Integrating Functional and Imperative Programming

by
David K. Gifford and John M. Lucassen

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Abstract

We present a class of programming languages that enables the advantages of functional and imperative computation to be combined within a single program. These languages, which we call *fluent* languages, have distinct sublanguages for functional and imperative programming. Sublanguage invariants are verified by a static checking system that simultaneously determines the *type* and the *effect class* of every expression. *Effect checking* is similar to type checking, but it is used to guarantee side-effect invariants instead of value invariants. Effect checking also makes it possible to implement polymorphism in a general, type-safe and efficient manner despite the presence of side-effects. Preliminary simulation results suggest that fluent programs are well suited for parallel processing.

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

1. Introduction

Functional languages are widely recognized for their expressive power and straightforward semantics. Their mathematical simplicity makes them a suitable vehicle for formal analysis and program verification. Moreover, the semantics of a functional language typically permit many different implementations, including normal order, applicative order, lazy, and eager evaluation and memoization [Abelson85]. These advantages have motivated a great deal of research into functional languages [Backus78], functional programming [Henderson80], and the design of computers that can execute functional programs rapidly and efficiently [Gurd85].

Imperative languages, on the other hand, retain certain advantages over functional languages. First, some programming tasks, for example input and output, are most naturally viewed in terms of operations with side-effects. Second, the implementations of most imperative languages are still more efficient than those of functional languages. [Morris82] and [Henderson80] discuss the relative merits of functional and imperative languages.

This paper describes a class of programming languages that offers the benefits of both functional and imperative programming by permitting programmers to mix functional and imperative computation in a single program. We call a programming language that directly supports both functional and imperative programming a *fluent* language.

One of the premises behind fluent languages is that side-effect information is important enough to be declared as part of the *type* of a program. Such side-effect declarations document a programmer's intent and

provide valuable information for optimization and parallel execution. Fluent languages support this programming methodology by verifying the type and side-effect assertions provided by the programmer.

In a fluent language side-effects are described in terms of *effect classes*. Every expression has an effect class; just as the *type* of an expression describes the *value* computed by the expression, the *effect class* of an expression describes *how* that value is computed. Effect classes will be defined in detail below. Examples of effect classes include "OBSERVER" and "FUNCTION": an expression that is an OBSERVER can observe side-effects, but it can not cause them, while a FUNCTION can neither cause nor observe side-effects.

The effect class of an expression determines the sublanguage to which the expression must be confined, which in turn determines which language facilities the expression may use and what subroutines it may call. At the same time, the side-effect specification of a subroutine determines which sublanguages may call it.

Restricted approaches to mixing functional and imperative programming have previously been suggested as a way to mitigate some of the shortcomings of functional programming, in particular the difficulties with persistent objects and input/output. For example, Backus suggested the notion of Applicative State Transition Systems [Backus78]. In an AST system all computation must occur in the functional language; the imperative part of a program consists of transition rules that are used to select functions that update the system state.

The designers of the programming language Euclid [Lampson78] also recognized the power of mixing imperative and functional programming. In Euclid, procedures are executed solely for their side-effects (they return no result value) and functions are executed solely for their result value (they can cause no side-effects). This strict separation between procedures and functions is intended to simplify program correctness proofs. However Euclid's statement oriented style and its lack of higher order functions and procedures limit its use for constructing substantial functional program components.

A third approach for mixing functional and imperative computation is to employ an inference system to determine which portions of an imperative program are functional. This approach has been used to decompose imperative programs for parallel processing [Marti83]. Effect classification systems that are based exclusively on inference have three limitations:

- *Expressive Power* — Fully automatic, 'hidden' effect inference leaves the programmer with no way to document or verify *intended* side-effect invariants, such as referential transparency. In the absence of documentation or verification small changes to a program may result in a dramatic change in performance.
- *Separate Compilation* — Separate compilation forces the inference system to make worst-case assumptions about external subroutines. The inductive nature of side-effect inferences tends to cause suboptimal side-effect classifications to propagate throughout the program.
- *Subroutine Variables* — It is difficult to determine the side-effects of calls on subroutine variables. In cases where data flow analysis fails to identify the subroutine(s) being called, the side-effect inference system must assume the worst.

Another viewpoint on the latter two limitations is that effect classification systems based exclusively on inference are sometimes unable to compute a tight bound on the effect class of an expression without additional information. It is precisely this additional information that fluent languages are designed to provide.

Unlike the three other approaches presented above (AST systems, Euclid and effect inference), fluent languages are designed to support a programming methodology in which programmers can freely mix functional and imperative programming. The proper mix is determined by the requirements of the application. A program written in a fluent language can be written in a completely functional style, in a completely imperative style, or in any combination of styles.

Our discussion of fluent languages is organized into the following topics:

- Section 2 — effect classes;
- Section 3 — a specific fluent language;
- Section 4 — effect checking;
- Section 5 — applications to polymorphism;
- Section 6 — implementation issues;
- Section 7 — simulation results; and
- Section 8 — conclusions and future work.

