We illustrate this below by showing how to extend *MFX* with a pre-declared type variable `int` and a set of pre-declared variables corresponding to the usual integer literals and operators.

The existing abstraction constructs are not powerful enough to extend the language with additional type, effect or region *constructors* (such as `array` and `list`). Constructs for introducing user-defined type constructors are presented in the next section.

In order to extend the language with a type `int` and a suitable set of integer literals and operators, it suffices to abstract every program over the type variable `int` and the ordinary variables `+`, `-`, and so forth:

```
(PLAMBDA (int:TYPE)
  (LAMBDA (+:(SUBR (int int) PURE int)
           -:(SUBR (int int) PURE int)
           *:(SUBR (int int) PURE int)
           /:(SUBR (int int) PURE int)
           =:(SUBR (int int) PURE BOOL)
           <:(SUBR (int int) PURE BOOL)
           0:int 1:int -1:int 2:int -2:int ...)
    e))
```

From the point of view of the programmer, the type `int` is an abstract type with four binary operators, two binary predicates, and a countably infinite number of literals. Because of type abstraction, there is no need to specify the *actual* type and the *actual* values to which the program is applied: it suffices to give a description (algebraic, axiomatic or otherwise) of their semantics. Since any reasonable definition will suffice, we will not belabor this point.

7.2 Higher-order Descriptions

MFX is based on the higher-order lambda-calculus of McCracken [McC82], which supports type functions and recursive types. We have omitted these features from *MFX* in order to simplify the presentation. However, *MFX* can easily be extended with both.

In this section we show how to extend *MFX* with higher-order descriptions.

7.2.1 Syntax

In order to express the kinds of higher-order descriptions, we need to extend the grammar of kinds accordingly. The new grammar of kinds is given below. A kind description is one of the following: a kind constant (one of REGION, EFFECT and TYPE), or a higher-order kind. There are no kind variables, since there is no construct for abstraction over kinds.

Kind =		– kinds (κ)
	REGION	– kind of regions
	EFFECT	– kind of effects
	TYPE	– kind of types
	(DFUNC (*Kind*) *Kind*)	– kinds of description functions

The kind (DFUNC (κ_1) κ_2) is a generalization of the kind "$\kappa_1 \Rightarrow \kappa_2$" in the higher-order lambda-calculus of McCracken [McC82], *i.e.* it is the kind of description functions that map descriptions of kind κ_1 to descriptions of kind κ_2.

The grammar of higher-order descriptions in general is given below. A higher-order description is one of the following: a description variable, a description function, or a description application.

HDesc =		– higher-order descriptions
	Dvar	– description variables
	(DLAMBDA (*Dvar*:*Kind*) *Desc*)	– description functions
	(*HDesc Desc*)	– description applications

The description function (DLAMBDA (d:κ) δ) corresponds to the type "$\lambda d{:}\kappa.\delta$" in the higher-order lambda-calculus of McCracken [McC82].

The updated grammar of descriptions in general is given below. A description is now one of the following: a region description, an effect description, a type description, or a higher-order description.

Desc =		– descriptions (δ)
	Region	– region descriptions
	Effect	– effect descriptions
	Type	– type descriptions
	HDesc	– higher-order descriptions

A description application, like a description variable, can in principle have any kind, including REGION, EFFECT or TYPE. The grammars of regions, effects and types must be updated accordingly. We omit the details.

In what follows, we assume that the constructs for description abstraction and application has been generalized to accept any number of parameters, using the same syntactic conventions as before.

7.2.2 Informal Semantics

The description function (DLAMBDA (d:κ) δ) acts as a function that, when applied to a description of kind κ, yields a description of the kind of δ. This is useful particularly in connection with a construct, such as DLET, for defining transparent description synonyms.

For example, the description function

```
(DLAMBDA (r:REGION)
  (MAXEFF (ALLOC r) (READ r) (WRITE r)))
```

maps any region ρ to the effect (MAXEFF (ALLOC ρ) (READ ρ) (WRITE ρ)). Since this description function takes a region as argument and returns an effect, it has kind

```
(DFUNC (REGION) EFFECT)
```

Descriptions may have free variables, provided that they are defined in the surrounding scope. For example, if **sum** is declared in the surrounding scope as a higher-order description of kind (DFUNC (TYPE TYPE) TYPE) and **prod** is declared as a higher-order description of kind (DFUNC (REGION TYPE TYPE) TYPE), then the description

```
(DLAMBDA (r:REGION t:TYPE)
  (sum UNIT (prod r t t)))
```

is well-formed and has kind

```
(DFUNC (REGION TYPE) TYPE)
```

Like a manifest description function, a higher-order description variable acts as a function that, when applied to a description of the appropriate kind, yields a description of some other kind — in other words, it acts as a *parameterized* region, effect, type, or other description.

For example, a higher-order description variable of kind

```
(DFUNC (TYPE) TYPE)
```

acts as a parameterized type. If the description variable **vector** has this kind, then the description applications (**vector** BOOL) and (**vector int**) are well-formed and have kind TYPE. If vectors are implemented as subroutines that map integers to vector elements, then the parameterized type **vector** could correspond to the description function (or parameterized representation type)

```
(DLAMBDA (t:TYPE)
  (SUBR (int) PURE t))
```

7.2.3 Description Conversion

The higher-order descriptions call for the addition of rules for alpha, beta and eta-conversion.

$$(\text{DLAMBDA } (d{:}\kappa)\ \delta) \simeq (\text{DLAMBDA } (d'{:}\kappa)\ \delta[d'/d]) \quad (d' \notin FDV_{desc}(\delta))$$
$$((\text{DLAMBDA } (d{:}\kappa)\ \delta_1)\ \delta_2) \simeq \delta_1[\delta_2/d]$$
$$(\text{DLAMBDA } (d{:}\kappa)\ (\delta\ d)) \simeq \delta \quad (d \notin FDV_{desc}(\delta))$$

Two description applications are convertible whenever their respective operators and operands are convertible:

$$\frac{\delta_1 \simeq \delta_1' \quad \delta_2 \simeq \delta_2'}{(\delta_1\ \delta_2) \simeq (\delta_1'\ \delta_2')}$$

Description conversion in the presence of parameterized regions and effects presents some interesting anomalies, due to the fact that the region and effect constructors are all monotonic and distributive. We will return to these anomalies at the end of this section.

7.2.4 Static Semantics

The kind inference rules for description abstraction and application are taken directly from the higher-order lambda-calculus. Note in particular that the kind of the actual parameter of a description application must match the kind of the formal parameter exactly: there is no "subkinding".

$$\frac{B[d \leftarrow \kappa] \vdash \delta : \kappa'}{B \vdash (\text{DLAMBDA } (d{:}\kappa)\ \delta) : (\text{DFUNC } (\kappa)\ \kappa')}$$

$$\frac{B \vdash \delta_1 : (\text{DFUNC } (\kappa_1)\ \kappa_2) \quad B \vdash \delta_2 : \kappa_1}{B \vdash (\delta_1\ \delta_2) : \kappa_2}$$

Our notion of description abstraction is more general than that of the higher-order lambda-calculus, since we have more than one base kind. However, this generalization is not visible in the inference rules.

The type and effect inference rules are not affected by the introduction of higher-order descriptions: polymorphic abstraction over higher-order descriptions is no different from polymorphic abstraction over descriptions of base kind.

7.2.5 Parameterized Types

Using abstraction over higher-order descriptions, it is possible to define a variety of mutable parameterized types, such as `array` and `list`, that are parameterized over the region to which the corresponding locations belong. For example, given an immutable parameterized type `vector`, of kind (DFUNC (TYPE) TYPE), with operators

```
vector-new:(POLY (t:TYPE) PURE
             (SUBR (int (subr (int) PURE t)) PURE
               (vector t)))
vector-get:(POLY (t:TYPE) PURE
             (SUBR ((vector t) int) PURE
               t))
```

it is possible to implement a mutable parameterized type `array`, of kind (DFUNC (REGION TYPE) TYPE), with operators

```
array-new:(POLY (r:REGION t:TYPE) PURE
            (SUBR (int t) (ALLOC r)
              (array r t)))
array-get:(POLY (r:REGION t:TYPE) PURE
            (SUBR ((array r t) int) (READ r)
              t))
array-set:(POLY (r:REGION t:TYPE) PURE
            (SUBR ((array r t) int t) (WRITE r)
              UNIT))
```

where the parameterized type `array` could correspond, for example, to the description function (or parameterized representation type)

```
(DLAMBDA (r:REGION t:TYPE)
  (vector (REF r t)))
```

Parameterized types such as `array` meet the objectives set forth in Chapter 2, which we reproduce below with a slight change to reflect the use of regions:

- the programmer can easily instantiate a given type in several different regions, to indicate how the values of the resulting types are used; and
- the method applies equally well to built-in and programmer-defined types.

7.2.6 The Side-effect Operators Revisited

Using abstraction over higher-order descriptions, it is possible to replace the type constructor REF and the operators NEW, GET and SET by a parameterized type **ref** and polymorphic subroutines **new**, **get** and **set**. To make these variables built-in, it suffices to abstract every program *e* over the description variable **ref** and the ordinary variables **new**, **get** and **set**:

```
(PLAMBDA (ref:(DFUNC (REGION TYPE) TYPE))
  (LAMBDA (new:(POLY (r:REGION t:TYPE) PURE
                     (SUBR (t) (ALLOC r)
                       (ref r t)))
            get:(POLY (r:REGION t:TYPE) PURE
                      (SUBR ((ref r t)) (READ r)
                        t))
            set:(POLY (r:REGION t:TYPE) PURE
                      (SUBR ((ref r t) t) (WRITE r)
                        UNIT)))
    e))
```

From the point of view of the programmer, **ref** is a parameterized type with three operators and no literals. Because of type abstraction, there is no need to specify the *actual* description function and the *actual* values to which the program is applied: it suffices to give a description (algebraic, axiomatic or otherwise) of their semantics.

The combination of the parameterized type **ref** and the polymorphic subroutines **new**, **get** and **set** is *almost* as powerful as the built-in type constructor REF and the built-in constructs NEW, GET and SET. However, it falls short in two respects: *implicit polymorphism* and *tonicity*.

The first difference, implicit polymorphism, lies in the fact that the GET and SET constructs are *implicitly* polymorphic: they can operate directly on locations of any type, without a need for the programmer to supply region or type information. The polymorphic subroutines **get** and **set**, on the other hand, must be applied to appropriate descriptions before they can be used.

Nevertheless, instances of NEW, GET and SET expressions can always be rewritten in terms of **new**, **get** and **set**, and vice versa. This is illustrated below, where **r** and **t** are the region and type components, respectively, of the type of **location**.

```
(NEW r t value)       ↔  ((PROJ new r t) value)
(GET location)        ↔  ((PROJ get r t) location)
(SET location value)  ↔  ((PROJ set r t) location value)
```

This rewriting could be done automatically by the compiler, with the aid of some relatively simple type inference. Therefore, the polymorphic subroutines **new**, **get** and **set** are essentially equivalent to the built-in constructs NEW, GET and SET.

The second difference lies in the fact that REF is a type constructor: according to the type inclusion rules, REF is monotonic in its first argument. The parameterized type `ref`, on the other hand, is not monotonic in any of its arguments. In other words, (REF ρ τ) is a subtype of (REF ρ' τ) whenever $\rho \sqsubseteq \rho'$, but(`ref` ρ τ) is a subtype of (`ref` ρ' τ) only if $\rho \simeq \rho'$.

Because of this difference, certain well-formed expressions are no longer well-formed when `ref` is substituted for REF. An example of such an expression appears below.

```
(LAMBDA (ref1:(REF r1 int)
         ref2:(REF r2 int))
  (IF p ref1 ref2))
```

this expression has type

```
(SUBR ((REF r1 int) (REF r2 int)) PURE
  (REF (UNION r1 r2) int))
```

where the region description (UNION `r1 r2`) reflects the uncertainty about the region to which the location in question actually belongs.

When `ref` is substituted for REF, this expression is not well-formed because no type is a supertype of both (`ref r1 int`) and (`ref r2 int`).

7.2.7 Tonicity

The difference between REF and `ref` comes down to the fact that in *MFX* there is no way to represent the *tonicity* of a higher-order description variable, *e.g.* the monotonicity or anti-monotonicity of the description variable with respect to a given argument. It would be possible to augment the kinds of description functions with tonicity information, just as we have augmented the types of subroutines with latent effect information. Such a tonicity specification would influence the convertibility of applications of the description function, just as the latent effect specification of a subroutine influences the effect specifications of applications of that subroutine.

