

Compiling FX on the CM-2*

Jean-Pierre Talpin[†] Pierre Jouvelot

CRI, Ecole des Mines de Paris, France

Abstract

Type and effect systems provide a safe and effective means of programming high-performance parallel computers with a high-level language that integrates both functional and imperative paradigms. Just as types describe what expressions compute, effects describe how expressions compute. Types and effects are associated with regions that describe where dynamically-allocated data structures reside in memory. We show how types, regions and effects can be used to discover when global operations on vectors are amenable to data parallelism in the presence of both side effects and higher-order functions.

To substantiate our claims that effects are an effective medium for addressing the issues of code generation for full-fledged languages on massively parallel computers, we describe the design and implementation of a CM-2 compiler prototype for the polymorphically typed FX language.

1 Introduction

The functional and imperative programming paradigms are often integrated together within sequential languages such as Common Lisp[11], Scheme[10] or Standard ML[7]. In such languages, implementors must exert care when designing code optimizers since side effects inhibit most of the nice properties of pure functional languages which are put at work in code transformations.

Going from sequentiality to parallelism, issues get significantly more complicated, both at the programmer and implementor levels. Concomitant use of side effects and parallelism leads to nondeterminism, which makes program understanding and debugging difficult because of the non-reproducibility of results. Restricting parallel programs to be deterministic, as advocated in[11], is a way of making parallel program design in higher-order imperative languages a more manageable task. Based on the concept of an effect system[5], we present here a compile-time technique that enforces such deterministic constraints and prove its effectiveness by describing a prototype compiler that targets the FX programming language[3] to the Connection Machine¹ (CM-2).

The purpose of the FX/CM compiler is to demonstrate the effectiveness of program analysis and code transformations based on type and effect information for high-level higher-order imperative languages. Our compiler uses the type and effect system presented in [13] to determine when operations on vectors are amenable to data parallelism in the presence of both side effects and higher-order functions. The absence of side-effects, for an operation mapped on every element of a vector, guarantees that its execution in parallel will not cause interferences. Such operations are run in parallel while others are conservatorily limited to sequential execution

*This work was partially supported by MIT contract GC-R-117153.

[†]Current address: ECRC GmbH, Arabella Strasse 17, D-8000 Munchen 81.

¹Connection Machine and *Lisp are trademarks of Thinking Machines Corporation.

on the CM-2 front end. Our compiler uses regions to discover when the lifetime of locally allocated data structures is compatible with the memory model of the CM-2, which encourages the allocation of parallel vectors in the stack.

An implementation of these compile-time techniques has been integrated to the FX system, providing a CM-2 compiler that generates *Lisp code. Test programs have been run on both a *Lisp simulator[12] and a CM-2 to evaluate the practicality and the performance of our approach.

Plan After presenting the FX language (section 2), we give (section 3) a brief overview of the Connection Machine architecture and its object language *Lisp, survey (section 4) the related work, present (beginning in section 5) the essential design ideas of our analysis and code generation techniques, discuss (section 12) the interesting implementation issues before concluding (section 13).

2 The FX Language

In order to simplify the presentation, this section introduces a core language that integrates, like FX or Standard ML, the principal features of functional and imperative programming.

2.1 Syntax

The expressions e of the language are the elements of the term algebra generated by the grammar described below. It uses enclosing parentheses in the reminiscence of Scheme[10].

$$e ::= x \mid (e \ e') \mid (\text{op } e) \mid (\text{lambda } (x) \ e) \mid (\text{let } (x \ e) \ e')$$

x is a value identifier. The constructs $(e \ e')$ and $(\text{op } e)$ stand for the application of a function e or an operator op to an argument. The expression $(\text{lambda } (x) \ e)$ defines the function whose parameter is x and whose result is the value of e . The construct $(\text{let } (x \ e) \ e')$ lexically binds x to the value of e in e' .

2.2 Static Domains

In this section, we present the domains used in the static semantics of our language. We are first going to equip the language with a type system. The following term algebra defines the three basic kinds of the static semantics: regions, effects and types.

$$\begin{array}{ll} \rho & ::= r \mid \varrho & \text{regions} \\ \sigma & ::= \emptyset \mid \varsigma \mid \sigma \cup \sigma \mid \text{init}(\rho) \mid \text{read}(\rho) \mid \text{write}(\rho) & \text{effects} \\ \tau & ::= \text{int} \mid \alpha \mid \text{vector}_\rho(\tau) \mid \tau \xrightarrow{\sigma} \tau & \text{types} \end{array}$$

Regions ρ are either constants r or variables ϱ . Every data structure corresponds to a region which abstracts the memory locations in which it is allocated at run time. Two values are in the same region if they may share some memory locations.