2. Effect Classes

Our goal is to define a programming language that permits functional and imperative computation to be mixed yet retains the benefits of both programming models. In particular, we are interested in retaining the potential for concurrent evaluation and memoization of functional subprograms. In order to determine whether two expressions can be executed concurrently without altering their semantics, one must determine whether the expressions have side-effects that might cause *interference*: by definition, two expressions interfere if concurrent execution may change their semantics because of store interactions. In order to determine whether an expression can be memoized, one must determine whether the expression is *referentially transparent*: by definition, an expression is referentially transparent if it has no side-effects and returns the same value whenever it is evaluated. In the language presented in this paper, information about interference and referential transparency is inferred from three orthogonal properties:

- the ability to *allocate* and initialize memory locations whose value may be changed,
- the ability to *read* the contents of memory locations whose value may be changed (*i.e.* to *observe* side-effects), and
- the ability to *write* new values to existing memory locations (*i.e.* to *cause* side-effects).

In general it is undecidable whether an expression will allocate, read or write memory locations whose value may be changed. Effect checking is *conservative*, and classifies an expression as having each of the three properties unless the opposite can be shown.

We will use the letters W, R and A (for *Write*, *Read* and *Allocate*) to refer to the three properties. The eight possible combinations of these properties form the Boolean lattice depicted in Figure 2-1. Although all eight combinations of properties are possible, they are not all *distinct* in terms of their implications for interference and referential transparency. We will now describe how the eight combinations can be mapped into four distinct *effect classes*.

First, consider referential transparency. The A property affects the referential transparency of an expression but has no effect on interference between expressions. Since an expression with the W or R property is not referentially transparent it does not matter whether such an expression also has the A property. In other words, A only matters in the absence of W and R, and the following combinations can be joined into classes without loss of information: WR and WRA, W and WA, and R and RA (see Figure 2-2).

Next, consider interference between expressions. Only one kind of interference exists: an expression with the property W interferes with expressions with W or R. It follows that for an expression that has property W the inclusion of R is irrelevant. Consequently, all combinations containing W (W, WR, WA and WRA) are equivalent, and can be joined into a single class. The lattice that results is depicted in Figure 2-3.

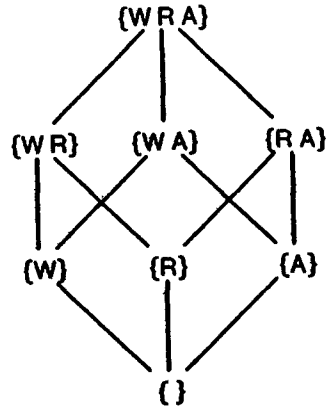

Figure 2-1: Possible combinations of W, R and A

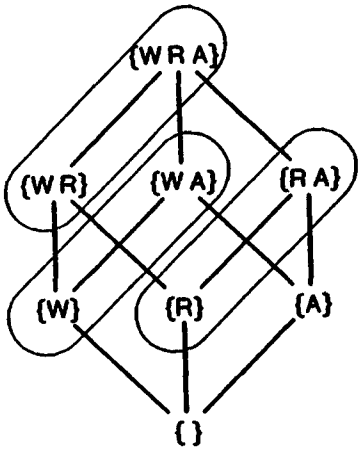

Figure 2-2: A only matters in the absence of W and R

We adopt the following names for the various classes of combinations of properties:

W, WR, WA, WRA	— PROCEDURE
R, RA	— OBSERVER
A	— FUNCTION
∅	— PURE (*i.e.* pure function)

Note that the lattice depicted in Figure 2-3 imposes a total order on the effect classes. This is not true in general; it is a consequence of our objective (to identify opportunities for concurrent execution and memoization) and of our choice of properties. As an example, consider *idempotent* expressions. An expression is idempotent if it has the same effect and returns equivalent values the expression it is evaluated repeatedly instead of just once. If our goal were to identify idempotent expressions, then we should distinguish between the class {W, WA} ('mutators' or 'write-only' expressions, which are idempotent) and the class {WR, WRA} ('read/write' expressions, which are not idempotent). This would result in a set of effect classes that is not totally ordered, as depicted in Figure 2-4.

If on the other hand our goal were to exploit memoization as much as possible, then we should distinguish between the classes {RA} ('observers') and {R} ('pure observers'). Expressions with effect class PURE OBSERVER have the property that they may be memoized for the duration of any time interval during which no expressions with effect class PROCEDURE are evaluated. This is the case because as long as there are no side effects, the ability of a PURE OBSERVER to *observe* side effects is irrelevant. The resulting effect class lattice is depicted in Figure 2-5.

Finally, the effect lattices presented here can be generalized by augmenting effect classes with information indentifying the locations which are read or written. Many different ways exist to identify locations, including:

- variable names
- collections [Lampson78]
- the fields of certain data structures (such as CAR and CDR) [Steele78]
- the values associated with a given indicator on the property lists of any identifiers [Marti80]

In general the more detailed the side-effect classes, the more accurately one can identify possible concurrency in a program. On the other hand excessive detail in the effect specifications may make programs unwieldy and hard to understand. For the present work a simple classification was chosen to keep side-effect specifications simple and straightforward.