Our research on this subject has suggested a four-fold classification: a description function δ_1 can be

- non-monotonic, *i.e.* $(\delta_1\ \delta_2) \simeq (\delta_1\ \delta_2')$ iff $\delta_2 \simeq \delta_2'$, for example

```
(DLAMBDA (t:TYPE)
  (SUBR (t) PURE t))
```

- monotonic, *i.e.* $(\delta_1\ \delta_2) \sqsubseteq (\delta_1\ \delta_2')$ iff $\delta_2 \sqsubseteq \delta_2'$, for example

```
(DLAMBDA (t:TYPE)
  (SUBR (int) PURE t))
```

- anti-monotonic, *i.e.* $(\delta_1\ \delta_2) \sqsubseteq (\delta_1\ \delta_2')$ iff $\delta_2 \sqsupseteq \delta_2'$, for example

```
(DLAMBDA (t:TYPE)
  (SUBR (t) PURE int))
```

- constant, *i.e.* $(\delta_1\ \delta_2) \simeq (\delta_1\ \delta_2')$ for all δ_2 and δ_2', for example

```
(DLAMBDA (t:TYPE)
  (SUBR (int) PURE int))
```

The last case may seem trivial, or even absurd; however, it logically complements the other three cases, as the examples clearly show.

The type constructor `ref`, which maps a region and a type to another type, is monotonic in its first argument and non-monotonic in its second argument.

We have not found any prior research on the subject of tonicity specifications. However, since our primary focus is on types and effects rather than type inclusion, a detailed investigation of higher-order kinds with tonicity specifications is beyond the scope of this thesis.

7.2.8 Parameterized Effects and Regions

We now turn to the anomalies presented by description conversion in the presence of parameterized effects and regions. The problem lies in ensuring the completeness of the description conversion and inclusion rules.

Since the region and effect constructors are all monotonic and distributive, every description function that returns a region or effect is also monotonic and distributive. As a result, the conversion and inclusion rules for applications of parameterized effects and regions would be *incomplete* unless they take these properties into account.

For example, *any* description function of kind

```
(DFUNC (REGION EFFECT) EFFECT)
```

is monotonic and distributive, *i.e.* if δ has the above kind, then

- $(\delta\ \rho\ \epsilon) \sqsubseteq (\delta\ \rho'\ \epsilon')$ whenever $\rho \sqsubseteq \rho'$ and $\epsilon \sqsubseteq \epsilon'$,
- $(\delta$ (UNION $\rho_1\ \rho_2)\ \epsilon)$ is convertible to (MAXEFF $(\delta\ \rho_1\ \epsilon)\ (\delta\ \rho_2\ \epsilon)$), and
- $(\delta\ \rho$ (MAXEFF $\epsilon_1\ \epsilon_2))$ is convertible to (MAXEFF $(\delta\ \rho\ \epsilon_1)\ (\delta\ \rho\ \epsilon_2)$).

Moreover, since there are no region constructors that take effects or types as arguments, and no effect constructors that take types as arguments, every region function is necessarily constant with respect to all its type and effect arguments, and every effect function is necessarily constant with respect to all its type arguments. For example, any description function of kind

```
(DFUNC (TYPE EFFECT) EFFECT)
```

is constant with respect to its first argument, *i.e.* if δ has the above kind, then $(\delta\ \tau\ \epsilon) \simeq (\delta\ \tau'\ \epsilon)$ for all τ and τ'.

These complications can be avoided by a language restriction that limits polymorphic abstraction to types, effects, regions and parameterized types. This restriction ensures that effect and region functions are always manifest, so that they can be eliminated by beta-reduction before the question

of convertibility arises. Since we have been unable to find any use for parameterized regions or effects, we believe that this restriction is a reasonable compromise.

7.3 Recursion

In this section we show how to extend *MFX* with recursive types and expressions (but not with recursive effects, regions, or kinds). The higher-order lambda-calculus of McCracken, which has served as a basis for much of our type system, supports recursive types; however, it requires explicit coercion between a description and its recursive instances [McC83, p. 6]. We show that such coercions are unnecessary in *MFX*.

Note that recursion is not needed to write non-terminating expressions in *MFX*: this is already possible through the use of side-effects. For example, it is possible to emulate Landin's LETREC construct by means of effects [Ree86]. However, this technique seems rather inappropriate in a language in which the effects of an expression are part of its specification.

7.3.1 Recursive Descriptions

In the description domain, we introduce a new construct, DLETREC (which stands for Description LETREC), for defining mutually recursive type descriptions. The syntax of this new construct is given below.

Desc =	...	– descriptions
	(DLETREC ((*Dvar Type*)*) *Desc*)	– recursion introduction

The description (DLETREC $((d_1\ \tau_1)\ldots(d_n\ \tau_n))\ \delta$) is convertible with the description

$$\delta[\tau_1'/d_1 \ldots \tau_n'/d_n]$$

where

$$\tau_i' = (\text{DLETREC}\ ((d_1\ \tau_1)\ldots(d_n\ \tau_n))\ d_i)$$

When the above conversion axiom is applied repeatedly to a DLETREC description, the description may expand without limit:

```
(DLETREC ((list (sum UNIT list'))
          (list' (prod BOOL list)))
  list)
≃
(sum UNIT
  (DLETREC ((list (sum UNIT list'))
            (list' (prod BOOL list)))
    list'))
≃
(sum UNIT
  (prod BOOL
```

```
    (DLETREC ((list (sum UNIT list'))
              (list' (prod BOOL list)))
      list)))
≃
(sum UNIT
  (prod BOOL
    (sum UNIT
      (DLETREC ((list (sum UNIT list'))
                (list' (prod BOOL list)))
        list'))))
```

and so on. If either the body or one of the definitions contains none of the bound variables, the conversion process may terminate after a finite number of steps:

```
(DLETREC ((list (sum UNIT list'))
          (list' BOOL))
  list)
≃
(sum UNIT
  (DLETREC ((list (sum UNIT list'))
            (list' BOOL))
    list'))
≃
(sum UNIT BOOL)
```

If one of the bound variables is defined as itself, either directly or indirectly, successive conversions yield a repeating, rather than an expanding, series of descriptions:

```
(DLETREC ((t1 t2)(t2 t1))
  t1)
≃
(DLETREC ((t1 t2)(t2 t1))
  t2)
≃
(DLETREC ((t1 t2)(t2 t1))
  t1)
```

and so on.

In general, applying the above conversion rule repeatedly yields a description, whether finite or infinite, that is free of DLETREC descriptions except for DLETREC descriptions whose body is a variable that is defined as itself.

Since all recursively defined descriptions are of kind TYPE, which is a base kind, the recursion equations of a DLETREC description form a *regular system* [Cou83], and all descriptions, whether finite or infinite, correspond to *regular trees*. Equality of regular trees is equivalent to equality of deterministic

finite automata, which is efficiently decidable [Cou83]. It follows that type conversion and inclusion are efficiently decidable even in the presence of the infinite descriptions generated by the DLETREC construct.

A DLETREC description whose body is a variable that is defined as itself corresponds to a *singular unknown* [Cou83]. Singular unknowns do not appear to be of any use, and it would be possible to define the language so that descriptions containing singular unknowns were considered syntactically invalid or ill-formed. However, this would create complications since kind-respecting beta-substitution would no longer preserve syntactic validity and well-formedness. We have therefore decided to admit singular unknowns. We arbitrarily treat all singular descriptions as convertible.

The kind inference rule for DLETREC descriptions is given below. It can be read as follows: provided that the τ_i have kind TYPE in a kind assignment in which each d_i has kind TYPE, the kind of the description as a whole is equal to the kind of the body in the same kind assignment.

$$\frac{\begin{array}{c}\forall i, 1 \leq i \leq n \;.\; B[d_1 \leftarrow \text{TYPE}] \ldots [d_n \leftarrow \text{TYPE}] \vdash \tau_i : \text{TYPE} \\ B[d_1 \leftarrow \text{TYPE}] \ldots [d_n \leftarrow \text{TYPE}] \vdash \delta : \kappa\end{array}}{B \vdash (\text{DLETREC } ((d_1\ \tau_1) \ldots (d_n\ \tau_n))\ \delta) : \kappa}$$

As indicated by the grammar and the kind inference rule, the DLETREC construct can be used only to define mutually recursive type descriptions — it cannot be used to define recursive region, effect, or higher-order descriptions. We have imposed this restriction deliberately, in order to keep the language simple.

As for recursive region or effect descriptions, they do not appear to add any power to the language because all region and effect constructors are idempotent: for example, defining d recursively as (UNION d r_1) is equivalent to defining d directly as r_1. The same is true of effects. Although the three base kinds have equal standing in most other respects, we believe that this restriction is a reasonable compromise.

Unlike recursive effects and regions, recursive higher-order descriptions could be quite useful. For example, the type function `list` of kind (DFUNC (REGION TYPE) TYPE) could be defined as the following recursive type function, given suitable parameterized types `sum` and `prod`:

```
(DLETREC ((list (DLAMBDA (r:REGION t:TYPE)
                  (sum UNIT (prod r t (list r t))))))
  ...)
```

Unfortunately, even if only first-order type functions were allowed, the descriptions generated by the DLETREC conversion rule would correspond to *algebraic trees*, because the recursion equations of a DLETREC description would form an *algebraic system* [Cou83]. Equality of regular trees is equivalent to equality of deterministic pushdown automata, the decidability of which is an open problem [Cou83].

Fortunately, many useful recursive type functions can be rewritten as nonrecursive type functions that return recursive types. For example, the type function `list` could be defined as follows:

```
(DLET ((list (DLAMBDA (r:REGION t:TYPE)
               (DLETREC ((list' (sum UNIT (prod r t list'))))
                 list'))))
  ...)
```

In view of this, we believe that our decision to disallow recursive higher-order descriptions is a reasonable compromise.

As a shorthand, we allow DLETREC to be used in the expression domain as well, with the following semantics:

(DLETREC $((d_1\ \tau_1)\ \ldots(d_n\ \tau_n))\ e$)

is equivalent to

(DLET $((d_1\ \tau_1')\ \ldots(d_n\ \tau_n'))\ e$)

where, as before, τ_i' $(1 \leq i \leq n)$ is equal to

(DLETREC $((d_1\ \tau_1)\ \ldots(d_n\ \tau_n))\ d_i$)

To avoid capture, the bound variables d_i must not appear free in the type of any free variable of the body e.

7.3.2 Recursive Expressions

MFX can easily be extended with the usual LETREC construct, the semantics of which can be expressed as a source-to-source transformation. The syntax of the LETREC construct is given by the following grammar clause:

Exp =	...	– expressions
	(LETREC ((*Var*:*Type Exp*)*) *Exp*)	– recursive expressions

The expression (LETREC $((x_1{:}\tau_1\ e_1)\ldots(x_n{:}\tau_n\ e_n))\ e$) has the following semantics. First, the e_i are evaluated, in order. Next, the resulting values are substituted for the corresponding variables x_i throughout the values of the e_i and throughout the body e. Finally, the body, thus modified, is evaluated and its value returned.

We have imposed the restriction that all the e_i must be LAMBDA or PLAMBDA expressions. This restriction is sufficient, although not necessary, to ensure that the e_i can be evaluated even though the recursively defined variables x_i are not yet bound to their values.