Effects σ can either be the empty effect \emptyset , an effect variable ς , or a store effect $\text{init}(\rho)$, denoting the initialization of a mutable value in the region ρ , $\text{read}(\rho)$, which describes the access of a data in the region ρ , or $\text{write}(\rho)$, which denotes the assignment of a value to a mutable data. Effects are gathered with the infix set-union operator \cup and can be compared using the set-inclusive relation \supseteq . Effects are introduced by operations that perform side-effects on mutable data structures such as vectors (see section 2.5).

Types τ are basic data types, such as *int*, type variables α , mutable vectors $vector_\rho(\tau)$ of type τ in the region ρ and function types $\tau \xrightarrow{\sigma} \tau'$ from τ to τ' with a *latent effect* σ . The latent effect of a function is the effect incurred when the function is applied.

2.3 Expansiveness

The notion of type polymorphism is the most distinguished feature of Milner's polymorphic typing discipline[6]. It reflects the property that an expression can have several different types that can be generically represented by a type scheme. Among various typing discipline, we decide here to choose the simplest one, based on the notion of *expansiveness* of expressions. An expression e is expansive if $exp[[e]]$ holds:

$$\begin{aligned} exp[[e]] = & \text{case } e \text{ of } x \mid (\text{lambda } (x) \ e) \Rightarrow \text{false} \\ & (e \ e') \mid (\text{op } e) \Rightarrow \text{true} \\ & (\text{let } (x \ e) \ e') \Rightarrow exp[[e]] \vee exp[[e']] \end{aligned}$$

In expressions such as $(\text{let } (x \ e) \ e')$, non-expansive expressions e can be handled by the textual substitution $e'[e/x]$ of e for x in e' , avoiding capture of bound variables. This simple technique provides an equivalent way of expressing the property that non-expansive expressions may admit multiple types without adding the complication of introducing type schemes in the static semantics.

2.4 Static Semantics

We formulate type and effect inference by a deductive proof system that assigns a type and an effect to every expression of the language. The context in which an expressions is associated to a type and an effect is represented by a type environment \mathcal{E} which maps value identifiers to types. Deductions produce typing judgments $\mathcal{E} \vdash e : \tau, \sigma$ which read: "In the type environment \mathcal{E} the expression e has type τ and effect σ ".

$$\begin{array}{lcl} \frac{x \in Dom(\mathcal{E})}{\mathcal{E} \vdash x : \mathcal{E}(x), \emptyset} & \text{(var)} & \frac{\mathcal{E}_x + \{x \mapsto \tau\} \vdash e : \tau', \sigma}{\mathcal{E} \vdash (\text{lambda } (x) \ e) : \tau \xrightarrow{\sigma} \tau', \emptyset} \quad \text{(abs)} \\ \frac{\mathcal{E} \vdash e : \tau, \sigma \quad \sigma' \supseteq \sigma}{\mathcal{E} \vdash e : \tau, \sigma'} & \text{(does)} & \frac{\mathcal{E} \vdash e : \tau \xrightarrow{\sigma'} \tau', \sigma \quad \mathcal{E} \vdash e' : \tau, \sigma'}{\mathcal{E} \vdash (e \ e') : \tau', \sigma \cup \sigma' \cup \sigma''} \quad \text{(app)} \\ \frac{\neg exp[[e]] \quad \mathcal{E} \vdash e : \tau, \emptyset}{\mathcal{E} \vdash e'[e/x] : \tau', \sigma'} & \text{(let)} & \frac{exp[[e]] \quad \mathcal{E} \vdash e : \tau, \sigma}{\mathcal{E}_x + \{x \mapsto \tau\} \vdash e' : \tau', \sigma'}{\mathcal{E} \vdash (\text{let } (x \ e) \ e') : \tau', \sigma \cup \sigma'} \quad \text{(ilet)} \end{array}$$

The rules for abstraction (abs) and application (app) show the interesting interplay between types and effects: effects flow from where functions are defined to where they are used.

2.5 Operations on Vectors

Since we are interested in studying the practical applications of effect systems to implement data parallelism, we focus on the FX module describing operations on vectors. This module integrates the facilities provided by both Scheme, Fortran90 and the scan model[1]. Vectors are represented by the abstract data type $vector_\rho(\tau)$ which denotes mutable arrays allocated in the region ρ whose elements are of type τ .