In summary, we classify expressions and subroutines in terms of three independent side-effect properties: *allocate*, *read* and *write*. Consideration of our goal (to identify opportunities for concurrent execution and memoization) leads us to group the eight possible combinations of these properties into four equivalence classes, which we call PROCEDURE, OBSERVER, FUNCTION, and PURE.

3. A Simple Fluent Language

This section presents a simple fluent language and its type system. The language has four distinct sublanguages which correspond to the effect classes introduced in the previous section. As we shall see each sublanguage has its own advantages and limitations.

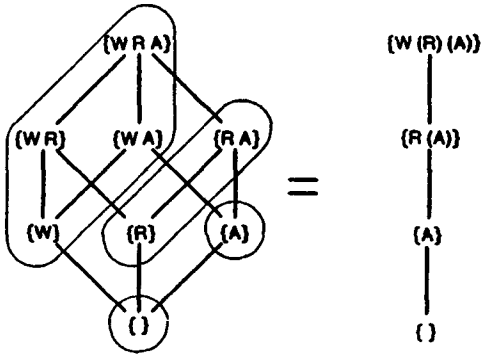

Figure 2-3: Combinations containing W are equivalent

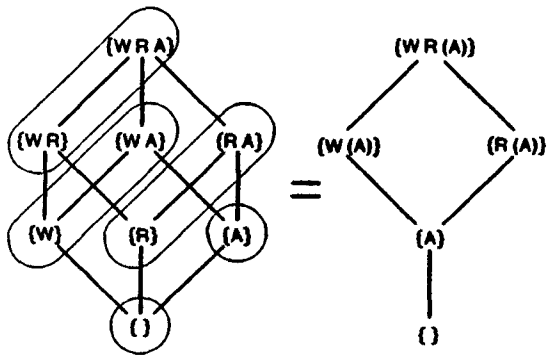

Figure 2-4: Effects need not be totally ordered

The grammar for the language is shown in Figure 3-1. E ranges over program expressions, T ranges over type expressions, and C ranges over effect classes.

The language features constants and variables, assignment, conditional evaluation, ordinary lambda abstraction and application, and type abstraction and application. Subroutines are constructed with the **lambda** expression. The effect class of the body of a **lambda** is reflected in its type: there are four arrow types $\rightarrow_C$, one for each type of subroutine. The expression **the** asserts that a given expression has a specified effect class and type, and may be used to declare the effect class of the body of a **lambda**. Polymorphic values are constructed with the **tlambda** expression which provides type abstraction.

Every expression in E has a type and an effect class. The effect class of an expression indicates which (if any) of the following *sublanguage invariants* holds for the expression:

- an OBSERVER does not cause side-effects;
- a FUNCTION does not cause side-effects, and its value is not affected by side-effects;
- a PURE is referentially transparent: it does not cause side-effects, its value is not affected by side-effects, and it returns the same value each time it is evaluated.

There is no sublanguage invariant for the effect class PROCEDURE: a PROCEDURE may cause and/or observe side-effects, and may return different values at different times.

In order to guarantee the sublanguage invariants, a fluent language imposes certain *sublanguage constraints*. Programs must obey these constraints just as they must be well-typed. The sublanguage constraints determine the kinds of expressions which may appear in each of the four sublanguages:

- a PROCEDURE is not subject to any restrictions; it may use all the facilities of the language;
- an OBSERVER may not call a PROCEDURE, and may not perform assignments;
- a FUNCTION may not call a PROCEDURE or OBSERVER, may not perform assignments, and may not access variables that can be updated;
- A PURE may not call a PROCEDURE, OBSERVER or FUNCTION, may not perform assignments, may not access variables that can be updated, and may not allocate variables that can be updated.

The sublanguage constraints presented here are fairly conservative. For example a subroutine containing assignments to local variables cannot be declared a FUNCTION, even though the subroutine may behave functionally. Side-effect classification could be made less

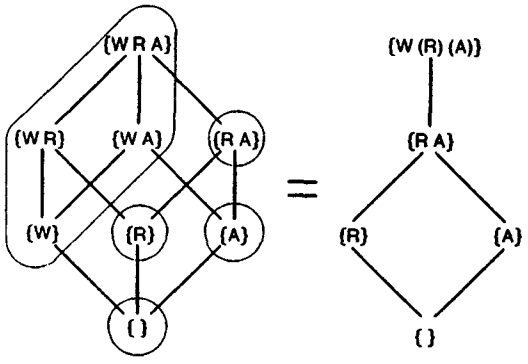

Figure 2-5: Effect Class Lattice with Pure Observers

Expressions

E ::=	
v	constant or variable
(**set!** v E)	assignment
(**let** (v E_1) E_2)	E_2 is evaluated with v bound to a location initialized with the value of E_1
(**if** E_1 **then** E_2 **else** E_3)	conditional evaluation
(**the** C T E)	effect and type assertion
(**lambda** (v:T) E)	ordinary lambda abstraction
(E_1 E_2)	ordinary application
(**tlambda** t E)	type abstraction
(E T)	type application