The type and effect inference rule for LETREC is given below. This rule can be read as follows: provided that each e_i is well-formed in a type assignment in which each x_i has the corresponding type τ_i (which must itself be of kind TYPE), and that the type of each e_i is a subtype of the corresponding τ_i, the type of the expression as a whole is equal to the type of the body in that

same type assignment. The effect of the expression is the least upper bound of the effects of the e_i and the effect of the body.

$$\frac{\begin{array}{c}\forall i, 1 \le i \le n \;.\; B \vdash \tau_i : \text{TYPE} \\ \forall i, 1 \le i \le n \;.\; A[x_1 \leftarrow \tau_1] \ldots [x_n \leftarrow \tau_n], B \vdash e_i : \tau_i' \\ \forall i, 1 \le i \le n \;.\; A[x_1 \leftarrow \tau_1] \ldots [x_n \leftarrow \tau_n], B \vdash e_i \,!\, \epsilon_i \\ \forall i, 1 \le i \le n \;.\; \tau_i' \sqsubseteq \tau_i \\ A[x_1 \leftarrow \tau_1] \ldots [x_n \leftarrow \tau_n], B \vdash e : \tau \\ A[x_1 \leftarrow \tau_1] \ldots [x_n \leftarrow \tau_n], B \vdash e \,!\, \epsilon\end{array}}{\begin{array}{lll} A, B \vdash (\text{LETREC } ((x_1{:}\tau_1\ e_1) \ldots (x_n{:}\tau_n\ e_n))\ e) & : & \tau \\ A, B \vdash (\text{LETREC } ((x_1{:}\tau_1\ e_1) \ldots (x_n{:}\tau_n\ e_n))\ e) & ! & (\text{MAXEFF } \epsilon_1 \ldots \epsilon_n\ \epsilon)\end{array}}$$

As an example of the use of the LETREC, we present an expression that computes 10 factorial using a recursive subroutine:

```
(LETREC ((fact:(SUBR (int) PURE int)
           (LAMBDA (n:int)
             (IF (<= n 1) 1
               (* n (fact (- n 1)))))))
  (fact 10))
```

The semantics of a LETREC expression can be expressed directly in terms of LET and DLETREC, using a construction that was suggested to the author by J. O'Toole. The construction is inspired by the Y-combinator, but has been adapted to ensure termination under applicative order evaluation. The basic idea is as follows:

- encapsulate each of the e_i within a LAMBDA expression that expects to be passed, as arguments, a set of functions which, when applied to themselves, yield the values of the x_i.
- throughout the scope of the bound variables, *i.e.* throughout the e_i and throughout the body, replace every occurrence of any of the bound variables by a self-application of the corresponding variable to all the x_i.

In order to make this self-application possible, these functions must all have recursive types.

The transformation is as follows: the expression

$$\begin{array}{l}(\text{LETREC } ((x_1 : \tau_1\ e_1) \\ \qquad\qquad \vdots \\ \qquad\quad (x_n : \tau_n\ e_n)) \\ \quad e)\end{array}$$

is equivalent to the LETREC-free expression

```
(DLETREC ((t1 (SUBR (t1 ...tn) PURE τ1))
                ⋮
          (tn (SUBR (t1 ...tn) PURE τn)))
  (LET ((x'1 (LAMBDA (x'1:t1 ...x'n:tn)
               e1[...(x'i x'1 ...x'n)/xi...] ))
        ⋮
        (x'n (LAMBDA (x'1:t1 ...x'n:tn)
               en[...(x'i x'1 ...x'n)/xi...] )))
    e[...(x'i x'1 ...x'n)/xi...]))
```

To illustrate this transformation, we show the transformed version of the previous example, namely the expression that computes 10 factorial using a recursive subroutine:

```
(DLETREC ((t (SUBR (t) PURE (SUBR (int) PURE int))))
  (LET ((fact' (LAMBDA (fact':t)
                 (LAMBDA (n:int)
                   (IF (<= n 1) 1
                     (* n ((fact' fact') (- n 1))))))))
    ((fact' fact') 10)))
```

It is crucial that the self-application takes place only *after* the original parameter (in this case, n) has been supplied: for example, the following expression would diverge due to the recursive call on the third line.

```
(DLETREC ((t (SUBR (t) PURE (SUBR (int) PURE int))))
  (LET ((fact' (LAMBDA (fact':t)
                 (LET ((fact (fact' fact')))
                   (LAMBDA (n:int)
                     (IF (<= n 1) 1
                       (* n (fact (- n 1))))))))
    ((fact' fact') 10))))
```

7.4 Immutable Regions

In this section we present an extension of *MFX* that is designed to permit a uniform treatment of mutable and immutable data types. This extension makes use of the types, effects and regions of *MFX*.

Most real programming languages provide a rich set of type constructors for data structuring, such as pairs, tuples, records, lists, and arrays. In many languages such types are *mutable*, *i.e.* the corresponding values have some state that can be updated and observed by the program.

In a language with an effect system, such as *MFX*, there is a pervasive difference between immutable and mutable values: reading the contents of an immutable value has effect PURE, whereas reading the contents of a mutable value has a READ effect on the corresponding region. Similarly, creating an immutable value has effect PURE, whereas creating a mutable value has an ALLOC effect on the corresponding region. In order to keep the effect descriptions of program fragments to a minimum, it is important for a language with an effect system to provide type constructors for immutable types.

Some programming languages offer the programmer two complete sets of type constructors, namely mutable ones and immutable ones [Lis79, pp. 108–120]. Using this approach, it is evident from the type of a value whether the value is mutable or immutable. However, the approach has two drawbacks. First, it requires substantial duplication in the specification of the type constructors and their operators. Second, since there is no way of viewing an immutable value as mutable, it is impossible to create, for example, a circular immutable data structure.

It is possible to combine mutable and immutable type constructors in a single, uniform framework. This technique, which makes use of the types, effects and regions of *MFX*, has the following characteristics:

1. there is no need for two separate sets of type constructors;
2. reading the contents of an immutable value has effect PURE;
3. creating an immutable value has effect PURE;
4. immutable values can be initialized imperatively.

The basic idea is very simple. Recall that all potentially mutable type constructors, such as pairs, tuples, records, lists, and arrays, are parameterized with respect the region to which the writable location(s) of the corresponding values belong. We arbitrarily designate a region, which we call the *immutable* region IM, upon which no WRITE effects are allowed. It follows that any pair, tuple, or other value whose writable locations belong to the immutable region is *immutable*. This establishes property 1 above.

Provided that the type and effect checker rejects any expression that has a (WRITE IM) effect, thereby enforcing the immutability of the region IM, it is safe to convert any (READ IM) or (ALLOC IM) effect to PURE. This establishes properties 2 and 3 above. Property 4 follows from the fact that the EXTEND

construct can be used for *any* target region, including the immutable region IM.

In summary, the notion of an *immutable* region permits a uniform treatment of mutable and immutable types, while at the same time permitting the imperative initialization of immutable values.

7.5 Polymorphism and Effects

In this section we present a variation of *MFX* that achieves polymorphic application with zero cost while keeping polymorphic values first-class. This is made possible by the *MFX* effect system.

In *MFX*, polymorphic values are first-class values, and the body of a polymorphic subroutine is evaluated each time the subroutine is applied. This design evolved as a generalization of the second-order lambda-calculus.

In polymorphic programming languages such as CLU [Lis79], all polymorphic subroutines must be defined at top-level, and the body of a polymorphic subroutine is evaluated only once for each actual type parameter to which it is applied; in the terminology of *MFX*, all polymorphic subroutines are automatically memoized. This approach has two main advantages:

1. the programmer can refer to the result of a polymorphic application by simply repeating the polymorphic application without causing unnecessary re-evaluation;
2. the state variables that are created when the body of the polymorphic subroutine is evaluated are automatically shared by all program fragments that apply the subroutine to the same actual type parameter.

Besides the fact that polymorphic subroutines are not first-class, we find that this approach has several other disadvantages:

1. the effect of a polymorphic application depends on whether it is the first application of a given polymorphic subroutine to a given actual type parameter, which cannot be predicted by the caller, and which may not be anticipated because polymorphic application normally has no effect;
2. the implementation must keep track of type information at run-time in order to determine whether a given polymorphic application is the first of a given polymorphic subroutine to a given actual type parameter;
3. the mechanism for sharing state variables allows such sharing only between program fragments that apply a polymorphic subroutine to the same actual type parameter, which seems an arbitrary and unnecessary restriction.

We believe that the semantics of languages such as CLU would be simpler, and their implementation more efficient, if the body of a polymorphic subroutine were evaluated just once, rather than once for each actual type parameter. This would address the latter two objections we raised above; and if the evaluation can take place when the expression is defined rather

than when it is applied, our first objection would also vanish. Unfortunately, evaluating the body of each polymorphic subroutine just once without further precautions would create a type loophole [Gor79a, p. 52]: it would permit the allocation of a mutable value of a type that is not closed, and the subsequent coercion of this value to multiple incompatible instantiations of this type.

In order to close this loophole, we propose a variation of the polymorphic abstraction construct of *MFX*, in which polymorphic abstraction is restricted to expressions without side-effects, *i.e.* expressions with effect PURE. Since an expression with effect PURE can be memoized without changing the meaning of the program, this restriction ensures that the body of a polymorphic subroutine can be evaluated just once, rather than once for each application (as in *MFX*) or once for each actual type parameter (as in CLU), without creating type loopholes. Moreover, provided that the programmer is informed that a program may diverge if it contains a polymorphic subroutine whose body may diverge, the body of a polymorphic subroutine can be evaluated when it is defined, rather than when it is first applied.

This proposed variation of *MFX* has the following characteristics:

1. polymorphic subroutines are first-class values;
2. there is no need to keep track of type information at run-time;
3. type computation and value computation are separated;
4. polymorphic application has no run-time cost and no side-effects; and
5. unnecessary re-evaluation of polymorphic subroutines is avoided.

This variation does not reduce the expressive power of the language: the body of a polymorphic subroutine may be a subroutine with arbitrary latent effects.

Due to the effect restriction, the body of a polymorphic subroutine can be evaluated just once, rather than once for each application or once for each actual type parameter. In practice, this means that polymorphic abstraction and application can simply be erased:

$$\mathit{Erase}(\text{PLAMBDA } (d{:}\kappa)\ e) = \mathit{Erase}(e)$$
$$\mathit{Erase}(\text{PROJ } e\ \delta) = \mathit{Erase}(e)$$

If the latent effect of a polymorphic subroutine is always PURE, it can be omitted altogether from all polymorphic subroutine types: for example, the type of the polymorphic self-composition functional can be written as

```
(POLY (t:TYPE e:EFFECT)
  (SUBR ((SUBR (t) e t)) PURE
    (SUBR (t) e t)))
```

We have adopted the effect restriction described in this section in the design of the programming language *FX* [Gif87]. Our experience with the various dialects of *FX* has confirmed our expectation that this approach is elegant, practical, expressive, and efficient.

Chapter 8. Practical Use of Effect Information

8.1 Introduction

The type and effect system of *MFX* has been designed to support a variety of program transformations and optimizations. When considering a program transformation or optimization, the compiler must determine whether the program transformation preserves the meaning of the program, and how this will affect the performance of a program. Effect and region information supports the former of these two tasks.

When compiling a program for concurrent evaluation, it is of critical importance to choose an appropriate grain size for the concurrency: the optimal grain size increases with the cost of process creation and synchronization. We have chosen to side-step this difficult issue by focusing on compilation for a dataflow architecture: the target language for our dataflow compiler has a fixed grain size.

We have implemented a compiler that translates *MFX* programs into dataflow graphs that reflect the implicit concurrency in the source program. Constraints on evaluation order are represented in the dataflow graphs by *conflict edges* and *delay edges*. We have extended an existing tagged-token dataflow simulator with operators that support (writable) memory locations and monitors; the dataflow graphs produced by the compiler can be evaluated directly by the simulator.

The purpose of this chapter is to illustrate how the effect information in an *MFX* program can be utilized to identify opportunities for concurrent evaluation, and to show how to translate an *MFX* program into a dataflow graph that reflects this implicit concurrency.

The rest of this chapter is organized as follows. We begin with a discussion of interference between program fragments. Next, we present an algorithm for computing minimal conflict graphs. The third and largest section of this chapter is dedicated to the dataflow compiler. We conclude by presenting some simulation results.

8.2 Interference

When analyzing a program for opportunities for concurrent evaluation, it is useful to be able to determine, given two expressions, whether these expressions can be evaluated concurrently without changing the meaning of the program. In a language such as *MFX*, in which subroutines and locations are first-class values, this is in general undecidable. We therefore settle for a conservative approximation. In this section we define a notion of *interference* between effect descriptions such that any two expressions that have effect descriptions that do not interfere can be evaluated concurrently without changing the semantics of the program.

We begin by considering effect descriptions without free effect or region variables; we subsequently generalize the definition to deal with arbitrary effect descriptions.

8.2.1 Closed Effect Descriptions

In a sequential language, evaluating two expressions concurrently can change the meaning of the program only if one expression may write to locations that the other expression may read or write [Ber66]. By the effect soundness proposition, the effect description of an expression gives a conservative approximation of the (global, writable) locations that the expression may read or write. This leads to the following definition of interference:

Definition. In the absence of MCALL effects, two closed effect descriptions ϵ_1 and ϵ_2 *interfere* with each other iff there exists some region constant r such that ϵ_1 has a WRITE effect on r and ϵ_2 has a READ or WRITE effect on r or vice versa (with ϵ_1 and ϵ_2 interchanged).