<code>make-vector</code>	initialization	$\tau \times \text{int} \xrightarrow{\text{init}(\rho)} \text{vector}_\rho(\tau)$
<code>vector-ref</code>	dereference	$\text{vector}_\rho(\tau) \times \text{int} \xrightarrow{\text{read}(\rho)} \tau$
<code>vector-set!</code>	assignment	$\text{vector}_\rho(\tau) \times \text{int} \times \tau \xrightarrow{\text{write}(\rho)} \text{unit}$
<code>vector-length</code>	length	$\text{vector}_\rho(\tau) \xrightarrow{\text{read}(\rho)} \text{int}$

The initialization performed by `(make-vector v n)` allocates a vector of length `n` initialized to the value `v`. The operation `(vector-ref v n)` dereferences the `n`th element of the vector `v`. The assignment of the `n`th element of the vector `v` to the value `e` is performed by the operation `(vector-set! v n e)`. The operation `(vector-length v)` returns the length of a vector `v`.

<code>identity</code>	identity perm.	$\text{int} \xrightarrow{\text{init}(\rho)} \text{vector}_\rho(\text{int})$
<code>permute</code>	regular perm.	$\text{vector}_\rho(\text{int}) \times \text{vector}_{\rho'}(\tau) \xrightarrow{\text{read}(\rho) \cup \text{read}(\rho') \cup \text{init}(\rho'')} \text{vector}_{\rho''}(\tau)$
<code>cshift</code>	circular shift	$\text{int} \times \text{vector}_\rho(\tau) \xrightarrow{\text{read}(\rho) \cup \text{init}(\rho')} \text{vector}_{\rho'}(\tau)$
<code>compress</code>	compression	$\text{vector}_\rho(\text{bool}) \times \text{vector}_{\rho'}(\tau) \xrightarrow{\text{read}(\rho) \cup \text{read}(\rho') \cup \text{init}(\rho'')} \text{vector}_{\rho''}(\tau)$

Permutations implement rearrangements of vectors. When vectors are implemented by distributed data structures, as on the CM-2 (see below), permutations implement inter-process communications. Here, `(identity n)` returns the identity permutation from 1 to `n`; `(permute p v)` performs the rearrangement of `v` according to the permutation `p`; `(cshift n v)` and `(eoshift n v v')` are the usual circular and end-off shift permutations. The expressions `(compress s v)` and `(expand s v v')` are less usual:

```

fx> (compress #(true false true) #(1 2 3))
= #(1 3)
fx> (expand #(true false true false true) #(1 2 3) #(0 0 0 0 0))
= #(1 0 2 0 3)

```

In `compress`, the vectors `s` and `v` should be of the same size. This operation selects and concatenates the elements of `vi` such that the `si` are true. In the operation `(expand s v v')`, `s` is of the same length that `v'` and bigger than `v`. When `sj` is true for the *i*th time, `vi` is selected, `v'j` otherwise.

<code>vector-map</code>	mapping	$(\tau \xrightarrow{\sigma} \tau') \times \text{vector}_\rho(\tau) \xrightarrow{\sigma \cup \text{read}(\rho) \cup \text{init}(\rho')} \text{vector}_{\rho'}(\tau')$
<code>vector-scan</code>	scanning	$(\tau \times \tau \xrightarrow{\sigma} \tau) \times \text{vector}_\rho(\tau) \xrightarrow{\sigma \cup \text{read}(\rho) \cup \text{init}(\rho')} \text{vector}_{\rho'}(\tau)$
<code>vector-reduce</code>	reduction	$(\tau \times \tau \xrightarrow{\sigma} \tau) \times \text{vector}_\rho(\tau) \xrightarrow{\sigma \cup \text{read}(\rho)} \tau$

In addition to the standard Scheme-like basic vector operations, the current vector module supports the mapping and reduction of first-class functions in the way of [1] and Fortran90 array extensions [2].

```

fx> (vector-scan and #(true false true false true))
= #(true false false false false)
fx> (vector-reduce and #(true false true false true))
= false

```

Here, `(vector-map f v)` applies the unary higher-order function `f` on every element of the vector `v` and returns the vector of the successive applications of `f`; `(vector-scan f v)` and `(segmented-scan f s v)` sum up the binary higher-order function `f` over every element of the vector `v` and returns the vector of the successive applications of `f`. In `segmented-scan`, summation is reset at `vi` if `si` is false. In `vector-reduce`, the sum is returned.

3 *Lisp and the CM-2

The CM-2 is based on the SIMD model (Single Instruction Multiple Data). It is composed of up to 64k processing elements (PE), wrapped by groups of 32 processors and local memory units, connected into a global hypercube communication network. A front-end workstation issues instructions and transfers data in a time-step fashion to the CM-2.