Types

T ::=	
t	type variable (such as **bool**)
($T_1 \rightarrow_C T_2$)	type of subroutines
($\forall$ t T)	type of polymorphic values

Effect Classes

C ::=

PROCEDURE | OBSERVER | FUNCTION | PURE

Figure 3-1: Syntax of a Simple Fluent Language

conservative at the expense of complexity, as pointed out by Reynolds in connection with a similar problem [Reynolds78]. The set of sublanguage constraints presented in this paper was chosen for simplicity. The constraints are easy to describe and easy to enforce and yet permit an interesting set of effect classes to be distinguished.

Note that the primitive operations of immutable types (such as numbers and streams) are generally PURE, whereas the primitive operations of mutable types (such as arrays) may have any effect class. For example, (fixed length) arrays can be created by a FUNCTION, updated by a PROCEDURE, and accessed by an OBSERVER, and their length can be determined by a PURE. When mutable values are passed as arguments to a FUNCTION or a PURE, the sublanguage constraints ensure that the resulting computation does not depend on any changeable attributes of the arguments.

The language presented in this paper is a subset of a larger fluent language that is currently under development. This larger language has a polymorphic type system [Reynolds74], permits abstraction over effect classes, and uses bounded quantification [Cardelli86] to take advantage of the ordering on effect classes.

4. Effect Checking

In order to guarantee that the sublanguage invariants hold it is necessary to enforce the sublanguage constraints. We call this verification process *effect checking*. Effect checking can be performed dynamically or statically.

In order to perform effect checking dynamically, the language implementation must keep track of the effect class of the subroutine that is executing, and if the subroutine is not a PROCEDURE the following precautions must be taken:

- Whenever a subroutine is invoked the implementation must ensure that its effect class is less than or equal to the effect class of the subroutine running at the time;
- If the current subroutine is an OBSERVER, a FUNCTION, or a PURE, the implementation must ensure that it does not perform any assignments;
- If the current subroutine is a FUNCTION or a PURE, the implementation must also ensure that the subroutine does not access any variables that can be updated; and
- If the current subroutine is PURE, the implementation must finally ensure that the subroutine does not allocate memory locations whose value may later be changed.

Although dynamic effect checking may be appropriate for certain hardware environments and for interpretive execution, this form of checking may involve substantial runtime overhead. Static checking on the other hand combines the advantage of early error reporting with the absence of runtime overhead. In addition, the static determination of effect class information provides a compiler with valuable information for optimization, parallel execution and memoization. For this reason the fluent language described in this paper has been designed specifically to permit static effect class inference.

Because of the similarity between type checking and effect checking, these two tasks have been integrated into a single set of type and effect inference rules. These inference rules indicate how the type and the effect class of an expression can be inferred from the types and the effect classes of its subexpressions. One inference rule exists for each kind of expression in the language.

In a fluent language the type of a subroutine reflects the effects of its body. In situations where the effect class of a subroutine does not matter to the programmer these effect specifications may interfere with the reusability of code. In order to minimize the adverse impact of over-specification, the type system takes advantage of the ordering on effect classes. For example, FUNCTION is a subclass of OBSERVER, and consequently, a FUNCTION type is a subtype of the corresponding OBSERVER type, and may appear wherever such a OBSERVER is expected.

The partial function $Type(A)[\![E]\!]$ maps an extended type assignment (A) and an expression (E) into a type; the partial function *Effect* maps an extended type assignment and an expression into an effect class.

Notation:

- The inference rules are expressed in terms of an extended type assignment A: a partial function that maps an identifier into a pair consisting of its type and the token CONST or VAR.
- $A[x\cdot\langle T,$ CONST/VAR$\rangle$ is the extension of A that maps the identifier x to the tuple $\langle T,$ CONST/VAR$\rangle$.
- $A{-}t$ is the restriction of A obtained by removing

from its domain any identifier x mapped into a tuple $\langle T$, CONST/VAR$\rangle$ such that t is free in T.

- $T_1 \subseteq T_2$ means that T_1 is a subtype of T_2; likewise, $C_1 \subseteq C_2$ means that C_1 is an effect that is less than E_2 in the partial order on effects.

The inference rules are presented in the following form:

$$\frac{premise_1,\ premise_2, \ldots}{conclusion_1,\ conclusion_2, \ldots}$$

The meaning of this construct is as follows: "if the premises ($premise_1$, $premise_2$, ...) are satisfied, then we may infer the conclusions ($conclusion_1$, $conclusion_2$, ...)".