In a concurrent language, evaluating two expressions concurrently can change the meaning of the program even if the expressions do not read and/or write any common locations. In *MFX*, however, all such interactions are mediated by monitors and critical sections. As a result, the definition of interference between READ and WRITE effects does not need to be revised in order to deal with explicit concurrency. Moreover, *MFX* prevents any interaction between MCALL effects and READ or WRITE effects: READ and WRITE effects on a monitored region are allowed only within the scope of an EXCLUSIVE expression for the corresponding region, where MCALL effects on the region are disallowed (but could not cause interference even if they were allowed, since the expression would deadlock). As a result, the definition of interference does not need to be revised in order to deal with interference between MCALL and READ or WRITE effects.

As for interference between MCALL effects, recall that evaluating two expressions concurrently can change the meaning of the program even if the expressions do not access any common locations. This is illustrated in the following program fragment, where `m1`, `m2` and `m3` are bound to the monitors for the regions `r1`, `r2` and `r3` respectively, and where `flag1`, `flag2` and

flag3 are bound to references in the regions r1, r2 and r3 respectively, all initialized to FALSE.

```
(COBEGIN
  (LET ((val1 (EXCLUSIVE m1 (GET flag1))))
    (BEGIN
      (EXCLUSIVE m2 (SET flag2 val1))
      (EXCLUSIVE m3 (SET flag3 TRUE))))
  (LET ((val3 (EXCLUSIVE m3 (GET flag3))))
    (EXCLUSIVE m1 (SET flag1 val3))))
```

The first branch of the COBEGIN expression binds val1 to the contents of flag1, which is FALSE, saves this value in flag2, and finally stores the value TRUE in flag3. Concurrently, the second branch of the COBEGIN expression binds val3 to the contents of flag3, which may be either FALSE or TRUE depending on the relative execution speed of the two branches, and saves this value in flag1. Since the contents of flag1 is read and stored in flag2 *before* the value TRUE is stored in flag3, the final contents of flag2 is always FALSE.

The EXCLUSIVE expressions in the first branch of the COBEGIN expression have effects (MCALL r1), (MCALL r2) and (MCALL r3) respectively. Thus, it might seem as if these expressions cannot interfere with one another. However, if the expressions

```
(EXCLUSIVE m1 (GET flag1))
```

and

```
(EXCLUSIVE m3 (SET flag3 TRUE))
```

were evaluated out of order, the second branch of the COBEGIN expression could update the contents of flag1 to TRUE *before* it is read by the first branch. This would leave the value TRUE in flag2, which does not agree with the standard semantics.

As the example demonstrates, MCALL effects interfere with each other even if their regions do not overlap. This leads to the following definition of interference:

Definition (revised). Two closed effect descriptions ϵ_1 and ϵ_2 *interfere* with each other iff both have MCALL effects or there exists some region constant r such that ϵ_1 has a WRITE effect on r and ϵ_2 has a READ or WRITE effect on r or vice versa (with ϵ_1 and ϵ_2 interchanged).

Although the region parameter ρ in the effect (MCALL ρ) plays no role in the definition of interference, it is far from useless:

- programmers can regard it as machine-verifiable documentation,
- the compiler uses it to mask effects on private monitored regions, and
- we have used it to specify a language restriction that prevents deadlock due to recursive invocation of a monitor.

8.2.2 Region and Effect Variables

When region variables are added, the definition of interference depends on whether or not aliasing between region variables and region constants is permitted. If aliasing were permitted, then a WRITE effect on a region variable would interfere with a READ or WRITE effect on any region constant or variable, and a READ effect on a region variable would interfere with a WRITE effect on any region constant or variable.

Since the use of disjoint regions is the primary means provided by *MFX* to represent the fact that two effects do not interfere with each other, we have decided to prohibit all aliasing in *MFX* (see Chapter 3). This guarantees that in any scope, the regions denoted by effect constants and effect variables are disjoint. This means that for the purpose of defining interference, effect variables can simply be treated as effect constants:

- For any region variable d, the effect (WRITE d) interferes with the effects (READ d) and (WRITE d) and vice versa.

For effect variables the situation is very different: since *MFX* does not impose any restrictions on actual effect parameters, an effect variable can correspond to (almost) any effect at all. This leads to the following definition of interference:

- An effect variable interferes with any effect variable, including itself, and with any READ, WRITE or MCALL effect.

In summary, two effects ϵ_1 and ϵ_2 interfere iff

$$\exists \rho, \rho' \ . \ (\text{MCALL } \rho) \sqsubseteq \epsilon_1 \ \wedge \ (\text{MCALL } \rho') \sqsubseteq \epsilon_2$$

or

$$\exists \rho \ . \ (\text{WRITE } \rho) \sqsubseteq \epsilon_1 \ \wedge \begin{cases} (\text{READ } \rho) \sqsubseteq \epsilon_2 \\ \quad \textit{or} \\ (\text{WRITE } \rho) \sqsubseteq \epsilon_2 \end{cases}$$

or

$$\exists d, d', \rho \ . \ d \sqsubseteq \epsilon_1 \ \wedge \begin{cases} d' \sqsubseteq \epsilon_2 \\ \quad \textit{or} \\ (\text{READ } \rho) \sqsubseteq \epsilon_2 \\ \quad \textit{or} \\ (\text{WRITE } \rho) \sqsubseteq \epsilon_2 \\ \quad \textit{or} \\ (\text{MCALL } \rho) \sqsubseteq \epsilon_2 \end{cases}$$

or vice versa (with ϵ_1 and ϵ_2 interchanged).

This definition of interference serves as the basis for the effect restriction on the COBEGIN expression (see Chapter 6). However, in a COBEGIN expression interference between MCALL effects is permitted.

8.2.3 The Question of Scope

In fact, an effect variable cannot correspond to just *any* effect: for example, an effect variable can never correspond to an effect on a private region that is defined within its scope. For example, consider the expression

```
(PLAMBDA (e:EFFECT)
  (LAMBDA (p:(SUBR () e UNIT))
    (PRIVATE r
      (LET ((x (NEW r BOOL FALSE)))
        (BEGIN
          (COBEGIN
            (p)
            (SET x TRUE))
          (GET x))))))
```

(*cf.* [Hal84, p. 246]). Since the effect variable `e` is bound before the private region `r` is declared, `e` cannot possibly correspond to an effect on `r`. Under the current rule for COBEGIN, however, the above expression is not well-formed, because the effects `e` and `(WRITE r)` are assumed to interfere.

We have considered changing the definition of interference, and the effect restriction on COBEGIN expressions, to reflect the fact that an effect variable can never correspond to an effect on a private region that is defined within its scope. Unfortunately, this more powerful definition of interference relies on information that does not propagate across abstraction boundaries. For example, when the expression

```
(PLAMBDA (e:EFFECT r:REGION)
  (LAMBDA ((p:(SUBR () e UNIT))
           (x:(REF r BOOL)))
    (BEGIN
      (COBEGIN
        (p)
        (SET x TRUE))
      (GET x))))
```

is considered in isolation, it can no longer be considered well-formed. Thus, the more powerful definition of interference may discourage the use of procedural abstraction, which is contrary to our philosophy.

In contemplating various ways to refine of the definition of interference, we have concluded that where to stop is ultimately a question of taste: the interference restriction should permit expressions that *obviously* do not interfere. We regard the current definition of interference as no more than a reasonable compromise.

8.3 Constructing a Minimal Conflict Graph

In this section we show how to analyze a program for opportunities for concurrent evaluation. We only cover the language constructs of *MFX-1*; in the next section we show how to deal with the additional constructs of *MFX-2* and *MFX-3*.

When analyzing a program for opportunities for concurrent evaluation, we consider each subroutine body separately. This is a natural decomposition of the problem, since each subroutine body is a maximal program fragment such that nothing is known about what expressions may come before or after. For each subroutine, we construct a *conflict graph*, which is a directed, acyclic graph whose nodes correspond to the expressions in the subroutine body and whose edges indicate constraints on evaluation order that are due to side-effects. The constraints expressed by the conflict graph must be respected by any implementation.

By the effect soundness proposition, two expressions can conflict only if their effects interfere. Recall that the effect of an expression consists of two components: its intrinsic effect, and its inherited effects. Since the inherited effects are inherited from other expressions within the same subroutine body, it suffices to consider interference between the *intrinsic* effects of expressions.

In the absence of conditional expressions, it is easy to construct the conflict graph: simply draw a conflict edge between any two expressions whose intrinsic effects interfere. In the presence of conditionals, a conflict edge should be drawn between any two expressions that are ordered under the standard evaluation order and whose intrinsic effects interfere.

This conflict graph is *sufficient*, in the sense that it represents all the constraints on evaluation order that are due to side-effects. We next turn to the problem of constructing a conflict graph that is *minimal* given the available effect information.

In order to define what constitutes a minimal conflict graph, we must make certain assumptions about the target architecture that will interpret the conflict graph. Some of these assumptions are justified by the requirement that the architecture be feasible; others are more arbitrary. For the purpose of this chapter, we make the assumption that the target architecture has the following properties:

- an application (ordinary or polymorphic) does not begin executing until the operator subexpression has returned a value;
- a branch of an IF expression does not begin executing until the predicate has returned;
- a NEW expression does not allocate and return a location until its argument subexpression has returned and the new location has been initialized;
- a GET expression does not read and return the contents of the location in question until its argument subexpression has returned;

- a SET expression does not update the contents of the location in question until both its argument subexpressions have returned, and does not return a value until the contents of the location has been updated.

We do not consider target architectures that evaluate expressions (such as the branches of a conditional) before it is known whether or not they should be evaluated, nor do we consider architectures that return place holders (such as futures, see [Hal85]) for values that have not yet been computed.

Given these assumptions, a conflict edge may be redundant because the correct evaluation order is guaranteed by the target architecture. For example, in the expression below there is no need for a conflict edge from the GET expression to the SET expression, even though they are ordered under the standard evaluation order and their intrinsic effects interfere:

```
(SET x ((BEGIN
           exp1
           (IF (GET x) exp2 exp3))
         exp4))
```

We can represent this information in the form of a *guarantee graph*. This is a directed, acyclic graph whose nodes correspond to the expressions in a subroutine body and whose edges indicate constraints on evaluation order that are guaranteed to be satisfied by the target architecture. By design, the edges in the guarantee graph do not impose any constraints on evaluation order. Consequently, any edge that appears in the conflict graph as well as in the guarantee graph can safely be omitted from the conflict graph:

Lemma. (Guarantee Redundancy) Adding or removing a guarantee edge to or from the corresponding conflict graph does not alter the constraints imposed on evaluation order.

If a conflict edge links two expressions that can also be reached via some other chain of conflict edges, this edge is redundant: omitting it from the conflict graph, or adding such an edge, does not change the constraints that the graph imposes on evaluation order. For example, if the subexpressions `exp1`, `exp2` and `exp3` in the expression `(BEGIN exp1 exp2 exp3)` all conflict with one another, it suffices to have a conflict edge from `exp1` to `exp2` and one from `exp2` to `exp3`. In general, two conflict graphs are equivalent whenever they have the same transitive closure:

Lemma. (Transitive Closure) Any two conflict graphs that have the same transitive closure impose the same constraints on evaluation order.

The following algorithm shows how to compute, for any conflict graph G_c and corresponding guarantee graph G_g, the minimal equivalent conflict graph G_m.

1. merge G_c and G_g into a single directed acyclic graph G;
2. compute G^*, the transitive closure of G;
3. compute the least graph G^- that has transitive closure G^*;
4. subtract G_g from G^- to obtain the minimal conflict graph G_m.

Proposition. The conflict graph G_m is the minimal conflict graph given the available effect information that imposes the same constraints on evaluation order as G_c.

Proof. By the guarantee redundancy and transitive closure lemmas above, the graphs G_c, G, G^*, G^- and G_m all impose the same constraints on evaluation order. Moreover, G_m is the *minimal* such graph: any edge that is in G^* but not in G_m cannot be generated by merging the graph with G_g, taking the transitive closure, or both. G_m is unique because each step of the algorithm, including step 3, produces a unique result. □

8.4 Compilation into Dataflow Graphs

In this section we show how to translate programs from *MFX*, with its precisely specified evaluation sequence, into dataflow graphs. We view *MFX* as a functional language that has been extended with side-effect operators. Accordingly, we have patterned our compiler after an existing dataflow compiler for the functional language ID [Nik87] [Tra86]. Our compiler consists of three phases:

- the *front end* parses the program, verifies that it is well-formed, and analyzes it for implicit concurrency;
- the *graph generator* converts the parsed, annotated program to a program graph; and
- the *back end* performs various optimizations and converts the program graph to a machine graph.

Except for the treatment of constraint edges, the graph generator is quite standard, and is therefore not described here. The back end is shared with the existing ID compiler, and is not described here.

