Polymorphic Effect Systems

by
John M. Lucassen * and David K. Gifford
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

We present a new approach to programming languages for parallel computers that uses an *effect system* to discover expression scheduling constraints. This effect system is part of a 'kinded' type system with three base kinds: *types*, which describe the value that an expression may return; *effects*, which describe the side-effects that an expression may have; and *regions*, which describe the area of the store in which side-effects may occur. Types, effects and regions are collectively called *descriptions*.

Expressions can be abstracted over any kind of description variable — this permits type, effect and region polymorphism. Unobservable side-effects can be *masked* by the effect system; an effect soundness property guarantees that the effects computed statically by the effect system are a conservative approximation of the actual side-effects that a given expression may have.

The effect system we describe performs certain kinds of side-effect analysis that were not previously feasible. Experimental data from the programming language FX indicate that an effect system can be used effectively to compile programs for parallel computers.

This work was supported in part by DARPA/ONR contract number N00014-83-K-0125

* Currently at IBM Tokyo Research Laboratory, 5-19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan

1. Introduction

We present a new approach to programming that is intended to combine the advantages of functional and imperative programming. Our approach uses an *effect system* in conjunction with a conventional type system to compute both the type and the effect of each expression statically. The *effect* of an expression is a concise summary of the observable side-effects that the expression may have when it is evaluated. If two expressions do not have interfering effects, then a compiler may schedule them to run in parallel subject to dataflow constraints. The effect system described in this paper is an integral part of the programming language FX [Gif87].

The effect system we present is capable of certain kinds of side-effect analysis that were not previously feasible. In particular, the effect system permits concurrency analysis in the presence of first-class function values, and it permits the masking of side-effects on local data values even in the presence of first-class, heap-allocated values of user-defined types. (A value is first-class if it can be stored, passed as an argument, and returned as a result.) In particular, the effect system is able to mask effects on first-class, user-defined, heap-allocated data structures, which no previously published static method can do. An effect soundness property guarantees that the effects computed statically by the effect system are a conservative approximation of the actual side-effects that a given expression may have.

We distinguish three sorts of effects: READ, WRITE, and ALLOC effects, where allocation includes initialization. Each effect is subscripted by the region where the effect may occur. Compound effects can be constructed as unions of simple effects, and thus effects form a lattice. The bottom of the effect lattice is the effect PURE, which is used to describe expressions that have no side-effects.

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)

The type and effect system is based on a 'kinded' type system for the second-order lambda calculus [McC79]. Kinds are the types of descriptions. The type and effect system has three base kinds: *types*, which describe the value that an expression may return; *effects*, which describe the side-effects that an expression may have; and *regions*, which describe the area of the store in which side-effects may occur. Types, effects and regions are closely interrelated; in particular, a subroutine type incorporates a *latent effect*, which describes the side-effects that the subroutine may have when it is applied; and a reference type incorporates a region, which describes where the reference is allocated. The kind system is used to verify the well-formedness of descriptions; the type and effect system is used to verify the well-formedness of expressions.

The effect system is designed to be useful to programmers, compiler writers, and language designers in the following respects:

- An effect system helps a programmer specify the side-effect properties of program modules in a way that is machine-verifiable. Effect specifications are a natural extension of the type specifications found in conventional programming languages. We believe that the use of effect specifications has the potential to improve the design and maintenance of imperative programs.
- An effect system helps the compiler identify optimization opportunities that are hard to detect in a conventional higher-order imperative programming language. We have focused our research on two classes of optimizations: evaluation order, including eager, lazy, and concurrent evaluation and code motion, and common subexpression elimination, which includes memoization. We believe that the ability to perform these optimizations effectively in the presence of side-effects represents a step towards integrating functional and imperative programming for the purpose of massively parallel programming.
- An effect system lets the language designer express and enforce side-effect constraints in the language definition. For example, by limiting polymorphism to expressions without side-effects, we have been able to construct the first type system known to us that permits an efficient implementation of fully orthogonal polymorphism (in which any expression can be abstracted over any description variable and in which all polymorphic values are first class) in the presence of side-effects.

In the remainder of this paper we review related work (Section 2), introduce the principles of type and effect checking by means of examples (Section 3), present the formal type and effect inference rules which constitute the static semantics of the language (Section 4), present the dynamic semantics and discuss the soundness of the type and effect system (Section 5), show how soundness results can be extended to a language with effect masking (Section 6), discuss our experience with an implementation of the FX programming language (Section 7), and summarize our results (Section 8).