*Lisp[12] is an extension of Common Lisp[11] that implements the PARallel Instruction Set (PARIS) and supports specific data structures: parallel variables or *pvars*. A pvar is a vector whose elements are allocated in the memory of every processing unit of the CM-2. In contrast to the usual implementation of vectors in Common Lisp systems, pvar components are unboxed values of fixed size such as booleans, fixnums or floats. The type of every pvar manipulated in *Lisp programs must be declared by the programmer.

On the CM-2, the local memory of each processing element is divided into a heap area and a stack area. By default, a *Lisp pvar expression is allocated in the current call stack frame. There are several ways of creating pvars. The most common way is to use implicit temporary allocation with the distribution operation `!!` or `coerce!!`. Pvars can also be explicitly stack allocated by using the `*let` construct:

```
*lisp> (*let ((x (coerce!! (+ 2 2) '(pvar fixnum))))
        (declare (type (pvar fixnum) x))
        (*deallocate (allocate!! x)))
```

In this example, enough space for a `fixnum` on the stack of each PE is allocated and then the value 4 is moved to this address on every PE, using the operation `coerce!!`. Note that declaring the type of `x` is required for the expression to be correctly compiled. Storage management of pvar expressions in the heap can also be explicit. The operations that perform heap allocation and reclamation respectively are `allocate!!` and `*deallocate`.

For arithmetic computations, the *Lisp system extends Common Lisp's generic operations on numbers. They are implemented by the so-called *bang-bang* functions. For example, the operation `++` is the equivalent of `+`. `!!` functions operate in parallel on every component of their two pvar arguments and return a pvar, such as in the following example:

```
*lisp> (++ (!! 1) (!! 1))
= #<pvar x :general *default-vp-set* (1024)>
```

4 Related Work

In order to go beyond the crude *Lisp system[12], several other programming paradigms have been suggested. The major proposals are the APL-inspired alpha and beta global operations on vectors in CM-Lisp [4], the paralation abstract data type and its element-wise operations in Paralation-Lisp[9] or the scan operations over segmented vectors in SV-Lisp[1].

In the *Lisp language, the low-level programming features that are introduced in order to efficiently use the CM are all easy to compile. Not so for these other languages. However, their reference manuals [4, 9, 1] are quite elusive on the issue of which compile-time analysis would be necessary for programs to be effectively compiled.

Nonetheless, the data-parallel constructs designed in the SV-Lisp language[1] provide a programming model that made its way into the design of the vector

module for the FX language. It proved here to be effectively compilable using our type and effect system. Our approach is thus not to introduce a new data model, but to describe how a sophisticated static system can be used to safely implement data parallel constructs on a massively parallel machine in the presence of side effects.

5 Overview of the Compiler

We introduce a series of new compile time techniques based on our type and effect inference system to detect when the use of operations on vectors is actually amenable to data-parallel execution (no inhibiting side-effects) on the CM-2 (no unimplementable parallel operations).

In FX, programs are implicitly typed, may have side-effects and may use first-class functions. Our compiler generates *Lisp programs that use pvars with explicit typing, explicit parallelism, explicit management of stack and heap storage and explicit name space assignment. (*Lisp uses multiple name spaces for function and value identifiers.) In the static semantics of FX, every expression is associated with its type and effect. The criterion of parallelizability of expressions is based on type and effect. Parallelizable expressions must manipulate scalar data structures and have no side-effects.

Parallelizable FX functions are translated into specific data structures, noted *f*-structure, built with the operator `make-f-struct`, which is a tagged pair of functions: the sequential version, the `f-seq` component, and the parallel one, the `f-par` component, operating upon pvars.

The compiler translates FX vector operations to the invocations of appropriate macros of the runtime library that implement the operations on unboxed pvars for the corresponding type of vectors operated upon.

Even though FX is a strongly typed language, the addition of `let`-bound identifiers introduces generic polymorphism. The presence of type and effect variables requires the use of run-time type dispatch for vector operations.

6 Vector Allocation

Vectors are represented by pvars. Vectors with scalar components, such as boolean or integer, are implemented by unboxed pvars. Vectors with non-scalar components, such as lists or other vectors, are represented by boxed pvars, called *front-end* pvars, which are pvars of addresses to objects that reside on the front-end i.e. front-end objects. Since *Lisp is dynamically typed, every pvar is associated on the front-end with a description header giving its actual address on every processing element and the size and type of its components.