- The inference rules for constants and variables are straightforward: a constant is PURE and a variable is an OBSERVER.

$$\frac{A(v) = \langle T, \text{CONST}\rangle}{\begin{array}{l} Type(A)[\![\ v\ ]\!] = T \\ Effect(A)[\![\ v\ ]\!] = \text{PURE}\end{array}}$$

$$\frac{A(v) = \langle T, \text{VAR}\rangle}{\begin{array}{l} Type(A)[\![\ v\ ]\!] = T \\ Effect(A)[\![\ v\ ]\!] = \text{OBSERVER}\end{array}}$$

- The inference rule for assignment is also straightforward: an assignment causes a side-effect, so it has effect class PROCEDURE. Note that v must be a variable.

$$\frac{\begin{array}{c} A(v) = \langle T, \text{VAR}\rangle \\ Type(A)[\![\ E\ ]\!] \subseteq T\end{array}}{\begin{array}{l} Type(A)[\![\ (\textbf{set!}\ v\ E)\ ]\!] = T \\ Effect(A)[\![\ (\textbf{set!}\ v\ E)\ ]\!] = \text{PROCEDURE}\end{array}}$$

- The effect of a **let**-expression is obtained by combining the effect of evaluating the binding expression (C_1), the effect of allocating a location for the variable v (FUNCTION), and the effect of evaluating the body (C_2).

$$\frac{\begin{array}{c} Type(A)[\![\ E_1\ ]\!] = T_1,\ Effect(A)[\![\ E_1\ ]\!] = C_1 \\ Type(A[\ v{:}\langle T_1, \text{VAR}\rangle\ ])[\![\ E_2\ ]\!] = T_2 \\ Effect(A[\ v{:}\langle T_1, \text{VAR}\rangle\ ])[\![\ E_2\ ]\!] = C_2\end{array}}{\begin{array}{l} Type(A)[\![\ (\textbf{let}\ (v\ E_1)\ E_2)\ ]\!] = T_2 \\ Effect(A)[\![\ (\textbf{let}\ (v\ E_1)\ E_2)\ ]\!] = \\ \qquad max(C_1, \text{FUNCTION}, C_2)\end{array}}$$

- The effect of a conditional expression is obtained by combining the effect of the predicate (C_1), the effect of the consequent (C_2) and the effect of the alternative (C_3). Note that the predicate must have type **bool** and that the consequent and alternative must have the same type.

$$\frac{\begin{array}{c} Type(A)[\![\ E_1\ ]\!] = \textbf{bool},\ Effect(A)[\![\ E_1\ ]\!] = C_1 \\ Type(A)[\![\ E_2\ ]\!] = T,\ Effect(A)[\![\ E_2\ ]\!] = C_2 \\ Type(A)[\![\ E_3\ ]\!] = T,\ Effect(A)[\![\ E_3\ ]\!] = C_3\end{array}}{\begin{array}{l} Type(A)[\![\ (\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3)\ ]\!] = T \\ Effect(A)[\![\ (\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3)\ ]\!] = \\ \qquad max(C_1, C_2, C_3)\end{array}}$$

- A effect and type specification expression is well-formed provided that the *actual* type and effect of the body are compatible with the *specified* type and effect:

$$\frac{\begin{array}{c} Type(A)[\![\ E\ ]\!] \subseteq T \\ Effect(A)[\![\ E\ ]\!] \subseteq C\end{array}}{\begin{array}{l} Type(A)[\![\ (\textbf{the}\ C\ T\ E)\ ]\!] = T \\ Effect(A)[\![\ (\textbf{the}\ C\ T\ E)\ ]\!] = C\end{array}}$$

- A **lambda**-expression is PURE, regardless of the effects of its body, because the body is not executed until later. The effect of the body (C) does appear in the *type* of the expression, so that the effect can be taken into account when the resulting subroutine is called. Within the subroutine body v is a constant.

$$\frac{\begin{array}{c} Type(A[\ v{:}\langle T, \text{CONST}\rangle\ ])[\![\ E\ ]\!] = T_1 \\ Effect(A[\ v{:}\langle T, \text{CONST}\rangle\ ])[\![\ E\ ]\!] = C\end{array}}{\begin{array}{l} Type(A)[\![\ (\textbf{lambda}\ (v{:}T)\ E)\ ]\!] = (T \rightarrow_C T_1) \\ Effect(A)[\![\ (\textbf{lambda}\ (v{:}T)\ E)\ ]\!] = \text{PURE}\end{array}}$$

- The effect class of an application is obtained by combining the effect of the operator expression (C_1), the effect of the argument expression (C_2), and the latent effect of the subroutine being called (C).

$$\frac{\begin{array}{c} Type(A)[\![\ E_1\ ]\!] = (T_2 \rightarrow_C T),\ Effect(A)[\![\ E_1\ ]\!] = C_1 \\ Type(A)[\![\ E_2\ ]\!] \subseteq T_2,\ Effect(A)[\![\ E_2\ ]\!] = C_2\end{array}}{\begin{array}{l} Type(A)[\![\ (E_1\ E_2)\ ]\!] = T \\ Effect(A)[\![\ (E_1\ E_2)\ ]\!] = max(C_1, C_2, C)\end{array}}$$