2. Relation to Other Work

In this section we compare our research with other work on conventional flow analysis, abstract interpretation, syntactic interference control, and earlier work on effect systems.

- In conventional flow analysis [Ban79, Bar78, Wei80] the local effects of each subroutine are first computed, and then the true effects of each subroutine are computed using a transitive closure or graph flow algorithm on the subroutine call graph. These methods require that the calling pattern between subroutines be known statically, so that the call graph can be computed. Thus, conventional flow analysis can not deal with first-class subroutines or separate compilation. Our effect system, on the other hand, is able to handle first-class subroutines as well as separate compilation because the latent effect of each subroutine is evident from its type. Our approach also differs from conventional flow analysis in that all effect information is visible to the programmer, and must therefore be expressed in human-readable form, preferably in terms that are related to the programmer's view of the program.
- Neirynck *et al.* [Nei87] show how to compute support sets and aliasing relationships statically using abstract interpretation. Within this framework the side-effects of a subroutine invocation can depend on the side-effects of its procedural arguments, in a way that resembles our notion of effect polymorphism. Neirynck *et al.* restrict their attention to a language in which subroutines and references are not storable, and in which references can not be returned out of the scope in which they are created (*i.e.* heap allocation is not supported). In our approach, the use of the type system allows both subroutines and references to be first-class values.
- In his paper on syntactic control of interference, Reynolds [Rey78] presents a simple syntactic method for computing the support set of an

expression, with the objective of permitting concurrent evaluation of two expressions only if their support sets do not overlap. This method does not distinguish between the variables accessed by the expression, on the one hand, and the support set of the value returned by the expression on the other hand. As a result, the restrictions on concurrent evaluation are unnecessarily conservative. Our effect inference rules, on the other hand, maintain a clear distinction between support sets (which are described by regions) and side-effects (which are described by effects).

- The programming language Euclid [Lam77, Pop77] was designed to aid program verification via static side-effect and aliasing restrictions. Euclid includes *collections*, which are similar to our notion of regions, and it has two kinds of abstractions: procedures, that are executed only for their side-effects, and functions, that are executed only for their result value. Functions are permitted to have side-effects on internal objects. Euclid's collections are not first-class values, and thus they are a source of non-uniformity in the language. Euclid's side-effect restrictions are less general than an effect system, and thus Euclid does not provide effect polymorphism or general effect masking.
- The Rabbit compiler [Ste78] distinguishes five different kinds of side-effects via a static analysis system. Like an effect system, Rabbit computes the side-effects of an expression based upon its effects and the side-effects of its subexpressions. However, Rabbit does not include effect polymorphism or effect masking, nor does it provide a mechanism to communicate the effects of a procedure from the point of definition to the point of use.
- The idea of effect systems, and their application to compilation and to first-class polymorphism, were originally introduced in [Gif86]. This earlier paper distinguishes four 'effect classes' ranging from PURE to PROCEDURE, and shows how the effect class of a subroutine can be determined syntactically. There is no abstraction over effect classes; moreover, there is no way to mask unobservable side-effects. In the present paper, the elevation of effects to the status of first-class descriptions contributes to the uniformity of the system. Moreover, we are able to state and prove [Luc87] an important effect soundness property.

3. Examples of Effects and Types

In order to give the reader an intuitive idea of how an effect system can be used we first present examples of effect polymorphism and effect masking. The examples are written in *MFX* [Luc87], a statically scoped Lisp dialect with a type and effect system. The name *MFX* stands for "mini-*FX*"; *MFX* is a subset of the full *FX* language [Gif87].

We first consider the polymorphic subroutine **twice**, which takes a subroutine of a single argument and returns the result of composing that subroutine with itself:

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

twice is polymorphic in both the argument type **t** and the latent effect **e** of its argument. This is reflected in the type of **twice**, which is inferred from the above definition by the type inference rules:

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

The **POLY** type reflects the fact that **twice** is polymorphic with respect to a type **t** and an effect **e**. Note that the *kinds* of the description variables **t** and **e** are specified in the **POLY** type. In what follows we will omit the latent effect of a **POLY** type whenever it is **PURE**. However, we will never omit the latent effect of a **SUBR** type.