Temporary allocation of pvars is preferred wherever possible in the generated *Lisp code in order to avoid the overhead of superfluous heap allocation. The FX compiler uses types and effects to decide when a returned value must be explicitly moved to the heap.

Definition 1 (Heap Allocation Criterion) *A vector operation (`op e1 ... en`) that performs the allocation of a vector, of type $vector_{\rho}(\tau)$, must allocate its value in the heap in one of the following circumstances:*

- *The vector operation is the argument of an application expression (`e []`) whose type is $ref_{\rho'}(\tau')$, $vector_{\rho'}(\tau')$, $list(\tau')$, or $\tau'' \xrightarrow{\sigma} \tau'$. The region ρ of the vector argument occurs free in τ' or σ .*

- The vector operation occurs in the body of a lambda expression of the form `(lambda (x) C[])` whose type is $\tau'' \xrightarrow{\sigma} \tau'$. The region ρ of the vector operation occurs free in τ' or in a subterm τ''' of τ'' , be it a data structure $\text{ref}_{\rho'}(\tau''')$, $\text{vector}_{\rho'}(\tau''')$ or $\text{list}(\tau''')$.

If the pvar outlives the stack frame it is allocated into, i.e. if it can be referenced in its lexical environment, it must be moved in the heap with an explicit call to `allocate!!`. The criterion presented above syntactically controls the cases in which this situations may happen and that can actually be checked by using the effect system.

7 Runtime Library

The runtime library provides *Lisp functions and macros to implement the FX vector module. For each vector operation in FX (such as `vector-ref`), specialized *Lisp macros and a generic function are defined in the runtime as follows:

- The *Lisp macros implement the data-parallel polymorphic FX operation for every type of unboxed pvars: booleans, characters, integers, reals and complex. Such an operation is also defined on front-end pvars (the macro `front-end-vector-ref` in the present example).
- The generic function (in the example, `vector-ref`) implements the operation for every type of pvars. It uses a dispatch construct depending on pvar headers to check the actual type of pvar arguments at run time and call the appropriate specialized macro.

By default, all global vector operations are performed in parallel. The FX operations that accept higher order functions, such as `vector-map`, obey a different rule which is that the front-end version, `front-end-vector-map`, implements the sequential version on generic pvar arguments.

Finally, note that *Lisp operations on pvar are restricted to the set of active PEs, or *VP set*. Similarly, parallel vector operations are in FX limited to their actual size. In the runtime library, this is implemented by the macro `(with-context-of e e')` that disables, during the execution of e' , the processing elements on which the pvar expression e is not defined.

8 Sequential Code Generation

Having briefly presented *Lisp, our FX vector module and its corresponding *Lisp runtime library, we can now describe the *Lisp code generation scheme for sequential expressions. The input of the compiler is an expression `te` decorated with its principal type τ and its minimal effect σ .

The sequential code generation scheme [14] is specified by an algorithm $SC[[te]]\vec{\tau}\vec{x}$, which relates a typed FX expression `te` that was successfully typed checked by the algorithm of [13] with its corresponding *Lisp code, in a given compilation context $\vec{\tau}$. The context $\vec{\tau}$ is a sequence of types that are used decide when vector allocations must be performed in heap (according to Definition 1). \vec{x} is the sequence of parallelizable predefined functions, such as `+`, `-`, etc.

Identifiers

The compiler translates a lambda-bound identifier x to `x`. Identifiers x that appear in \vec{x} are predefined parallelizable functions. They are thus compiled by an *f*-structure `(make-f-struct x x!!)`.

Abstraction

To translate an abstraction $(\text{lambda } (x:\tau) \text{ te}):\tau'$, the compiler compiles the body te of the source FX abstraction into target code c . If the type τ' agrees with the predicate PF , defined below, the lambda-abstraction is parallelizable. An f -structure initialization is generated, which pairs up the sequential code of the function, generated by SC , with its parallel version, returned by parallel code generator PC , presented in the next section.

If the function is not parallelizable then the compiler only returns the sequential code of the lambda abstraction. The compiler uses the following static criterion, PF , based on function types, to determine if a lambda of type τ' is parallelizable.

