J. Symbolic Computation (1987) **4**, 269–293

Functional Programming with Combinators

JACEK GIBERT

Department of Computer Science, University of Melbourne, Parkville, Victoria, 3052, Australia

(*Received* 22 *October* 1984; *Revised* 1 *December* 1986)

Combinators are shown to provide a very suitable basis for implementations of functional and symbolic computation in computer architecture. A powerful combinator reduction system is developed which meets programmers and machine requirements for (i) efficiency of representation and execution of symbolic algorithms, and (ii) availability of algebraic manipulation needed to analyse symbolic computations. An algebraic model is constructed to provide rigorous semantics for the system. The reduction language of the system aims at exposing efficient flows of data and fine-grain parallelism, and a computer architecture, which is proposed to run the system, utilizes both sequential and parallel processing modes in order to achieve maximum efficiency of symbolic computation. Finally, an implementation of the interpreter and functional simulator for the architecture is described.

1. Introduction

As a result of developments in the design of non von Neumann style of programming, in particular Backus' Functional Programming (FP), and also in the refinement of novel hardware concepts such as control flow, data flow or reduction machines, programming languages for symbolic computationare no longer viewed only as part of the theoretical foundations of programming or as specification languages, but are also seen as practical tools for programmers.

However at present it still remains the case that most practical programming languages used for symbolic computation are only incomplete versions of various logics developed to deal with symbolic objects, e.g. LISP vs. lambda calculus (Eick & Fehr, 1983) or FP vs. combinatory theory (Backus, 1978). Various reasons for this situation will be discussed in the following paragraphs.

The well developed mathematical theory of lambda calculus provides powerful tools for symbolic execution of programs (McCarthy, 1960). All lambda-defined functions are all computable functions and the simple axiom schemes α and β give a notion of symbolic computation via reduction. However, the α and β reductions are inefficient when performed directly by a machine, because substitutions in the presence of bound variables used in these reductions are veryexpensive, and an efficient execution of a program is gained at the price of inconsistencies in semantics of a programming language (Eick & Fehr, 1983). The problem of efficiency can be avoided by

0747–7171/87/060269+25 $03.00/0

using combinators as a basis for a programming language while preserving the complete semantics of combinatory theory.

The recent development in combinatory theory (Engeler, 1977; (Engeler, 1981; Engeler 1984; Obtulowicz & Wiweger, 1981; see also Meyer, 1982) reveals how to use it as a theory of equations for a class of ordinary algebraic structures. In this approach algorithmic problems such as "find x such that F(x)" are reduced to combinatorial equations of the form: $t_1(x) = t_2(x)$, where t_1 and t_2 are terms over an algebraic structure. To solve the problem F simply means to find an element a of the structure which satisfies the equation, i.e. $t_1(a) = t_2(a)$. A number of ways for solving such combinatorial equations are well known, one of which is the lambda abstraction. Similarly, to solve the problem F by symbolic manipulation would mean firstly, describe a as a closed term (a program), i.e. term with no free variables, secondly simplify it according to axioms of the algebraic structure.

A combinator based functional language such as Backus' FP supports "structured programming" (Backus, 1978). It offers compactness of notation, as single combinators express highly complex operations on structured objects. The programs automatically possess algebraic properties of combinators which allow a programmer to reason about the programs at the function level rather then at the object level (Backus, 1981) and provide simple tools for program transformation and verification.

Furthermore, the combinatory approach appears to build a bridge between programmers and machine requirements for a language and a machine architecture for symbolic computation. Machines to model combinatory systems have uncomplicated structures and they run very efficiently as compared to lambda calculus machines, (Turner, 1979b; Hughes, 1982; Peyton-Jones, 1982; Jones & Muchnick, 1982; Stoye *et al.*, 1984). The absence of environment makes a combinatory code very attractive for implementation in data driven or data flow architectures, (Sleep 1980; Maurer & Oldehoeft, 1983; also Amamiya & Hasegawa, 1984).

However the choice of combinators seems to be a difficult one. In his FP Backus (1978) uses very high level combinators (e.g. functionals such as apply-to-all, insert, and construction) and he restricts them to play only the role of program forming operators (PFO's). PFO's allow him to construct a simple tool for verification of FP programs: the FP Algebra of Programs. But in this way FP suffers in expressive power as general higher order functions, or new PFO's cannot be defined. Moreover, all functions and functionals of FP, except conditional, are assumed to be strict ("undefined-preserving") in order to give a simple fixed point semantics for the language. However many functional computations are expressed more naturally by non-strict functions (Friedman & Wise, 1976), the simplest example of which is the conditional (if-then-else statement). The FP algebra can be used to deal with recursive programs containing conditionals only if they are of a certain type: linear (Backus, 1978) and nonlinear "overrun-tolerant" (Williams, 1982).