A more elaborate example of polymorphism is the **mapcar** subroutine, which is defined below. We have omitted the details of the recursive definition; note, however, that the type and the effect of the body of **MAPCAR** are declared in advance using **THE**. The effect constructor **MAXEFF** returns the effect that is the least upper bound of its arguments.

```
mapcar =
 (PLAMBDA (t1:TYPE t2:TYPE r:REGION
           e:EFFECT)
   (LAMBDA (f:(SUBR (t1) e t2)
            input:(listof r t1))
     (THE (MAXEFF (ALLOC r) (READ r) e)
          (listof r t2)
          (IF (null? input) empty
            ((PROJ cons r)
             (f (car input))
```

```
                (mapcar f (cdr input)))))))
```

mapcar is polymorphic in the type of the elements of its input list, the type of the elements of its output list, the region in which these lists are allocated, and the latent effect of the subroutine that is being mapped. This is reflected in the type of **mapcar**, which is inferred from the above definition by the type inference rules given suitable declarations for **listof**, **null?**, **cons**, **car** and **cdr**:

```
mapcar :
 (POLY (t1:TYPE t2:TYPE r:REGION
        e:EFFECT) PURE
   (SUBR ((SUBR (t1) e t2) (listof r t1))
         (MAXEFF (READ r) e (ALLOC r))
         (listof r t2)))
```

The latent effect of **mapcar** has three distinct components: the effect of reading the input list, the effect of applying the mapping subroutine, and the effect of allocating the output list. Note that the **listof** type constructor incorporates both the region in which the list is allocated and the type of the elements of the list.

Our final example is a simple case of effect masking:

```
example =
 (LET ((y ((PROJ cons @red) 1 2)))
   (set-car! y 2)
   (car y))
```

This expression allocates a cons-cell initialized to $(1,2)$ in the region **@red**, mutates the **car** of the pair to be 2, and then returns the **car** of the pair. Although the expression allocates, writes, and reads the region **@red**, the effect masking rules can prove that this expression has no observable side-effects. As a result, the expression as a whole has effect PURE. In general, effects on a given region can be masked from the effect of a given expression whenever the region does not appear free in the type of the expression or in the type of any free variable of the expression. This is discussed in more detail in Section 6.

4. Inference Rules for Types and Effects

This section presents a set of type and effect inference rules for the language *MFX*. *MFX* is based on the second-order typed lambda-calculus and on the higher-order 'kinded' lambda-calculus of McCracken [McC79, Rey74]. The language consists of three layers: expressions, descriptions (which include types, effects and regions), and kinds (which are the 'types' of descriptions). There are three kind constants: REGION, EFFECT and TYPE; there are no higher-order kinds in *MFX*.

Kind =

– kinds (κ)

REGION

– the kind of regions

EFFECT

– the kind of effects

TYPE

– the kind of types

There are two general classes of *MFX* expressions: expressions that come from the higher-order lambda-calculus, and expressions that deal with side-effects.

Exp =

– expressions (e)

Var

– ordinary variable (x)

(LAMBDA (*Var*:*Type*) *Exp*)

– ordinary abstraction

– (compare with $\lambda x{:}\tau\ .\ e$)

(*Exp Exp*)

– ordinary application

(PLAMBDA (*Dvar*:*Kind*) *Exp*)

– polymorphic abstraction

– (compare with $\Lambda d{:}\kappa\ .\ e$)

(PROJ *Exp Desc*)

– polymorphic application

(NEW *Region Type Exp*)

– allocating and initializing a location

(GET *Exp*)

– reading a location

(SET *Exp Exp*)

– writing a location

Expressions that cannot be reduced are considered to be *values*. The only expressible values are the abstraction expressions listed below. Other values, such as locations, are not directly expressible and will be presented later.

Val =

– values (v)

(LAMBDA (*Var*:*Type*) *Exp*)

– ordinary abstraction

(PLAMBDA (*Dvar*:*Kind*) *Exp*)

– polymorphic abstraction

The expressions NEW, GET and SET interact with the store. We distinguish three ways of interacting with the store: the allocation and initialization of memory locations, the reading (or dereferencing) of locations, and the writing (or updating) of locations. As we will see, each of these store interactions is reflected in the syntactic *effect* of the corresponding expression.