Definition 2 (Parallelizability Criterion) *A lambda-expression of type $\tau \xrightarrow{\sigma} \tau'$ is parallelizable (satisfies the criterion PF) if and only if its latent effect σ is \emptyset or a union of effect variables, and if the types τ and τ' are scalar data types, type variables or function types that satisfy the criteria $PF(\tau \xrightarrow{\sigma} \tau') = PA(\tau) \wedge PA(\tau') \wedge PE(\sigma)$, where*

$$\begin{array}{ll}
 PA(\tau) = \text{case } \tau \text{ of} & PE(\sigma) = \text{case } \sigma \text{ of} \\
 \text{bool} \mid \text{int} \mid \text{real} \mid \alpha & \Rightarrow \text{true} & \emptyset \mid \zeta & \Rightarrow \text{true} \\
 \tau' \xrightarrow{\sigma} \tau'' & \Rightarrow PF(\tau) & \sigma \cup \sigma' & \Rightarrow PE(\sigma) \wedge PE(\sigma') \\
 \text{otherwise} & \Rightarrow \text{false} & \text{otherwise} & \Rightarrow \text{false}
 \end{array}$$

This criterion has both compile-time and runtime aspects. Its compile-time aspect can be informally justified in the following way. First, the lambda expression must have no side effects: no write effects $\text{write}(\rho)$ must occur in σ , since they could generate non-determinism at run time. Also, no initialization $\text{init}(\rho)$ or read effects $\text{read}(\rho)$ may appear, as they would indicate that the function allocates or manipulates non-scalar values which are unimplementable on the Connection Machine. Second, the types τ and τ' must be scalar variables or function types.

Runtime checks are only required in the presence of effect variables in σ , to distinguish whether these effect variables actually are \emptyset or not in each given instance. These effect variables are introduced by the latent effects of higher-order functions, other lambda-bound function identifiers. However, we know that every pure first-order function is compiled as an f -structure. Thus, we use the predicate f-struct-p to decide at runtime whether the free function identifiers of a lambda-abstraction actually correspond to other f -structures. When this condition is met, an f -structure is returned.

For instance, the code generated for $(\text{lambda } (g) (\text{lambda } (x) (g \ x)))$ is as below. (The identifier id is fresh and that *Lisp function *funccall applies a function value operating on pvars to its arguments.)

```

(lambda (g)
  (let ((id #'(lambda (x)
                (funcall (if (f-struct-p g) (f-seq g) g) x))))
    (if (f-struct-p g)
        (make-f-struct id #'(lambda (x!!) (*funccall (f-par g) x!!)))
        id)))

```

Application

As shown in the previous example, the compiler translate application expressions $(\text{te te}'):\tau!\sigma$ by translating the subexpressions te and te' into *Lisp code c and c' and by generating the code that checks whether c is an f -structure or not.

Simplifications

More efficient compilation mechanisms for vector operations are given in the subsection 10. However, simple syntactic rewriting rules can already be used to improve the code generated by the previous technique. They are the following:

<code>(f-seq (make-f-struct c c'))</code>	$\Rightarrow c$	<code>(funcall #'c c')</code>	$\Rightarrow (c c')$
<code>(f-par (make-f-struct c c'))</code>	$\Rightarrow c'$	<code>(*funcall #'c c')</code>	$\Rightarrow (c c')$
<code>(f-struct-p (make-f-struct c c'))</code>	$\Rightarrow t$	<code>(if t c c')</code>	$\Rightarrow c$
<code>(f-struct-p #'(lambda (x) c))</code>	$\Rightarrow \text{nil}$	<code>(if nil c c')</code>	$\Rightarrow c'$

9 Parallel Code Generation

We describe the *Lisp parallel code generation scheme implemented by the algorithm $PC[\mathbf{te}]\vec{x} = c$ which, given a typed FX expression \mathbf{te} and a sequence of parallel value identifiers \vec{x} , generates the parallel *Lisp code for it, for execution on the CM-2 processors. Parallel value identifiers in \vec{x} are either predefined arithmetic operations, such as $+$ or $-$, which are implemented on the CM-2, or user value identifiers bound by lambda abstractions.

Identifier

The compiler translates a lambda-bound identifier \mathbf{x} , appearing in the sequence \vec{x} , by $\mathbf{x}!!$. Free identifiers correspond to values that are imported in the parallel code. Scalar identifiers \mathbf{x} (as well as constants $1, \dots, \mathbf{n}$) are distributed and coerced to a pvar of the corresponding type. Functions identifiers \mathbf{f} are translated to `(f-par f)`.

Identifiers of variable type use the type dispatching expression `(distribute x)` of the FX runtime library to precisely distinguish at run time between the previous cases. Other (mutable) data structure identifiers are guaranteed by the static semantics to never be used in parallel code.

Abstraction

In the compilation of lambda-abstractions, since *Lisp doesn't create real closures, the free pvar identifiers of the compiled lambda expression must be heap allocated, because the lambda abstraction may escape from the stack frame at which those pvar identifiers are allocated.