The remainder of this section is organized as follows. We begin by describing two transformations on *MFX-1* programs that simplify further processing. Next, we describe the target language. We then turn to the problem of ensuring the correct evaluation order, and we present an algorithm that computes the minimal sufficient set of extra edges that must be added to the dataflow graph of each subroutine, given the available effect information. We then turn to the issues that arise in the compilation of *MFX-2* and *MFX-3*.

8.4.1 Type Erasure

By the static typing and typeless semantics propositions, static type checking prevents all run-time type errors and all attempts to use undeclared variables or uninitialized locations, and no type, effect or region information is needed at run-time. The compiler takes advantage of these properties: after verifying that the source program is well-formed and analyzing it for implicit concurrency, the compiler erases all type information before translating the program into a dataflow graph. For example, the program fragment

```
(LAMBDA (f:(SUBR ((REF r int)) PURE int)
         x:int)
  (f (NEW r int x)))
```

is transformed into the typeless program:

```
(LAMBDA (f x)
  (f (NEW x)))
```

The type erasure algorithm is the same as that given in Chapter 4.

8.4.2 Lambda Lifting

MFX supports nested subroutine definitions and closures; the target language supports neither of these. The target language does, however support partial application. The compiler employs a technique called *lambda lifting* to convert any source program that uses closures into an equivalent program in which all subroutines are defined globally and have no free variables [Joh85]. For example, the program fragment

```
(LET ((p (LAMBDA (f x)
           (LET ((g (LAMBDA (h) (h x x))))
             (g f)))))
  ...)
```

in which the subroutine g is nested and has free variable x, is transformed into:

```
(LETREC ((p' (LAMBDA (f x)
               (g' x f)))
         (g' (LAMBDA (x h)
               (h x x))))
  ...)
```

in which the subroutine g' is defined at top-level and has no free variables. If several mutually recursive subroutines have different sets of free variables, then they must all be abstracted with respect to all of these free variables. The details of this process are straightforward [Joh85].

8.4.3 The Target Language

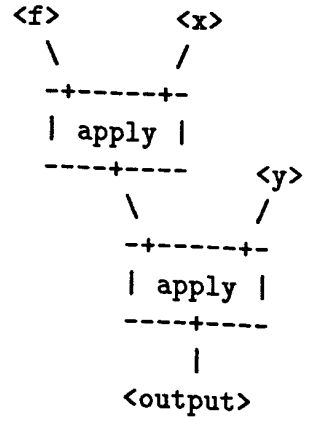
Our target language is the program graph language developed by the FLA research group [Arv87]. Programs in the target language are (directed, acyclic) *dataflow graphs* in which every node represents an operator and every edge represents a path along which *data tokens* can flow from one operator to the next. We use three kinds of operation nodes to implement the functional subset of *MFX*: application nodes, conditional evaluation nodes, and identity nodes. In addition, we use two kinds of constant nodes: ordinary constants (such as TRUE and FALSE) and subroutine nodes.

In the target language, every subroutine is represented by a dataflow graph and a corresponding *frame*. The frame of a graph indicates how the graph can be connected to other graphs: for each input edge of the graph the frame has a corresponding *source*, and for each output edge the frame has a corresponding *sink*. Frames correspond to basic blocks in conventional compilers: there are no edges that point into or out of the graph for a frame. Evaluation proceeds one frame at a time: when a frame becomes eligible for evaluation (*e.g.* as a result of subroutine application), a copy of the graph is made and a suitable input token is placed on each input edge. The evaluation of the graph terminates when every output edge has produced an output token. We use two kinds of frames: subroutine bodies and conditional branches.

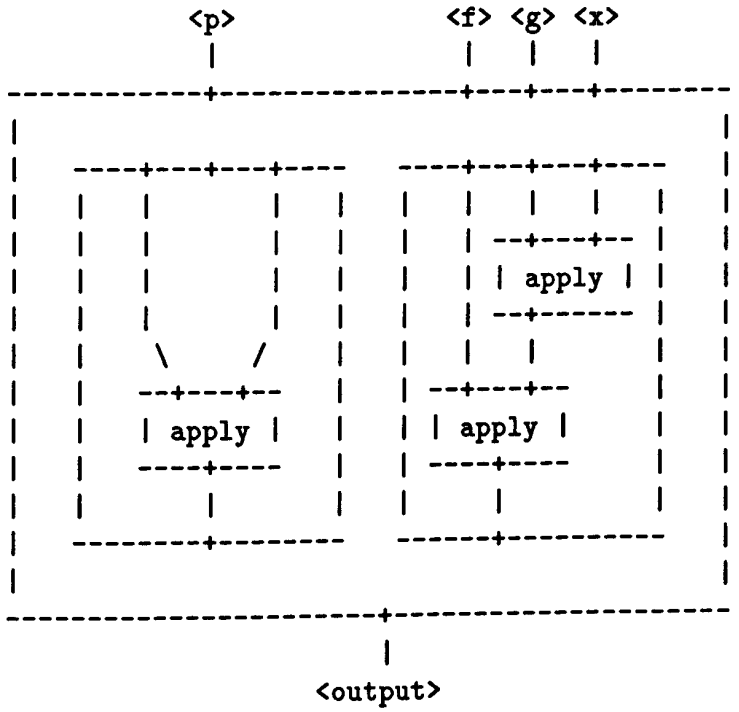
To keep the analysis and presentation simple, we deal only with frames that have a single output. Although the target language supports conditional branches with multiple outputs, we do not make use of this feature since there is no corresponding feature in *MFX*.

We use three kinds of operation nodes: application nodes, conditional evaluation nodes, and identity nodes.

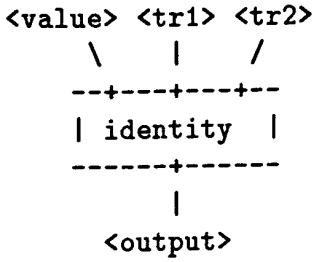
An application node has two inputs, one for the operator and one for the operand, and one output for the result. Application nodes may be curried. As an example, the graph for the expression `(f x y)` is shown below.



A conditional evaluation node contains two frames, one for each branch of the conditional. Each frame contains the entire dataflow graph for the corresponding branch. The frames indicate how the input and output edges of these graphs correspond to the free variables and result value of each branch. The conditional node itself has one input for the predicate value, one input for each free variable of the branches, and a single output for the result. As an example, the graph for the expression (IF p (f x) (f (g x))) is shown below.

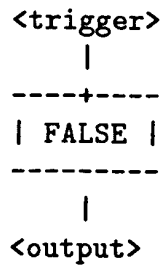


An identity node has one *value* input, zero or more *trigger* inputs and one output. An identity node passes on the value received on the value input, but only after it has received values on *all* its trigger inputs. The illustration below depicts an identity node with two trigger inputs.



We use two kinds of constant nodes: ordinary constants (such as TRUE and FALSE) and subroutine nodes.

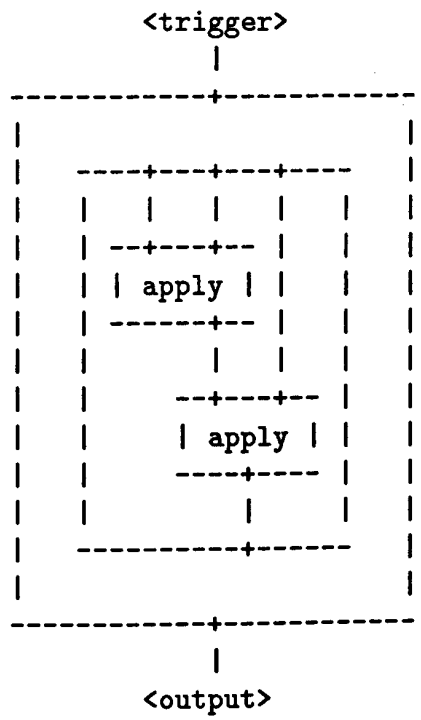
An ordinary constant node has one input and one output. The input of a constant node is called a *trigger* input: it is used to indicate to the node when it should produce a value on its output. We will not be concerned with trigger edges, since the existing ID compiler supplies them automatically when they are omitted. The illustration below depicts a constant node that produces the value FALSE.



A subroutine is represented by a frame containing the dataflow graph for its body. The frame indicates how the input and output edges of this graph correspond to the actual parameters and return value of the subroutine. A subroutine may have any number of parameters; multiple parameters are supplied by curried application. Like an ordinary constant node, a subroutine node has a single trigger input and a single value output. The graph for the subroutine

```
(LAMBDA (f:(SUBR (int int) PURE int)
         x:int y:int)
 (f x y))
```

is shown below.



Note that the subgraph for the body is connected to two parameter sources that provide the parameter values to the graph, and to a sink that receives the result of the application. Subroutines with free variables are not allowed in the target language.

In order to express *MFX* programs that use the side-effect operators NEW, GET and SET, we have extended the target language with three new built-in operation nodes: `alloc`, `read` and `write`.

- An `alloc` node has one input and one output. When a value arrives at the input, an `alloc` node allocates a new (writable) memory location, initializes it to that value, and sends it to the destination(s) of its output. `alloc` nodes are used to implement NEW expressions.
- An `read` node has one input and one output. When a location arrives at the input, a `read` node reads the contents of that location and sends it to the destination(s) of its output. `read` nodes are used to implement GET expressions.
- A `write` node has two inputs and one output. When a location arrives at the first input and a value at the second input, a `write` node updates the location in question with the value, and sends the value NIL to the destination(s) of its output. `write` nodes are used to implement SET expressions.

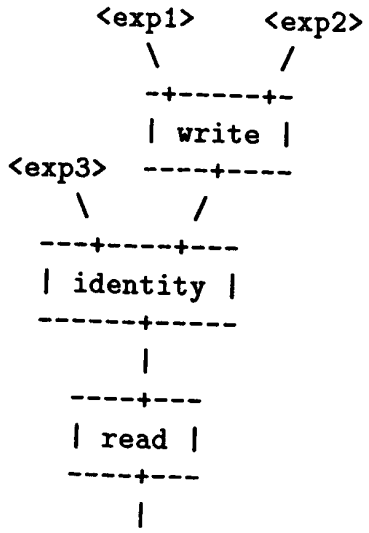
Note that these new operation nodes destroy the referential transparency of the target language just as NEW, GET and SET destroy the referential transparency of the source language.

8.4.4 Conflict and Delay Edges

In the target language, evaluation order is constrained only by the flow of data tokens from node to node. The compiler must therefore ensure that operations that may interfere with one another, such as `read` and `write` operations on locations in the same region, are evaluated in the order dictated by the standard semantics. This is done by inserting *conflict edges* and *delay edges* into the target dataflow graph.

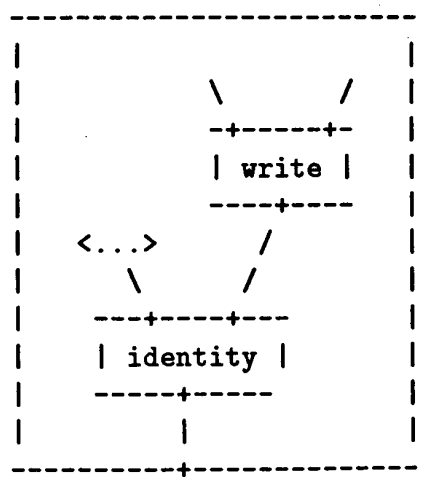
The purpose of a conflict edge is to ensure that two nodes are always evaluated in a certain order. More precisely, the purpose of a conflict edge from a node `n1` to a node `n2` is to ensure that all the effects of `n1` take place before any of the effects of `n2` take place. To this end, the conflict edge connects the output of `n1` to an identity node that delays one of the inputs of `n2`. The output of `n1` is interpreted as a signal that all the effects of `n1` have taken place.

For example, in the diagram below, evaluation of the node corresponding to the expression `(read exp3)` is delayed until after the node corresponding to the expression `(write exp1 exp2)` has been evaluated.



The purpose of a *delay* edge from a node to the output edge of the graph containing that node is to ensure that the graph does not produce a value until the node in question has been evaluated. More precisely, its purpose is to ensure that the graph does not produce a value until all the effects of the node have taken place. To this end, the delay edge connects the output of the node in question to an identity node that delays the output of the graph.

For example, in the diagram below, the value computed by the graph labeled `<...>` is delayed until after the `write` node in question has been evaluated.



There are only four kinds of nodes that can have side-effects: `alloc`, `read`, `write`, and `apply` nodes.