Syntactically, effects are specified in terms of *regions*, which correspond to infinite sets of locations. Thus, effects may be of the following forms:

Effect =

– effect descriptions (ϵ)

Dvar

– effect variable

(ALLOC *Region*)

– allocating in a given region

(READ *Region*)

– reading from a given region

(WRITE *Region*)

– writing to a given region

(MAXEFF *Effect**)

– combination of zero or more effects

PURE

– "no effect"; synonym for (MAXEFF)

Syntactically, regions can be region constants, region variables, the empty region (it contains no locations), and unions of multiple regions:

Region =

– region descriptions (ρ)

Rconst

– region constant (r)

Dvar

– region variable

(UNION *Region*$^+$)

– union of one or more regions

In *MFX*, the type of an (ordinary or polymorphic) subroutine incorporates not only the type (or kind) of the formal parameter and the type of the returned value, but also the *latent effect* of the subroutine. The type of a location incorporates not only the type of its contents, but also the *region* to which the location belongs.

Type =

– type descriptions (τ)

Dvar

– type variable

(SUBR (*Type*) *Effect Type*)

– types of ordinary subroutines

– (compare with $\tau \xrightarrow{\epsilon} \tau'$)

(POLY (*Dvar*:*Kind*) *Effect Type*)

– types of polymorphic subroutines

– (compare with $\forall d : \kappa \in \tau$)

(REF *Region Type*)

– types of locations

The set of *descriptions* is made up of all types, effects and regions:

Desc =

– descriptions (δ)

Region

– region descriptions

Effect

– effect descriptions

Type

– type descriptions

A description is well-formed with respect to a *kind assignment* B : *Dvar* $\rightarrow$ *Kind* iff it has a *kind* with respect to B. The kind assignment maps description variables to their kinds. We omit the kind inference rules, which are straightforward.

The generalized subtype relation $\sqsubseteq$ on descriptions is defined to correspond to the subset relation on the underlying sets of locations, state interactions, or values respectively. For example, $\rho \sqsubseteq$ (UNION ρ, ρ'), PURE $\sqsubseteq \epsilon$, and (REF $\rho\ \tau$) $\sqsubseteq$ (REF (UNION ρ, ρ') τ). We omit the formal definition. The generalized type conversion relation $\simeq$ on descriptions corresponds to set equality on the underlying sets. Thus, $\tau \simeq \tau'$ iff $\tau \sqsubseteq \tau'$ and $\tau' \sqsubseteq \tau$.

An *expression* is well-formed with respect to a *type assignment* A : *Dvar* $\rightarrow$ *Kind* and a kind assignment B iff it has a *type* with respect to A and B. The type assignment maps ordinary variables to their types. The type inference rules are given below. They have been interleaved with the effect inference rules in order to illustrate the relation between types and effects.

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. If e has no free variables, we will simply write $e : \tau$ and $e \,!\, \epsilon$.

We write $FV(e)$ for the union of the free description variables and the free region constants of the expression e; likewise, we write $FV(\delta)$ for the free description variables and region constants of the description δ.

The sole type axiom states that the type of an ordinary variable is given by the type assignment; the effect axioms state that every ordinary variable and every value has effect PURE.

$$\begin{array}{c} A, B \vdash x : A(x) \\ x \,!\, \text{PURE} \\ v \,!\, \text{PURE} \end{array}$$

The types and effects of larger expressions are given by type and effect inference rules.

(Ordinary Abstraction)

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

(Ordinary Application)

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

(Polymorphic Abstraction)

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

(Polymorphic Application)

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

(Allocating and Initializing a Location)

$$\frac{\begin{array}{c} B \vdash \rho : \text{REGION} \\ B \vdash \tau : \text{TYPE} \\ A, B \vdash e : \tau' \ \wedge\ \tau' \sqsubseteq \tau \\ A, B \vdash e \,!\, \epsilon \end{array}}{\begin{array}{lll} A, B \vdash (\text{NEW } \rho\ \tau\ e) & : & (\text{REF } \rho\ \tau) \\ A, B \vdash (\text{NEW } \rho\ \tau\ e) & ! & (\text{MAXEFF } \epsilon\ (\text{ALLOC } \rho)) \end{array}}$$

(Reading a Location)

$$\frac{\begin{array}{c} A, B \vdash e : (\text{REF } \rho\ \tau) \\ A, B \vdash e \,!\, \epsilon \end{array}}{\begin{array}{lll} A, B \vdash (\text{GET } e) & : & \tau \\ A, B \vdash (\text{GET } e) & ! & (\text{MAXEFF } \epsilon\ (\text{READ } \rho)) \end{array}}$$

(Writing a Location)

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

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 immediate subexpressions that may be evaluated in order to evaluate the expression, and the *intrinsic* effect of the expression is the effect that is introduced by the expression itself.