It is the case in the following example, which also shows that our compilation technique support the presence of higher order functions. The function `(lambda (f x) (lambda (y) (f x y)))` partially applies \mathbf{f} to \mathbf{x} and is translated it by the following parallel code:

```
#'(lambda (f!! x!!)
  (declare (type pvar x!!)
           (type (function (pvar) pvar) f!!))
  (let ((x1!! (heap-allocate x!!)))
    (declare (type pvar x1!!))
    #'(lambda (y!!)
        (declare (return-pvar-p t)
                 (type pvar y!!))
        (*funcall f!! x1!! y!!))))
```

where `x1!!` is fresh and `heap-allocate` is the generic runtime type-dispatching function for `allocate!!` on pvars. The declaration `return-pvar-p` is used to explicitly tell the *Lisp compiler that an expression returns a pvar.

Application

The parallel code generated for an application is decomposed into the code generation for its subexpressions and the invocation of the parallel function via the *Lisp construct `*funcall`.

Parallel `if` expressions are translated into *Lisp `if!!` expressions. The compilation of `if` differs from standard applications for two reasons. First, the semantics of the *Lisp expression `(if!! e e' e'')` is to execute both `e'` and `e''` and return the value of `e'` where `e` is true and `e''` otherwise. To ensure termination, we add code to check that at least one processor is active before executing the `if!!` form itself. Second, `if!!` expressions can only return pvars even though FX `if` expressions may return functions. Thus, calls to these functions must be interned within the branches of the `if` construct.

Before being compiled, the expression `((if b 1+ 1-) x)` is first transformed into the following,

```
((lambda (y) (if b (1+ y) (1- y))) x)
```

where the function expression `(if b 1+ 1-)` is abstracted over a fresh `y`. This expression gets compiled into the following *Lisp code, where `*or` performs a machine-wide reduction. If no processors are active, `nil` is returned by `*or`.

```
((lambda (y!!) (if (*or t!!)
                  (if!! b!! (1+!! y!!) (1-!! y!!))
                  (!! 0)))
 x!!)
```

10 Optimization of Vector Operations

The code generated by the scheme described above can be dramatically improved in a variety of ways by using simple type and effect based optimizations, especially on vector operations. A vector operation `(op e)` is a particular case of application statement. The default mechanism for optimizing such an operation is to look at the type of the vector $vector_\rho(\tau)$ operated upon in the expression.

When τ is a scalar data type, such as *real*, the vector is implemented by an unboxed pvar of scalar components. In this case, `op` is translated by a call to the appropriate *Lisp macro. When τ is a non-scalar datatype, such as a *vector*, the vector is implemented as a boxed front-end pvar, and the operation `op` by the macro `front-end-op` of the runtime library. Otherwise, τ is a type variable and the compiler translates the operation into a call to the generic function of the runtime library implementing the operation, named `op`. In a way similar to types, effects can be used to specialize some higher-order vector operations which require runtime tests. (See [14] for more details on these various optimizations.)

11 Compilation of let

Managing the `let` construct within the simplified compilation scheme shown above is easy. Value definitions such as `(let ((x e)) e')` are handled according to our basic compilation scheme by translating them into `((lambda (x) e') e)`.

An identifier f is defined as a function when it is bound in a `let` expression to an explicit lambda expression. Function definitions are translated by using the Common-Lisp `labels` construct. When `(lambda (x) e)`, bound to f in a `let` construct, is a function that can be parallelized, its parallel implementation is associated to $f!!$. The sequence of parallelized functions \vec{x} is updated to associate the identifiers f of parallelizable functions with the lambda-bound function identifiers \vec{f} on which the parallelizability of f actually depends at run time.

An occurrence of a function identifier f in function position (`f e`) is left as such during sequential code compilation. An occurrence in value position, such as in `(lambda (x) f)` or `(e f)` is translated, as previously, by the allocation of an f -structure if f was added to \vec{x} . If f is parallelizable (modulo the possible run-time check on its free function identifiers) the f -structure (`make-f-struct #'f #'f!!`) is returned. Otherwise, `#'f` is generated.

Example For instance, the following definition:

```
(lambda (g)
  (let ((f (lambda (x) (g x))))
    f))
```

is compiled as:

```
(lambda (g)
  (labels (((f x) (funcall (if (f-struct-p g) (f-seq g) g) x))
            ((f!! x!!) (*funcall (f-par g) x!!))))
    (if (f-struct-p g) (make-f-struct #'f #'f!!) #'f)))
```

During parallel code generation, an occurrence of a parallelizable function identifier f in function position is translated to $f!!$. The static semantics guarantees that non-parallelizable functions occurring in parallelizable FX expressions are never used. The occurrence of f in a value position, such as in `(lambda (x) f)` or `(e f)`, is translated into `#'f!!`.