- An `alloc` node cannot interfere with any other computation. Thus, there is never any need to connect a conflict or delay edge to either the input or the output of an `alloc` node.
- By assumption, a `read` node does not read the contents of the location in question until its input (a location) has arrived. Thus, a `read` node can be delayed by delaying this input. Since a `read` node produces a result token only after reading the location in question, its output can be interpreted as a signal that all its effects have taken place.
- By assumption, a `write` node does not update the contents of the location until both its inputs (a location and a value) have arrived. Thus, a `write` node can be delayed by delaying either of these inputs. Since a `write` node produces an acknowledgment token only after writing the location in question, its output can be interpreted as a signal that all its effects have taken place.
- By assumption, an `apply` node does not begin executing until its subroutine input has arrived. However, it may start executing the body of the subroutine even before the actual parameter values are known. Thus, an `apply` node can be delayed only by delaying its parameter input. To ensure that an `apply` node produces a value only after all its effects have taken place, the compiler inserts appropriate delay edges into the graph for each subroutine, using the algorithm given below. This ensures that the output of an `apply` node can be interpreted as a signal that all its effects have taken place.

8.4.5 Constructing the Conflict and Delay Graphs

For each subroutine, the compiler constructs a *conflict graph* and a *delay graph*. Each is a directed, acyclic graph whose nodes are the expressions in the subroutine body. The graphs are subsequently merged with the guarantee graph for the subroutine and minimized; the remaining conflict and delay edges are inserted into the target dataflow graph.

The edges of the conflict graph indicate where in the corresponding dataflow graph conflict edges edges are needed in order to ensure that every evaluation of the dataflow graph produces a result that agrees with the standard semantics; likewise, the delay graph indicates where delay edges edges are needed.

The simplest algorithm for computing the conflict and delay graphs would be as follows:

1. to compute the conflict graph G_c, draw a conflict edge between any two expressions in the subroutine body that are ordered under the standard evaluation order and whose intrinsic effects interfere;
2. to compute the delay graph G_d, draw a delay edge from any expression that has an intrinsic effect other than ALLOC to the outermost expression of the subroutine body.

However, this algorithm is not quite acceptable: it creates edges that point into and out of conditional branches, but the target language requires that each conditional branch be a self-contained graph that can be placed in a frame.

- Edges that point *into* a conditional branch can easily be accommodated by the target language: they can be treated as if they were free variables. We have adopted this approach: every conditional node generated by the compiler has one input for the predicate value, one input for each free variable of the branches, and one input for each conflict or delay edge pointing into a branch, and the frames for the branches each have one source for each free variable and one source for each conflict or delay edge. The implementation ensures that the tokens that arrive on these edges are switched to the appropriate branch expression as needed.
- Dealing with edges that point *out of* a conditional branch is more problematic. One way to address this problem is to generalize the conditional construct to have multiple outputs: one for the value of the conditional and one for each edge that points out of a branch [Veen85]. For each such edge, a dummy source has to be created that produces a token when the branch from which the edge originates is not taken. We have adopted a simpler approach: the compiler inserts *delay edges* into the graph for each conditional branch to ensure that a conditional branch produces a value only after all the effects of the chosen branch have taken place. Edges

that would have pointed out of a conditional branch can then replaced by edges that originate from the output of the conditional.

In order to ensure a correct treatment of conditionals, we have augmented the algorithm given above with a third step:

3. redraw any edge that points out of a conditional branch so that it emanates from the output of the conditional instead.

This algorithm (steps 1, 2 and 3) can be organized as follows. Enumerate the expressions in the subroutine body, in any order that is consistent with the standard evaluation order, and maintain a list of the expressions whose evaluation always precedes that of the current expression. The branches of past conditional expressions (and their subexpressions) must be omitted from this list. For each new expression, scan the expressions in the list and draw an conflict edge for each expression that has an intrinsic effect that interferes with the intrinsic effect of the new expression. Moreover, draw a conflict edge for each conditional expression for which the least upper bound of the cumulative (*i.e.* inherited and intrinsic) effects of the branches interferes with the intrinsic effect of the new expression. When all the expressions in a subroutine body or conditional branch have been processed, scan the expressions in the list and draw a result delay edge for each expression that has an intrinsic effect other than ALLOC.

The conflict and delay graphs computed by the above algorithm can be minimized by the same procedure as the conflict graph considered earlier:

1. merge G_c, G_d and G_g into a single directed acyclic graph G;
2. compute G^*, the transitive closure of G;
3. compute the least graph G^- that has transitive closure G^*;
4. subtract G_g from G^- to obtain the minimal conflict and delay graph G_m.

Proposition. The graph G_m is the minimal conflict and delay graph given the available effect information that imposes the same constraints on evaluation order as the graphs G_c and G_d.

8.4.6 Private Regions

As we showed in Chapter 5, the semantics of PRIVATE and EXTEND can be expressed in terms of polymorphic abstraction and application. The compiler takes advantage of this fact, and transforms every program into an equivalent program that does not contain any PRIVATE or EXTEND expressions. For example, the expression

```
(PRIVATE r
  (LET ((x (NEW r BOOL FALSE)))
    (BEGIN
      (SET x TRUE)
      (GET x))))
```

is transformed into the typeless program:

```
((LAMBDA ()
   (LET ((x (NEW FALSE)))
     (BEGIN
       (SET x TRUE)
       (GET x)))))
```

Similarly, the expression

```
(EXTEND z r
  (LET ((x (NEW r BOOL FALSE)))
    (BEGIN
      (SET x TRUE)
      x)))
```

is transformed into the typeless program:

```
((LAMBDA ()
   (LET ((x (NEW FALSE)))
     (BEGIN
       (SET x TRUE)
       x))))
```

The compilation of the subroutine ensures, by means of a delay edge, that the value of `x` is not returned until after the `write` operation has been performed.

In the present compiler, the dataflow graphs for PRIVATE and EXTEND are actually subjected to procedure integration. This has the advantage that the result edges in the body of the subroutine may turn out to be redundant, or may be replaced by constraint edges that are more direct. In fact, any result delay edge that delays the value of the body of a PRIVATE expression solely because of an effect on the private region of the PRIVATE expression can simply be omitted, since the private region is inaccessible after the expression returns.

8.4.7 Monitors and Explicit Concurrency

In order to express *MFX-3* programs that use monitors, we have extended the target language with three new built-in operation nodes: `new-monitor`, `acquire-monitor` and `release-monitor`.

- A `new-monitor` node has one input and one output. When a trigger token arrives at the input, a `new-monitor` node allocates a new (writable) memory location, initializes it to IDLE, and sends it to the destination(s) of its output. `new-monitor` nodes are used to implement MONITORED expressions. The trigger token is generated automatically by the surrounding frame, and is of no further concern.
- An `acquire-monitor` node has one input and one output. When a location arrives at the input, an `acquire-monitor` node determines whether

the contents of that location is IDLE, and if so it updates the contents to BUSY and sends an acknowledgment token to the destination(s) of its output. If the contents of the location is BUSY, the destination of the output is added to an (initially empty) list of pending requests associated with the location. The entire operation is performed atomically.

- A `release-monitor` node has one input and one output. When a location arrives at the input, a `release-monitor` node sends an acknowledgment token to the destination(s) of its output. If there are no pending requests in the list associated with the location, the contents of the location is updated to IDLE. Otherwise, one of the destinations on the destination lists (for example the one that has been on the list longest) is removed from the list and is sent an acknowledgment token. `acquire-monitor` and `release-monitor` nodes are used to implement EXCLUSIVE expressions.

It is an error if the location passed to `acquire-monitor` contains something other than IDLE or BUSY, or if the location passed to `release-monitor` contains something other than BUSY. When evaluating a dataflow graph that was generated by the compiler, these errors cannot occur.

Compiling the expression (COBEGIN **exp1** ... **expN**) into a dataflow graph is relatively straightforward: it suffices to generate dataflow graphs for the subexpressions **exp1** through **expN**, and a single constant node that produces the value NIL. The output edge of the constant node is the output edge of the COBEGIN graph. No delay edges need to be added: this is done automatically, as needed, by the algorithm described earlier in this section.

As we showed in Chapter 6, the semantics of MONITORED can be expressed in terms of abstraction and application. The compiler takes advantage of this fact, and transforms every program into a form that does not contain any MONITORED expressions. For example, the expression

```
(MONITORED r m
  exp)
```

is transformed into the typeless program:

```
(LET ((m (new-monitor)))
  exp')
```

where **exp'** is the result of erasing all type, effect and region information from **exp**, and where `new-monitor` is a special subroutine that invokes the corresponding operation node (this subroutine is not available to the programmer).

Similarly, the semantics of EXCLUSIVE can be expressed in terms of LET, LAMBDA and two special subroutines, `acquire-monitor` and `release-monitor`, that invoke the corresponding operation nodes (these subroutines are not available to the programmer). For example, the expression

```
(EXCLUSIVE exp1
  exp2)
```

is transformed into the typeless program:

```
(LET ((m exp1')
      (body (LAMBDA () exp2')))
  (LET ((ack1 (acquire-monitor m))
        (result (body))
        (ack2 (release-monitor m)))
    result))
```

where exp1′ and exp2′ are the result of erasing all type, effect and region information from exp1 and exp2, respectively.

By encapsulating the body of the EXCLUSIVE expression into the subroutine (LAMBDA () exp2′) we kill two birds with one stone: first, we avoid name conflicts with the local variable m; second, compiling this subroutine in the usual way ensures that all the effects of the body take place between the calls to acquire-monitor and release-monitor. Conflict edges must be drawn from the acquire-monitor node to the apply node that invokes the body, and from there to the release-monitor node. Note that the acknowledgment values returned by acquire-monitor and release-monitor are discarded: only the value of the body is returned.

8.5 Simulation Results

The compiler described in this chapter has been implemented, and has been run on a variety of test cases and sample programs. The dataflow graphs produced by the compiler have been executed on the tagged-token dataflow simulator built by the FLA group [Arv87]. In this section we discuss some of these simulation results.

8.5.1 Simple Functional Programs

As might be expected, simple functional programs are evaluated in asymptotically optimal time as long as the computation is not limited by the number of processors. For example, consider the subroutines `fact` and `prod` defined below. They are defined such that `(fact` n`)` and `(prod 1` n`)` return the same value for all $n \geq 1$.

```
(LETREC ((fact:(the (SUBR (int) PURE int)
          (LAMBDA (n:int)
            (IF (< n 3) n
                (* n (fact (- n 1))))))))
        (prod:(SUBR (int int) PURE int)
          (LAMBDA (low:int high:int)
            (IF (= low high) low
                (LET ((mid (/ (+ low high) 2)))
                  (* (prod low mid)
                     (prod (+ mid 1) high))))))
  ...)
```

The first subroutine, `fact`, uses simple tail-recursive iteration; the second subroutine, `prod`, uses a divide-and-conquer technique. Accordingly, `fact` should take time proportional to n, whereas `prod` should take time proportional to $\lceil \log_2 n \rceil$. In both cases, the total number of instructions executed should be linear in n. This is verified by empirical results: using $T(n)$ for the latency (in instruction cycles) and $I(n)$ for the number of instructions executed, we have

$$T_{fact}(n) = \begin{cases} 28, & \text{for } n \leq 3 \\ 11n + 6, & \text{for } n \geq 3 \end{cases}$$

$$I_{fact}(n) = \begin{cases} 49, & \text{for } n \leq 3 \\ 21n + 7, & \text{for } n \geq 3 \end{cases}$$

and

$$\begin{aligned} T_{prod}(n) &= 12\lceil \log_2 n \rceil + 28 \\ I_{prod}(n) &= 44n + 7 \end{aligned} \qquad \text{for } n \geq 1$$

As might be expected, `prod` is faster than `fact` for all n greater than a certain value, namely for $n > 5$. The number of instructions executed by `(prod 1` n`)` is slightly more than twice the number of instructions executed by `(fact` n`)`, regardless of n, for all $n > 1$.