5. Semantics

The soundness of the *MFX* effect inference rules can be expressed only in terms of its semantics. In this section we present a structured operational semantics of *MFX* that is expressed in terms of rewrite rules. These rules, which closely resemble the rewrite rules for the lambda-calculus, operate on tuples consisting of an expression, which models the computation that remains to be performed, and a store, which models the memory of the computation. The store maps locations to values. In order to deal with the locations that may arise during the course of computation, we expand the set of expressions to include locations.

Definition. A *store* is a finite function $\sigma : Loc \rightarrow Val$ that maps locations to values.

Definition. A *state* is a tuple $\langle e, \sigma \rangle$ of the form $(Exp \times Store)$.

Definition. A *terminal state* is a state $\langle v, \sigma \rangle$ of the form $(Val \times Store)$.

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. 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 $T(l_{\rho,\tau}) = \tau$. If ρ is a region constant, then the location $l_{\rho,\tau}$ belongs to the region corresponding to that region constant; otherwise, it belongs to the union of the corresponding regions. This situation reflects either *uncertainty indifference* about the region constant to which the location actually belongs.

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

Computation proceeds by repeatedly reducing the current state according to a set of reduction axioms and inference rules, until a terminal state results. The relation 'reduces to' or '$\stackrel{\text{red}}{\Longrightarrow}$' on $(\mathit{State} \times \mathit{State})$ gives the states, if any, to which a given state can be reduced. The reduction axioms are given below:

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

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

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

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

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

The reduction inference rules show how to reduce a state $\langle e, \sigma\rangle$ that does not match any reduction axiom by reducing a designated subexpression of e. The rules have been designed to ensure left-to-right, applicative order evaluation. We present only the rules for ordinary application; the remaining rules are similar.

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

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

A state is well-formed, $\mathcal{WF}_{\mathit{state}}(\theta)$, iff its expression component is well-formed, the contents of the locations that are bound in the store are well-formed and of a type that agrees with the tags of the corresponding locations, and every location that occurs in the state has the same tags everywhere and is bound in the store.

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

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

In order to express the effect soundness property, 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 and initialized 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 $\theta \stackrel{\text{red}}{\Longrightarrow} \theta'$

For the language defined thus far, these sets contain either zero or one location; however, we will not make use of this fact.

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

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

6. Effect Masking

Under certain circumstances, side-effects that cannot be observed outside of a given expression can be *masked* by the effect system. The rule for effect masking is developed in two steps. First, we present a construct for declaring *private regions* along with an effect masking rule for side-effects involving these private regions, and we demonstrate the soundness of this masking rule relative to our operational semantics. Second, we show how this rule can be modified to mask local side-effects even when private regions are not used. Effect masking allows imperative program fragments to be embedded in functional programs,

provided that these imperative fragments have functional semantics. Thus, effect masking allows functional programs to be implemented using imperative constructs. In addition, effect masking increases the feasibility of compile-time garbage collection in the presence of first-class subroutines and references.

The PRIVATE expression declares a private, anonymous region for local use that becomes inaccessible when the expression returns. Side-effects on this region cannot be observed outside of the expression, and need not be reported in the syntactic effect of the expression; we say that such effects can be *masked*. Since the locations belonging to a private region cannot be accessed after the expression returns, they can be safely deallocated when the expression returns.

The grammar clause for the PRIVATE expression is given below.

Exp =
- expressions

(PRIVATE *Dvar Exp*)
- declaration of a private region

The type and effect inference rule for the PRIVATE expression is given below. Note that 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. Effect masking is accomplished by substituting the empty region ψ for the private region variable d in the effect description ϵ; the empty region is defined such that any effect on ψ is interconvertible with PURE.

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

This rule resembles a composition of the rules for polymorphic abstraction and polymorphic application, except that (i) no actual region parameter is specified, (ii) the formal region parameter must not appear free in the type of the body, and (iii) any effects on the formal region parameter are masked.