- A **tlambda**-expression is PURE just like an ordinary **lambda**-expression. The effect of the body need not appear in the type of the resulting polymorphic value because the body is required to be PURE.

$$\frac{\begin{array}{c}Type(A{-}t)[\![\ E\ ]\!] = T \\ Effect(A{-}t)[\![\ E\ ]\!] = \text{PURE}\end{array}}{\begin{array}{l}Type(A)[\![\ (\textbf{tlambda}\ t\ E)\ ]\!] = (\forall\ t\ T) \\ Effect(A)[\![\ (\textbf{tlambda}\ t\ E)\ ]\!] = \text{PURE}\end{array}}$$

- The body of a **tlambda**-expression is guaranteed to be PURE so the effect class of a type application is simply the effect of the operator expression (C).

$$\frac{\begin{array}{c}Type(A)[\![\ E\ ]\!] = (\forall\ t\ T) \\ Effect(A)[\![\ E\ ]\!] = C\end{array}}{\begin{array}{l}Type(A)[\![\ (E\ T_1)\ ]\!] = T[\ T_1/t\ ] \\ Effect(A)[\![\ (E\ T_1)\ ]\!] = C\end{array}}$$

5. Polymorphism and Side-effects

Fluent languages admit polymorphic values in the presence of side-effects in a simple, type safe and general way. Furthermore, they permit type application to be implemented with no run-time cost despite the presence of side-effects. Fluent languages accomplish this by insisting that polymorphic values be created with **pure** expressions.

The polymorphic type system of the fluent language presented in this paper is based on Reynolds' second-order lambda-calculus [Reynolds74]. Our language differs from the lambda-calculus in that it permits side-effects; our type system however is identical to that of the lambda-calculus. Consequently our language permits static type checking and has the property that every well-typed program corresponds to an equivalent program in the corresponding untyped language. This untyped program is generally more efficient than the original program because any computation that is devoted solely to type manipulation can be omitted. A set of *type erasure* rules indicates how to map a well-typed program into an equivalent program in the corresponding untyped language. For the second-order lambda-calculus the rules for type application and projection are as follows (see [Mitchell85]):

$$(\textbf{tlambda}\ t\ E) \Rightarrow E \qquad (1a)$$
$$(E\ T) \Rightarrow E \qquad (1b)$$

These erasure rules are unsound in the presence of side-effects because they create a type loophole involving polymorphic 'own' variables (as illustrated in [Gordon79], pp. 51-53). In a language with assignment each polymorphic variable must have a distinct instance for each interpretation of its free type variables. One way of accomplishing this is to have type application create a new environment for the creation of variables using the following rewrite rules:

$$(\textbf{tlambda}\ t\ E) \Rightarrow (\textbf{lambda}\ ()\ E) \qquad (2a)$$
$$(E\ T) \Rightarrow (E) \qquad (2b)$$

These type erasure rules guarantee a type safe implementation. Unfortunately, this formulation causes type application to incur a run-time cost. Other imperative polymorphic languages, including ML, CLU and Ada, are designed so that type application does not require any computation. In the case of these other languages certain concessions were made to achieve this goal:

- ML [Gordon79] ensures type safety by prohibiting assignment to global polymorphic references, which prevents subroutines from sharing mutable polymorhic references. Standard ML [Milner84] prohibits polymorphic references altogether. In addition polymorphic values are not first-class values in either version of ML.
- CLU [Liskov79] and Ada [Ada80] ensure type safety by replicating polymorphic modules and their variables for each interpretation of the bound type variables. Modules must be declared at top level and are not first-class values. In CLU, references to polymorphic values also are not referentially transparent because the first reference may trigger initialization, which may result in observable side-effects.

The approach that we have adopted for fluent languages is that the body E of a polymorphic value definition (**tlambda** t E) must have effect class PURE. Consequently, the semantics of type erasure rules (1a,b) and (2a,b) are the same, and an efficient implementation based on rules (1a,b) can be used. The result is a type-safe implementation of polymorphism in which type abstraction and application do not require any computation.

Unlike the approaches adopted by ML, CLU, and ADA, the proposed restriction for fluent languages does not result in a loss of expressive power. In a fluent langauge several subroutines can share and update polymorphic variables. Such subroutines would be contained in the body E of a **tlambda**.

6. Implementation

Since a fluent language is a hybrid of an imperative language and a functional language different implementation techniques are appropriate for its different sublanguages.

The FUNCTION and PURE sublanguages are "conventional" functional programming languages: because there are no side-effects, evaluation order is constrained only by data flow. Like other functional programming languages the FUNCTION and PURE sublanguages encourage a programming style in which the potential concurrency in a computation is easily identified.