8.5.2 Imperative Programs and Effect Masking

In *MFX*, a functional program can call an imperative subroutine without losing the concurrency in the surrounding program, provided that this subroutine has a functional specification. For example, consider the expression below, which declares a private region r, defines the subroutines `circular-list` and `nth`, and returns the sum of the 30eth elements of two circular lists containing the value 1 and 2 respectively.

```
(PRIVATE r
  (LETREC ((circular-list:(SUBR (int) (ALLOC r)
                                (list r int))
            (LAMBDA (val:int)
              (EXTEND r r'
                (LET ((cons (PROJ cons r' int))
                      (emptylist (PROJ emptylist r' int))
                      (rplacd (PROJ rplacd r' int)))
                  (LET ((l (cons val emptylist)))
                    (BEGIN (rplacd l l) l))))))
           (nth:(SUBR ((list r int) int) (READ r)
                      int)
             (LAMBDA (l:(list r int) n:int)
               (LET ((car (PROJ car r int))
                     (cdr (PROJ cdr r int)))
                 (IF (= n 0) (car l)
                     (nth (cdr l) (- n 1))))))))
    (+ (nth (circular-list 1) 30)
       (nth (circular-list 2) 30)))))
```

The subroutine `circular-list` calls `rplacd`, which presumably has a (WRITE r) effect. However, this WRITE effect is masked by the EXTEND construct, so that `circular-list` itself has a latent effect of only (ALLOC r). Thus, the two calls on `circular-list` should be performed concurrently, even though both calls mutate lists in the region r. Indeed, the above expression takes precisely two instruction cycles longer than a similar expression in which the body of the LETREC expression is just `(nth (circular-list 1) 30)`. The expression as a whole returns 3, and has effect PURE.

In order to verify the performance implications of effect masking on larger programs, we implemented a subroutine that sorts an array of integers, using recursive descent quick-sort on large arrays, and switching to bubble-sort when the array size falls below a certain threshold. When a large number of processors is available, the recursive descent should be unfolded in parallel, so that the overall execution time, which is dominated by the merging of the sorted sub-arrays, should be linear in n. This prediction has been verified empirically.

Chapter 9. Conclusion

In this chapter we discuss the limitations of our approach, sketch some topics for future research, and summarize the results of our research.

9.1 Limitations

Our research has focused on the use of human-readable, machine-verifiable program specifications. In this section we discuss some of the limitations of this approach.

9.1.1 Benevolent Side-Effects

The term *benevolent side-effects* refers to side-effects on a data structure that are not observable to the user of the data abstraction. A typical example is the reorganization of a data structure that is conceptually immutable. From a program specification point of view, it may be desirable to mask benevolent side-effects. However, from the point of view of a compiler that performs automatic concurrency analysis, benevolent side-effects must not be masked, since concurrent evaluation of expressions with so-called benevolent side-effects could lead to erroneous results. Thus, there is a tension between two important clients of the effect system – the programmer and the compiler. Even if it is agreed that benevolent side-effects ought to be masked, it seems difficult to provide a general language facility for doing so in a way that is machine-verifiable. For these reasons, it seems unlikely that our type and effect system can easily be extended to support benevolent side-effects.

9.1.2 Futures

Many functional or near-functional programming languages provide constructs for eager and lazy evaluation. Two such constructs are *futures* in Multilisp and *promises* in Scheme:

- "the expression (`FUTURE` *e*) immediately returns a future for the value of the expression *e* and concurrently begins evaluating *e*. When the evaluation of *e* yields a value, that value replaces the future." [Hal85, p. 502];
- "the expression (`DELAY` *e*) returns an object called a *promise* which at some point in the future may be asked (by the `FORCE` procedure) to evaluate *e* and deliver the resulting value." [Ree86, p. 10].

Promises, which provide lazy evaluation, are simply memoized closures without arguments, and could be implemented as such within the *MFX* language. In fact, promises are provided as a standard type in the *FX* language [Gif87]: Since the difference between a promise and its value is known to the programmer, a promise can have a distinctive type, so the type system can be used to propagate its latent effect specification to all the points in the program where it may be forced, even if it may be passed as a parameter, returned as a value, or stored in a data structure.

Futures differ from promises in two respects. First, the semantics of a future specify concurrent evaluation, whereas a promise is evaluated only when it is forced. Consequently, futures provide a combination of eager and lazy evaluation, at the option of the implementation. This does not pose any special problems for the effect system, provided that the body of a future is restricted to be an expression that has no direct READ or WRITE effects. Second, a future can be used and passed wherever its value is needed, whereas a promise must be forced explicitly. Thus, the specification of a future is indistinguishable from the specification of its value.

It is this second difference that makes futures incompatible with our type and effect system: since a future has the same type as its value, there is no way to use the type system to propagate its effect specification to the points in the program where it may be used. We have not found a good way to extend our type and effect system to deal with languages that have futures or other constructs that involve call-by-name or call-by-need parameters.

9.1.3 Jumps, Exits and Continuations

In languages that rely on the compiler to identify opportunities for concurrent evaluation, the use of jumps, exits or continuations can reduce the amount of implicit concurrency drastically. For example, if the first subexpression of (BEGIN **exp1 exp2**) contains a nonlocal jump, exit, or continuation invocation, then the expressions cannot be evaluated concurrently even if their effects do not interfere. Although it may be possible to extend our type and effect system with certain special effects, such as GOTO and COME-FROM, representing various control transfers, we have not found a good way to avoid the adverse impact of such control transfers on concurrency.

9.2 Future Research

In the course of our research, we have come across far more opportunities than we have been able to investigate in detail. In this section we discuss some of the opportunities for future research that have arisen from our work.

9.2.1 Type, effect and region inference

In *MFX*, the programmer must supply complete declarations of all formal parameters, and every polymorphic subroutine must be applied to an appropriate set of description parameters before the resulting value can be used. In many programs, however, much of this information can be inferred from context. Certain programming languages, such ML and Poly, perform such type inference [Gor79a] [Mat85]. The type and effect system of *MFX*, however, is far more complex than the type system of either Poly or ML. In particular,

- ordinary application permits implicit subtyping;
- polymorphic subroutines are first-class values; and
- the constructors MAXEFF and UNION have strong algebraic properties.

In view of these factors, the general type inference problem for *MFX* may be undecidable.

However, it may be possible to solve a special case of the type inference problem, namely *implicit projection*. This term refers to the use of a polymorphic value where a monomorphic instance of it is expected. For example, if `mapcar` is abstracted over the argument type, return type, and latent effect of the mapping subroutine, then the ordinary application `(mapcar f l)` is an instance of implicit projection, shorthand for the expression

```
((PROJ mapcar τ1 τ2 ε)
 f l)
```

where `f` : (SUBR (τ_1) ϵ τ_2). In *MFX*, not only the return type of `f` propagates automatically through the enclosing application of `mapcar`, but also its latent effect.

Implicit projection greatly simplifies the textual appearance of programs, and may make programs easier to write, read, and maintain. We have adopted implicit projection in the design of the programming language *FX* [Gif87]. In general, however, the limits of type, effect and region inference in general are unclear.

9.2.2 Implicit Effect Masking

In *MFX*, effect masking is performed only on PRIVATE and EXTEND expressions. In fact, however, it seems that the effect masking rules could be applied to *any* expression, using the following rules:

$$\frac{\begin{array}{c} A,B \vdash e : \tau \\ A,B \vdash e \,!\, \epsilon \\ B \vdash d : \text{REGION} \\ x \in FV(e) \Rightarrow d \notin FDV_{desc}(A(x)) \\ d \notin FDV_{desc}(\tau) \end{array}}{A,B \vdash e \,!\, \epsilon[\psi/d]}$$

$$\frac{\begin{array}{c} A,B \vdash e : \tau \\ A,B \vdash e \,!\, \epsilon \\ B \vdash d : \text{REGION} \\ x \in FV(e) \Rightarrow d \notin FDV_{desc}(A(x)) \end{array}}{A,B \vdash e \,!\, (\text{MAXEFF } \epsilon[\psi/d]\ (\text{ALLOC } d))}$$

Although it would be possible to add these rules to *MFX*, we have not done so for two main reasons.

- Although it seems that the type and effect soundness propositions would remain valid, our current proofs would no longer suffice: the reduction axioms rely upon the presence of the PRIVATE and EXTEND expressions to indicate when fresh region constants should be allocated, and the soundness proofs rely upon these region constants, and on the *PRIVATE* and

EXTEND auxiliary expressions, to express what constitutes a well-formed state.

- By shifting the burden of effect masking from the programmer to the compiler, we would promote greater reliance on the compiler to identify optimization opportunities that may be performance-critical. This would be contrary to our philosophy: we believe that the invariants that make such optimizations possible should be expressed in the program.

Nevertheless, the principles behind implicit effect masking seem to be sound, and the concept appears to work well in practice. We have adopted implicit effect masking in the design of the programming language *FX* [Gif87].

9.2.3 Constrained quantification

The *MFX* type and effect system can be extended with *bounded quantification* over types and effects [Car85]; in fact, the compiler described in Chapter 8 supports bounded quantification. The most useful form of bounded quantification is probably the following pattern: in order to constrain a formal effect variable d_1 to effects that contain no WRITE or MCALL effects, it suffices to abstract the polymorphic subroutine in question over a fresh region variable d_2 and to bound d_1 above by (MAXEFF (ALLOC d_2) (READ d_2)). This technique can be used, for example, to define a version of `mapcar` that can be applied only to subroutines that do not interfere with themselves. Due to the effect bound on the formal parameter, the compiler described in Chapter 8 would automatically compile this version of `mapcar` so that the applications of the mapping function to the individual elements are performed concurrently as the recursion unfolds.

Ideally, one would like to be able to express more general constraints on effect and region variables than mere subeffect relationships. For example, it would be nice if one could express the absence of self-interference (as in the example above) without resort to an additional region parameter. Similarly, it would be nice if one could constrain two formal effect variables d_1 and d_2 to effects that do not interfere with each other, or if one could express the constraint that a given effect variable does not correspond to any effect on a given region variable. Finally, it would be nice to have a means of indicating explicitly whether or not certain region variable are allowed to be aliases for certain other region variables or constants: this would eliminate the need for the current general anti-aliasing rule, which has proved to be rather restrictive in practice.

The main disadvantage of bounded quantification and of more general constrained quantification seems to be the sheer complication of the resulting type and effect system. This has an impact in three areas:

- programmers will suffer if they have to specify the necessary constraints on formal parameters;
- the semantics of the language and the soundness proofs will be complicated dramatically;

- type and effect checking and inference will be more complicated and less efficient.

Despite these complications, we believe that constrained quantification is an attractive means of augmenting the power of a type or type and effect system.

9.2.4 Processes as First-Class Values

It appears that *MFX* can be extended with asynchronous processes with unlimited lifetimes that can be treated as first-class values. Such processes can be regarded as a combination of futures and promises:

- The expression (`FORK` e) returns an object called a *process* and concurrently begins evaluating e. At some point in the future, the process may be asked (using the `JOIN` expression) to deliver the resulting value.

Since a process must be joined explicitly, it can have a distinctive type, so the type system can be used to propagate its latent effect specification to all the points in the program where it may be joined. Thus, the possible effects caused by joining a process can be analyzed syntactically.

A forked process differs from a promise in that its semantics specify concurrent evaluation, whereas a promise is evaluated only when it is forced. This does not pose any special problems, provided that the body of a process is restricted to be an expression that has no direct READ or WRITE effects.

Because of its semantics, a FORK expression may be evaluated lazily, and a JOIN expression eagerly, without changing the semantics of the program. This can be modeled using two new effect constructors, START and TOUCH, which are defined so that

$$(\text{MCALL } \rho) \simeq (\text{MAXEFF } (\text{START } \rho)\ (\text{TOUCH } \rho))$$

Using this equivalence, the effect of a FORK expression can be defined as the ALLOC and START effects of its body, while the TOUCH effects of the body form the latent effect of the resulting process. The constraints on evaluation order can then be expressed in terms of an asymmetric definition of interference between START and TOUCH effects.

A forked process differs from a COBEGIN branch in that its lifetime is not limited. In particular, a process that accesses a given monitored region may survive the MONITORED expression that created the region. As a result a private monitored region can not simply be reclaimed automatically when the expression that created it returns.