The simplest way to define the semantics of the PRIVATE expression would be with the following reduction axiom, where r denotes a fresh region constant:

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

Unfortunately, this definition would invalidate the type preservation proposition: if the PRIVATE expression that masks the effects on the private region is removed, the effect of the expression may actually increase. Since type preservation is the foundation of our type and effect soundness propositions, we have developed a technique for reducing expressions such as the PRIVATE expression while retaining the type preservation property. Formally, the semantics of the PRIVATE expression are defined in terms of an *auxiliary expression*, which resembles the PRIVATE expression except that the bound variable has been replaced by a region constant. The auxiliary expression, *PRIVATE*, serves as a syntactic marker that indicates that effect masking is taking place. When a PRIVATE expression is reduced to a *PRIVATE* expression, a fresh region constant is chosen and embedded in the expression. The body of the *PRIVATE* expression is then reduced recursively, while the 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 *PRIVATE* expression can be reduced to its body.

The syntax of the *PRIVATE* expression is given by the type and effect inference rule shown below.

Exp =
- expressions

(*PRIVATE* *Dvar Exp*)
- PRIVATE in progress

In order to represent the fact that the region constant chosen in the reduction of a PRIVATE expression must be fresh, we extend the state of the computation with a third component (besides the expression and the store), namely a *region map* $\gamma : r \rightarrow \{\text{USED}\}$. States will now be of the form $\langle e, \sigma, \gamma \rangle$, and the existing reduction axioms and reduction inference rules are edited accordingly. (An alternative technique is given by Felleisen and Friedman [Fel87]).

The region map is used by the reduction axiom for PRIVATE expressions:

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

The resulting *PRIVATE* expression can be recursively reduced using the following reduction inference rule:

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

When the body has been reduced to a value, the *PRIVATE* expression can be eliminated using the following axiom:

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

Even though we have taken special care to retain a syntactic marker throughout the reduction process whenever effect masking is taking place, the effect soundness proposition no longer holds as originally stated, because it fails to take side-effects on private regions into account. Specifically, if a state $\theta \,!\, \epsilon$ 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.

We will now reformulate the effect soundness proposition so that it allows locations to be allocated, read, or written in regions that are private to a given expression even though these effects are not specified by the syntactic effect of the expression. The modified proposition operates by confining attention to the regions that are accessible in the *context* that surrounds the expression. This automatically excludes the private regions of the expression.

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.

Definition. A region constant r is *accessible* in a context 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 $r \in FV(\epsilon)$. We write $Acc(C)$ to denote the region constants that are accessible in the context C.

Using this notation, we can express the regions that are accessible in a given context C as simply $Acc(C)$. We can now revise the effect soundness proposition. It is expressed in a somewhat peculiar way in order to simplify a comparison with the original 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 or through the private regions of the expression. In other words, if $\theta = \langle C[e], \sigma, \gamma \rangle$, $\theta' = \langle C[e'], \sigma', \gamma' \rangle$ and $\theta \stackrel{red}{\Longrightarrow} \theta'$, and $e \,!\, \epsilon$ where

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

then

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

Whenever it is possible to wrap a given expression in a PRIVATE expression for a certain region without making the program in question ill-typed, the effects on that region can be masked as if this PRIVATE were actually present. The rule for this so-called *implicit* effect masking is derived directly from the rule for PRIVATE:

$$\frac{\begin{array}{c} A, B \vdash e : \tau \\ A, B \vdash e \,!\, \epsilon \\ B \vdash d : \text{REGION} \\ x \in FV(e) \Rightarrow d \notin FV(A(x)) \\ d \notin FV(\tau) \end{array}}{A, B \vdash e \,!\, \epsilon[\psi/d]}$$

Its soundness follows from the fact that there is a meaning-preserving source-to-source transformation that introduces the appropriate PRIVATE expression.

7. Experience with a prototype FX implementation

We have developed a prototype compiler for the *FX* programming language that uses effect information to constrain the parallel evaluation of expressions [Luc87]. The compiler enforces evaluation order constraints that guarantee serial evaluation semantics. The target language of the compiler is a dataflow graph that has been annotated with scheduling constraints; a dataflow graph representation was chosen because (i) the dataflow model allows us to express concurrency without considering such issues as processor allocation and computation grain size, and (ii) a parallel simulator for this model was available [Arv87, Tra86].

To date we have used the compiler and the simulator to check, compile and run a variety of programs, most of them fairly small. In general, the experimental results agree with our predictions.

Figures 1 and 2 show the result of an experiment that tested the value of effect masking on a small program that computes the sum of 1! to 10!. Figure 1 shows the execution of the program when it was compiled with effect masking disabled, while Figure 2 shows the execution of the program with effect masking enabled. Factorial was programmed in a conventional manner using an iterative method with side-effects; when effect masking was enabled, these local effects were masked, resulting in a functional specification for the factorial function, which in turn resulted in the concurrent evaluation of the iterations of the outer loop. The figures show the number of ALU operations the simulated dataflow machine executed during each instruction cycle. In this example, effect masking resulted in a substantial improvement in elapsed time. In general, of course, the results depend on the susceptibility of the program to the

particular kinds of concurrency analysis supported by the language.