Subroutines written in the OBSERVER sublanguage are capable of observing side-effects. However, the language definition guarantees that no side-effects are visible during the evaluation of an OBSERVER subroutine. It follows that for the purpose of implementation the OBSERVER sublanguage is exactly like the FUNCTION sublanguage: evaluation order is constrained only by data flow, and the same implementation techniques are appropriate. Note that 'pure observer' expressions, if recognized by the language, may be memoized as long as the computation remains provably in the OBSERVER domain.

In the PROCEDURE sublanguage subexpressions may have side-effects which constrain their evaluation order. Nevertheless, programs written in this sublanguage may contain opportunities for concurrent execution and memoization that can be identified statically by examining the side-effect classifications of the programs' subexpressions. For implementation purposes, we regard a PROCEDURE as a FUNCTION augmented with some additional constraints on evaluation order. Below, we will describe how to construct, for each subroutine, an explicit side-effect constraint graph which represents all the constraints on evaluation order which are not implicit in the data dependencies.

In order to compute the side-effect constraint graph of a subroutine proceed as follows. Consider the subexpressions in evaluation order (for conditional evaluation consider all possible evaluation orders). Ignore FUNCTION and PURE expressions, since they do not cause or observe side-effects, and focus on the remaining subsequence of PROCEDURE and OBSERVER expressions.

In order to construct the side-effect constraint graph one could simply draw a dependency arc from each OBSERVER node to all of the following PROCEDURE nodes, and from each PROCEDURE node to all the following PROCEDURE and OBSERVER nodes. However, there is a more compact representation. All effect constraint graphs with the same transitive closure contain the same information. In fact, every program has a unique *minimal* effect constraint graph which can be constructed directly from the expression sequence. The proofs of minimality and uniqueness, as well as the method by which the graph is constructed, are all based on the fact that the subgraph containing only the PROCEDURE nodes of the graph corresponds to a total order. Because the PROCEDURE nodes are totally ordered, each OBSERVER can interfere only with directly preceeding and following PROCEDURE nodes. Similarly, each PROCEDURE can only interfere with the PROCEDURE or the group of OBSERVER nodes that directly preceed or follow the PROCEDURE node. Consequently, it suffices to draw a dependency arc from each OBSERVER to the PROCEDURE immediately following it (if any), and from each PROCEDURE to the directly following PROCEDURE or group of OBSERVER nodes.

Although the side-effect constraint graph is constructed without redundant arcs some of these arcs are redundant when data flow constraints are taken into account. This is illustrated in Figure 6-1 for the sample program

```
(p1 (set! x y) (p2 x))
```

where `p1` and `p2` are procedure constants and `x` and `y` are integer variables. In this example the side-effect constraint between evaluation of `y` and the assignment to `x` is redundant, because the assignment needs the value returned by `y`; likewise the constraint between the invocation of `p2` and the invocation of `p1` is redundant because the latter needs the value returned by the former.

The redundant arcs of the side-effect constraint graph can be removed as follows. Compute the *data flow* constraint graph for the program as if there were no side-effects, and merge the graph with the *side-effect* constraint graph. Next, find the minimal spanning relation of this graph, *i.e.* the minimal set of edges that has the same transitive closure. Finally, subtract the data flow constraint graph from the result, to obtain the

set of non-redundant side-effect constraints. The data flow constraint graph for the program of Figure 6-1 is shown in Figure 6-2; the remaining non-redundant side-effect constraint is shown in figure 6-3.

7. Performance

The language described in this paper is a subset of a larger fluent language that is currently under development. We have implemented a type and effect checker for this larger language, as well as several simulators, each representing a particular trade-off between concurrency and complexity. In general we find that the portions of a fluent program that are written in the OBSERVER, FUNCTION and PURE sublanguages are able to utilize multiple processors to increase throughput.

As an illustration of the performance gains which can be achieved, consider matrix multiplication. We have implemented a matrix multiplication benchmark in which matrices are represented as arrays, which are mutable objects. Functional languages have severe difficulties dealing with arrays efficiently. In a *fluent* language on the other hand arrays created by an imperative subprogram can be passed to and manipulated by functional subprograms. The matrix multiplication utility is implemented as an OBSERVER and substantial concurrency can be realized. Simulation confirms that whereas the number of elementary operations is a cubic in the array size (S=$114n^3$+$66n^2$+$146n$+159), execution time is linear in the array size (P=$134n$+96).

The current simulator simulates a hypothetical shared-memory architecture in which FORK and JOIN can be performed in zero time and in which there are always more processors than tasks. A more realistic simulator is currently under construction.