If forked processes are allowed to survive surrounding MONITORED expressions, effects on a private monitored region cannot be masked. This is illustrated in the following program fragment. In this program fragment we assume that all effects on a MONITORED region can be masked and derive a contradiction.

```
(MONITORED r1 m1
  (LET ((y (NEW r1 int 0))
        (fork (NEW r1 (PROCESS pure UNIT) (FORK NIL)))
        (flag (NEW r1 bool FALSE)))
    (LET ((f (FORK
               (MONITORED r2 m2
                 (LET ((x (NEW r2 int 1)))
                   (BEGIN
                     (FORK
                       (BEGIN
                         (WHILE (not (EXCLUSIVE m1 (GET flag)))
                           NIL)
                         (LET ((ff (EXCLUSIVE m1 (GET fork))))
                           (BEGIN
                             (JOIN ff)
                             (EXCLUSIVE m2
                               (EXCLUSIVE m1
                                 (SET y (GET x))))))))
                     (EXCLUSIVE m2 (SET x 2))))))))
      (BEGIN
        (EXCLUSIVE m1 (SET fork f))
        (EXCLUSIVE m1 (SET flag TRUE))
        (EXCLUSIVE m1 (GET y))))))
```

The intended operation of this program fragment is as follows. First, the variables `y`, `fork` and `flag` are allocated in the monitored region `r1` and initialized. Next, the fork `f` is created and stored in the reference `fork`, and `flag` is set to TRUE. Finally, the contents of `y` is read and returned. Since `y` was initially `0`, the program fragment may return the value `0`. Another possibility is due to the fork `f`. This fork declares a private monitored region `r2`, where it allocates and initializes a variable called `x`. It then creates a second fork, updates the contents of `x` to 2, and returns NIL. This second fork, finally, waits until the contents of `flag` is TRUE, which indicates that `f` has been stored in `fork`, retrieves `f`, JOINs it (to force the contents of `x` to be updated to 2), copies the contents of `x` to `y`, and returns NIL. Since all this may take place before the outermost expression reads and returns the contents of `y`, the program fragment may also return the value 2.

Since the process `f` has latent effect PURE (by assumption), the compiler need not ensure that the expression (JOIN `ff`) is completed before the contents of `x` is copied to `y`. This leads to a third possible outcome: because of the incorrect effect declaration, the program fragment may return the value `1`. However, this outcome is inconsistent with the requirement that every actual execution sequence is equivalent to some execution sequence in which the expressions within each process are evaluated in applicative order.

It follows that the assumption we made must be incorrect: effects on a private monitored region cannot be masked when processes may have unlimited lifetimes.

Despite these complications, we believe that it is possible to design a set of constructs for creating and joining first-class process values in *MFX* in a way that preserves and extends the properties of the type and effect system.

9.2.5 Object-level monitors

In *MFX*, each monitor corresponds to a particular monitored region. Since there is at most one monitor for each region, and since this region appears free in the type of the monitor, distinct monitors have different types. This makes it impossible to manipulate sets of monitors or sets of objects containing monitors.

In certain programming languages, such as MESA, it is possible to define classes of *monitored objects*, in which each object has its own monitor lock [Lam80, p. 110]. The use of individual monitors can drastically increase the maximum concurrency that can be obtained. Although *MFX* currently provides no such capability, it seems that it could be augmented with a construct for defining a class of monitored objects that belong to the same monitored region but that have individual monitor locks.

9.2.6 Application of Effect Information

In this thesis we have sketched one application of effect information in some detail, namely compilation for a dataflow architecture. In fact, however, the information provided by a type and effect system can be used for compilation for a variety of other architectures, including conventional shared-memory and distributed-memory multiprocessors. Although this thesis has avoided the issue of grain size (see Chapter 8), it appears that a compiler for a multiprocessor can use effect information to identify opportunities for concurrent evaluation in a way that is similar to the way in which such information is used by our dataflow compiler. Furthermore, a compiler for a multiprocessor may be able to use knowledge about the distinction between mutable and immutable regions (see Chapter 7) to maintain cache coherence.

As we showed in Chapter 5, a type and effect system with private regions can be used for storage reclamation. Unlike conventional garbage collection algorithms, the reclamation of private regions is not based on the mere existence of references, but rather on high-level information about whether or not these references can be used. Specifically, the reclamation of a particular location is independent of whether or not references to that location may still exist. As a result, it appears that storage reclamation based on region information may be particularly suited for distributed storage reclamation.

Finally, type and effect systems can also improve compilation for uniprocessor architectures in at least two respects: concurrency can be used to keep a uniprocessor busy during slow I/O operations and page faults, and memoization can reduce the overall amount of computation required.

9.3 Summary

In a language with a *type and effect* system, every expression has a *type*, which describes the value returned by the expression, and an *effect*, which describes the potential effects of the expression. Effects are defined in terms of *regions*, which correspond to partitions of the store. The type of every subroutine includes a specification of the *latent effect* of the subroutine; the type of every location includes a specification of the region to which the location belongs.

In analyzing the theory and practice surrounding type and effect systems, we have shown that

- a conventional type system, to wit the typed lambda-calculus, can be extended into a type and effect system;
- the familiar type polymorphism of the second-order lambda-calculus can be generalized to polymorphism with respect to types, effects, and regions;
- effects on private regions can, under certain circumstances, be masked by the type and effect inference system;
- an effect system based on regions can be used to ensure that all interactions between concurrent expressions are mediated by monitors and critical sections;
- effect information can be used to compile imperative programs into dataflow graphs that reflect the implicit concurrency in the source program.

We believe that our research represents a step towards the integration of functional and imperative programming by providing a single programming model in which imperative and functional program fragments can coexist and interact. Moreover, we believe that it represents a step towards the integration of implicit and explicit concurrency by providing a programming model in which the analysis of implicit concurrency is not hampered by the presence of explicit concurrency.

Bibliography

Abe85 *Structure and Interpretation of Computer Programs*, Harold Abelson, Gerald Jay Sussman and Julie Sussman, MIT Press/McGraw Hill (1985)

Arv87 *Executing a Program on the MIT Tagged-Token Dataflow Architecture*, Arvind and Rishiyur S. Nikhil, MIT LCS Computation Structures Group Memo No. 271 (March 1987)

Ban79 *An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables*, John P. Banning, Fifth Annual ACM Symposium on Principles of Programming Languages (January 1979), pp. 29–41

Bar84 *The Lambda Calculus — Its Syntax and Semantics*, H. P. Barendregt, Studies in Logic and the Foundations of Mathematics, Vol. 103, North-Holland (1984)

Bar78 *A Practical Interprocedural Data Flow Analysis algorithm*, Jeffrey M. Barth, Communications of the ACM, Vol. 21, No. 9 (September 1978), pp. 724–736

Ber66 *Analysis of Programs for Parallel Processing*, A. J. Berstein, IEEE Transactions on Electronic Computers, Vol. EC-15, No. 5 (October 1966), pp. 757-763

Bri72 *Structured Multi-programming*, Per Brinch-Hansen, Communications of the ACM, Vol. 15, No. 7 (July 1972), pp. 574–578

Car85 *On Understanding Types, Data Abstraction, and Polymorphism*, Luca Cardelli and Peter Wegner, Brown University Technical Report No. CS-85-14 (1985)

Car86 *A Polymorphic λ-calculus with Type:Type*, Luca Cardelli, DEC Systems Research Center (May, 1986)

Cou83 *Fundamental Properties of Infinite Trees*, B. Courcelle, Theoretical Computer Science, Vol. 25, No. 2 (March 1983), pp. 95–169

Fel87 *A Calculus for Assignments in Higher-Order Languages*, Matthias Felleisen and Daniel P. Friedman, Fourteenth Annual ACM Symposium on Principles of Programming Languages (January 1987), pp. 314–325

Gif86 *Integrating Functional and Imperative Programming*, David K. Gifford and John M. Lucassen, 1986 ACM Conference on LISP and Functional Programming (August 1986), pp. 28–38

Gif87 *FX-87 Reference Manual*, David K. Gifford *et al.*, MIT LCS TR-407, in preparation (August 1987)

Gor79a *Edinburgh LCF*, Michael J. C. Gordon, Robin Milner and Christopher Wadsworth, Lecture Notes in Computer Science no. 78, Springer Verlag (1979)

Gor79b *The Denotational Description of Programming Languages — An Introduction*, Michael J. C. Gordon, Springer Verlag (1979)

Hal84 *The Semantics of Local Storage, or What Makes the Free-List Free? (Preliminary Report)*, Joseph Y. Halpern, Albert R. Meyer and B. A. Trakhtenbrot, Eleventh Annual ACM Symposium on Principles of Programming Languages (January 1984), pp. 45–257

Hal85 *Multilisp: A Language for Concurrent Symbolic Computation*, Robert H. Halstead, Jr., ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4 (October 1985), pp. 501–538

Hoa74 *Monitors: An Operating System Structuring Concept*, C. A. R. Hoare, Communications of the ACM, Vol. 17, No. 10 (Oct. 1974), pp. 549–557

Hud86 *A Semantic Model of Reference Counting and its Abstraction (Detailed Summary)*, Paul Hudak, 1986 ACM Conference on LISP and Functional Programming, pp. 351–363

Joh85 *Lambda Lifting: Transforming Programs to Recursive Equations*, Thomas Johnsson, in *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science No. 201, Springer Verlag (1985), pp. 190-203

Lam77 *Report on the Programming Language Euclid*, Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell and Gerald J. Popek, SIGPLAN Notices, 12 (1977), pp. 1–79

Lam78 *Revised Report on the Programming Language Euclid*, Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell and Gerald J. Popek, Xerox Technical Center TR CSL-78-2 (1978)

Lam80 *Experience with Processes and Monitors in Mesa*, Butler W. Lampson and David D. Redell, Communications of the ACM, Vol. 23, No. 2 (February 1980), pp. 105–117

Lis79 *CLU Reference manual*, Barbara H. Liskov *et al.*, MIT LCS TR-225 (1979)

Mar83 *The Bath Concurrent Lisp machine*, Jed B. Marti and John Fitch, Eurocal '83, European Computer Algebra Conference, Lecture Notes in Computer Science No. 162 (1983), pp. 78–90

Mat85 *POLY Manual*, D. C. J. Matthews, SIGPLAN Notices, Vol. 20, No. 9 (September 1985), p. 52–76

McC79 *An Investigation of a Programming Language with a Polymorphic Type Structure*, Nancy Jean McCracken, Ph. D. Thesis, Syracuse University School of Computer and Information Science (June 1979)

McC82 *A Finitary Retract Model for the Polymorphic Lambda-Calculus*, Nancy Jean McCracken, submitted for publication in *Information and Control* (1982)

Mit84 *Type Inference and Type Containment*, John Clifford Mitchell, International Symposium on Semantics of Data types, Lecture Notes in Computer Science no. 173 (1984), pp. 257–277

Nei86 *Computation of Aliases and Support Sets*, Anne Neirynck, Prakash Panagaden and Alan J. Demers, Cornell University TR 86-763 (July 1986)

Nik87a *Id Nouveau Reference Manual — Part I: Syntax*, Rishiyur S. Nikhil, MIT LCS Computation Structures Group Memo (April, 1987)

Nik87b *Id Nouveau Reference Manual — Part II: Operational Semantics*, Rishiyur S. Nikhil and Keshav K. Pingali, MIT LCS Computation Structures Group Memo (April, 1987)

Plo81 *A Structural Approach to Operational Semantics*, G. Plotkin, Computer Science Department Report, Aarhus University (1981)

Pop77 *Notes on the Design of Euclid*, G. J. Popek, J. J. Horning and R. L. London, Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3 (March 1977), pp. 11–18

Ree86 *Revised*^{3} *Report on the Algorithmic Language Scheme*, Jonathan Rees, William Clinger (eds.), SIGPLAN Notices, Vol. 21, No. 12 (December 1986), pp. 37–79

Rey74 *Towards a Theory of Type Structure*, John C. Reynolds, International Programming Symposium, Lecture Notes in Computer Science No. 19, Springer Verlag (1974), pp. 408–425

Rey78 *Syntactic Control of Interference*, John C. Reynolds, Fifth Annual ACM Symposium on Principles of Programming Languages (January 1978), pp. 39–46

Sco82 *Domains for Denotational Semantics*, Dana S. Scott, in *Automata, Languages and Programming*, Lecture Notes in Computer Science No. 140, Springer-Verlag (1982), pp. 577–613

Ste78 *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*, Guy Lewis Steele Jr., MIT AI TR-474 (May 1978)

Sto77 *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Joseph E. Stoy, MIT Press (1977)

Ten76 *The Denotational Semantics of Programming Languages*, R. D. Tennent, Communications of the ACM, vol. 19 no. 8 (1976), pp. 437–453

Ten83 *Semantics of Interference Control*, R. D. Tennent, Theoretical Computer Science, Vol. 27 (1983), pp. 297–310

Tra86 *A Compiler for the MIT Tagged-Token Dataflow Architecture*, Kenneth R. Traub, S.M. Thesis, MIT Laboratory for Computer Science (August, 1987)

Vee85 *The Misconstrued Semicolon — Reconciling Imperative Languages and Dataflow Machines*, Arthur Hugo Veen, Doctoral Thesis, Technische Hogeschool Eindhoven, the Netherlands (September 1985)

Wei80 *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables*, William E. Weihl, Seventh Annual ACM Symposium on Principles of Programming Languages (January 1980), pp. 83–94

Index of Definitions