8. Conclusions

We have presented a static effect system with effect and region polymorphism and effect masking. The type and effect information can be used to implement certain kinds of side-effect analysis that were not previously feasible in the presence of first-class subroutines and reference variables, particularly concurrency analysis and the the masking of local side-effects. Empirical results from a prototype implementation suggest that the effect system can be used by a parallel compiler.

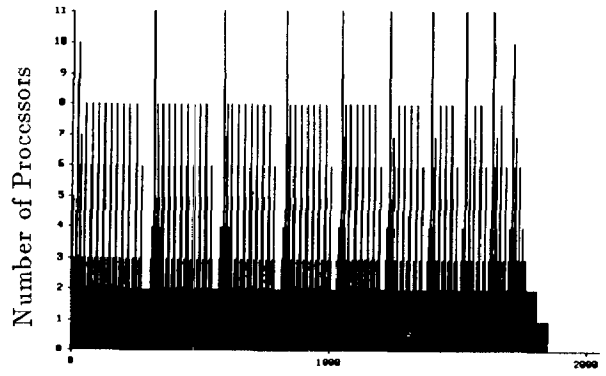

Figure 1: Execution without Masking

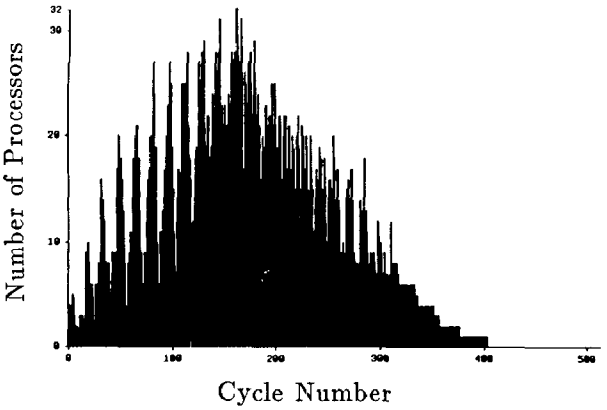

Figure 2: Execution with Masking

9. Bibliography

Arv87 *Executing a Program on the MIT Tagged-Token Dataflow Architecture*, Arvind, 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

Bar78 *A Practical Interprocedural Data Flow Analysis algorithm*, Jeffrey M. Barth, Communications of the ACM, Vol. 21, No. 9 (September 1978), pp. 724–736

Gif86 *Integrating Functional and Imperative Programming*, David K. Gifford, 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-409, MIT Laboratory for Computer Science, September 1987.

Fel87 *A Calculus for Assignments in Higher-Order Languages*, Matthias Felleisen, Daniel Friedman, Fourteenth Annual ACM Symposium on Principles of Programming Languages (January 1987), pp. 314–325

Lam77 *Report on the Programming Language Euclid*, Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, Gerald J. Popek, SIGPLAN Notices, 12 (1977), pp. 1–79

Luc87 *Types and Effects — Towards the Integration of Functional and Imperative Programming*, John M. Lucassen, Ph. D. Thesis, MIT Laboratory for Computer Science LCS TR-408 (August 1987)

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)

Nei87 *Computation of Aliases and Support Sets*, Anne Neirynck, Prakash Panagaden, Alan J. Demers, 14th Annual ACM Symp. on Principles of Programming Languages, Munich, West Germany, January 21-23, 1987, pp. 274-283.

Pop77 *Notes on the Design of Euclid*, G. J. Popek, J. J. Horning, 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

Rey74 *Towards a Theory of Type Structure*, International Programming Symposium, Lecture Notes in Computer Science no. 19 (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

Ste78 *Rabbit: A Compiler for Scheme (A Study in Compiler Optimization)*, AI-TR-474, MIT AI Laboratory, May 1978.

Tra86 *A Compiler for the MIT Tagged-Token Dataflow Architecture*, Kenneth R. Traub, S.M. Thesis, MIT Laboratory for Computer Science (August, 1987)

Wei80 *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables*, Seventh Annual ACM Symposium on Principles of Programming Languages (January 1980), pp. 83–94