12 Implementation

The FX compiler for the CM-2 was implemented by using the initial implementation of the FX-91 interpreter, consisting of a parser, a kind and type checker and a simple interpreter. The FX-91 interpreter is written in Scheme and runs under T[8] and has been adapted to *Lisp. The techniques described in this paper have all been implemented.

The FX/CM Compiler has first been tested on the *Lisp simulator[12]. We have then used the CM-2 installed at ETCA (Arcueil, France) and run more interesting data-parallel algorithms, such as a `life` program, a `quicksort` algorithm using segmented scanning and a matrix transposition algorithm.

Example We illustrate the speed-up obtained by our compiler the segmented matrix transposition algorithm `segment-transpose`, described below, which uses higher-order functions and segmented vectors as 2D matrices.

```
(define (segment-transpose m segment)
  (let ((id (segment-identity segment))
        (offset (1+ (vector-reduce max id))))
    (permute m
      (vector-map2 + (segment-index segment)
                    (vector-map (lambda (i) (* offset i))
                                id)))))
```

It was executed on 8×8 to 128×128 integer matrices on a 8K processor CM-200a. Busy times are almost independent of the problem size, except for the last case. For a 128×128 matrix, the VP set exceeds the actual machine size and thus multiple operations are performed on each processor, slowing down the overall execution.

Matrix size	8×8	16×16	32×32	64×64	128×128
Elapsed Time (s)	1.48×10^{-2}	1.51×10^{-2}	1.83×10^{-2}	1.52×10^{-2}	1.74×10^{-2}
Busy Time (s)	6.47×10^{-3}	6.48×10^{-3}	6.43×10^{-3}	6.43×10^{-3}	1.04×10^{-2}

13 Conclusion

The FX/CM compiler prototype supports the claim that effect systems can be used to integrate, for the first time, functional and imperative programming on massively parallel architectures. Effects are used to decide whether potentially parallel constructs can actually be implemented as such without leading to non-determinism or do not use local data structures that inhibit their efficient implementation on existing parallel hardware. Regions are used to optimize space allocation strategies and limit the garbage collection overhead.

Acknowledgements

We thank J. P. Massar (Thinking Machines Corporation) for discussions on the issue of implementing a pvar garbage-collector in *Lisp, C. Millour (Consultant at ETCA) for his experience in the use of *Lisp, P. Clermont and S. Petiton (ETCA) for allowing us to access the CM-2 at SEH.

References

- [1] BLELLOCH, G. E. Vector model for data-parallel computing. The MIT Press, 1990.
- [2] ANSI, *Standard for the Information Systems Programming Language Fortran: S8(X3.9-198x)*, 1989.
- [3] GIFFORD, D. K., JOUVELOT, P., LUCASSEN, J. M., AND SHELDON, M. A. FX-87 Reference Manual. *MIT/LCS/TR-407*, MIT Laboratory for Computer Science, September 1987.
- [4] HILLIS, W. D. The Connection Machine. The MIT Press, 1985.
- [5] LUCASSEN, J. M., AND GIFFORD, D. K. Polymorphic Effect Systems. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1988.
- [6] MILNER, R. A Theory for type polymorphism in programming. In *Journal of Computer and Systems Sciences*, Vol. 17, pages 348-375. 1978.
- [7] MILNER, R., TOFTE, M., HARPER, R. The definition of Standard ML. *The MIT Press*, 1990.
- [8] REES, J. A., ADAMS, N. I., AND MEEHAN, J. R. *The T Manual, fourth edition*. Yale University 1984.
- [9] SABOT, G. W. *The Paralation Model*. MIT Press 1990.

- [10] REES, J., AND CLINGER W., EDITORS. Fourth Report on the Algorithmic Language Scheme, 1988.
- [11] STEELE, G. L. *Common Lisp, the language*. Digital Press 1990.
- [12] *Lisp Reference Manual, version 4.0. Thinking Machines Corporation, Cambridge, 1987.
- [13] TALPIN, J. P., AND JOUVELOT, P. Polymorphic Type, Region and Effect Inference. In the *Journal of Functional Programming*, volume 2, number 3. Cambridge University Press, 1992.
- [14] TALPIN, J. P. *Aspects Théoriques et Pratiques de l'Inférence de Type et d'Effet*. Doctoral dissertation. University Paris VI, May 12th, 1993.