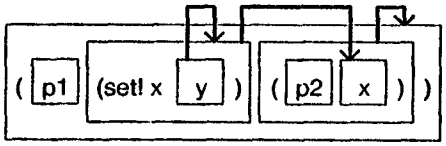

Figure 6-1: Example of Side-effect Constraints

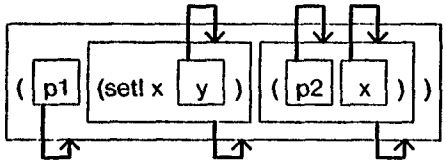

Figure 6-2: Example of Data Flow Constraints

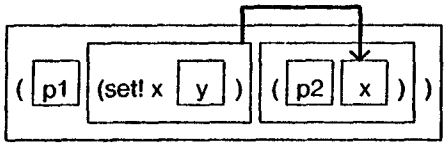

Figure 6-3: Non-redundant Side-Effect Constraints

8. Conclusions and Future Work

Fluent languages provide a single model of computation that encompasses both functional and imperative programming. In a fluent programming language programmers may freely mix functional and imperative programming, subject to the sublanguage constraints. These constraints, which restrict the permissible side-effects of expressions, are enforced by a process similar to type checking.

Currently one disadvantage of fluent languages is the need for explicit effect class declarations. Explicit declarations are used to simplify static effect checking. We are investigating techniques, akin to type inference, which may reduce the burden placed on the programmer.

A possible second disadvantage follows from our insistence that all effect specifications should be verifiable. Since the problem of optimal effect classification is undecidable, it is inevitable that any effect checking system will rule out certain correct programs. This problem can be solved only by permitting 'effect loopholes', *i.e.* by accepting programs that cannot be proven to satisfy their own specifications.

A final possible disadvantage concerns the impact of mandatory verifiable specifications on software reusability. Type and effect polymorphism (not discussed here) go a long way towards making specifications as flexible as possible. In addition, we point out that a fluent language contains both a functional and an

imperative subset — programmers are free to confine themselves to either subset, in which case they need not deal with the effect system at all. The benefits of a fluent language may be obtained, by those who desire to do so, at the expense of somewhat more careful program design.

The primary advantage of a fluent language is that functional and imperative programming can be mixed within a single program, while the benefits of both kinds of programming are retained:

- the ability to program in the imperative language when its programming model is more natural or permits a more efficient implementation;
- the ability to program in a functional language when its simplicity and implicit parallelism are desired;
- the ability to have subroutines which are functions of mutable values and yet can be evaluated concurrently with other ongoing computation;
- the identification of expressions which are referentially transparent and can therefore be memoized;
- documentation in the form of effect class specifications, and the enforcement of these specifications with an effect checking system;
- performance improvements because of parallel execution and improved optimization.

This set of benefits has not been realized in previous programming models, and we believe that our proposal will find value in many applications.

References

[Abelson85] Harold Abelson, Gerald Jay Sussman, Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press/McGraw Hill (1985)

[Ada80] *Reference Manual for the Ada Programming Language*, U.S. Dept. of Defense, GPO 008-000-0035408 (1980)

[Backus78] J. Backus, *Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, Vol. 21, No. 8 (August 1978), pp. 613-641

[Cardelli86] Luca Cardelli, Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, to appear in Computing Survey

[Gordon79] Michael J. C. Gordon, Robin Milner, Christopher Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science no. 78 (1979)

[Gurd85] J. R. Gurd, C. C. Kirkham, I. Watson, *The Manchester Prototype Dataflow Computer*, Communications of the ACM, Vol. 28, No. 1 (January 1985), pp. 34-52.

[Henderson80] Peter Henderson, *Functional Programming — Application and Implementation*, Prentice-Hall International (1980)

[Lampson78] B. W. Lampson, J. J. Horning, Ralph L. London, J. G. Mitchell, G. J. Popek, *Revised Report on the Programming Language Euclid*, Xerox Technical Center TR CSL-78-2 (1978)

[Liskov79] Barbara H. Liskov *et al.*, *CLU Reference manual*, MIT LCS TR-225 (1979)

[Marti80] Jed B. Marti, *Compilation Techniques for a Control-Flow Concurrent Lisp System*, 1980 Lisp Conference, pp. 203-207

[Marti83] Jed B. Marti, John Fitch, *The Bath Concurrent Lisp machine*, Eurocal '83, European Computer Algebra Conference, Lecture Notes in Computer Science No. 162, pp. 78-90 (1983)

[Milner84] Robin Milner, *A Proposal for Standard ML*, 1984 Symposium on Lisp and Functional Programming, August 1985, pp. 184-197

[Mitchell85] John Clifford Mitchell, Gordon D. Plotkin, *Abstract Types have Existential Types*, Twelfth Annual ACM Symposium on Principles of Programming Languages, January 1985, pp. 37-51

[Morris82] James H. Morris Jr., *Real Programming in Functional Languages*, in Darlington (Ed.), *Functional Programming and its Applications* (1982), pp. 129-176

[Reynolds74] John C. Reynolds, *Towards a Theory of Type Structure, International Programming Symposium*, Lecture Notes in Computer Science no. 19, pp. 408-425 (1974)

[Reynolds78] John C. Reynolds, *Syntactic Control of Interference*, Fifth Annual ACM Symposium on Principles of Programming Languages (January 1978), pp. 39-46

[Steele78] Guy L. Steele, *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*, MIT AI TR-474 (May 1978)