The limitations of FP can be remedied by embedding FP into some complete algebraic combinatory structure such as Bohm's combinatory

monoid (Bohm, 1982) or a combinatory algebra (Gibert, 1983a) which uses the standard combinators of Combinatory Logic. Such combinatory structures preserve the algebraic nature of the functional language and provide a complete semantics for it. At the same time the standard combinators can be viewed as simple operators which "do nothing more than move data around", and therefore they provide for simple operational semantics and a possible machine architecture.

The standard combinators such as I, S, K, B, C and Y have been used for implementing functional programs by Turner (Turner, 1979b). These combinators are attractive because of the simplicity of the abstraction algorithm, i.e. the transformation process from conventional functional expressions into "free-from-bound-variables" combinatory code. Turner has shown that the resulting code for a functional program generated using these combinators is excessively long, far removed from a source program and too inefficient for practical applications. Although he later improved the performance of the abstraction algorithm by using new combinators S',B', C' (Turner, 1979a), the big gap between a combinatory code and a source program still remained, making the understanding of a computation process and debugging practically impossible because intermediate values during the computation could not be tracked down.

A solution may lie in providing a collection of combinators that closely correspond to a functional notation, so some intermediate "state" of evaluation would be easier to interpret. This approach has been taken by Hughes (1982) with his "super-combinators", which are dynamically defined generalized combinators (i.e. combinators defined during the elimination of bound variables from lambda expressions). Unfortunately, the loss of a fixed set of primitive combinators in Hughes' approach results in the costs of interpreting virtual instructions.

We have further developed Hughes' approach of using an unbounded number of combinators and put combinators into classes via combinatory schemata to obtain a limited base set of machine instructions. In (Gibert & Shepherd, 1983) the classes have been derived directly for Backus' FP from the equational specification of the language which has provided the basis for design of an elegant, algebraically structured FP compiler. In (Gibert, 1983) and (Gibert, 1984a) the classes of combinators have been generalized in order to serve as a basis for an efficient implementation of functional programming. An abstraction algorithm have been modified in such a way that it maps a potential for parallelism in functional programs into the combinatory code by abstracting more than one variable at a time (Abdali, 1976; Maurer & Oldehoeft, 1983; Gibert, 1983; Gibert, 1984a).

This paper demonstrates a new model for symbolic computation, based upon combinatory theory, which: (i) has an elementary algebraic construction that provides a natural environment for mechanical verification and analysis of programs, (ii) allows one to express programs in complete and compact forms which exhibits their control structures more clearly, therefore permitting algebraic manipulations and efficient parallel processing.

In particular, it is shown that a partial algebra of functions *Jaf*, defined here to be our model, is a consistent extension of the combinatory algebra, and that it gives simple semantics for practical programming languages such

as Backus' FP. It is argued that our language, *Jal*, developed along with the *Jaf* algebra can be seen as a powerful combinatory reduction language which is intuitively accessible to programmers and can be implemented efficiently in a machine architecture, *JAM*achine (Gibert, 1984b). The efficiency of the implementation is gained from such properties of *Jal* expressions as distributivity and freedom from bound variables. These properties simply mean that subexpressions of a *Jal* expression contain all necessary information to be evaluated separately in a parallel manner. The short description of the *Jal* language is illustrated by some examples of symbol manipulating programs.

The second section reviews the definition and basic properties of the *Jaf* algebra of functions, of which a preliminary version was presented in (Gibert, 1983).

The third section discusses programs defined by combinatorial equations between terms of *Jaf*, and shows that unique partial solutions of the equations can be found mechanically by algebraic manipulations. We prove that the *Jaf* algebra is essentially the extensional combinatory algebra.

The fourth section presents the implementation of the *Jal* language on JAM(achine) and demonstrates the convenience of using it in developing a practical functional programming language by modelling Backus' FP in *Jal* in a very natural way. Finally an interpreter and functional simulator of JAM is described. JAM is designed to execute functional programs in a data flow fashion but it can retain sequential evaluation for programs, or parts of them, which are inherently sequential.

2. *Jaf* Algebra of Functions

In this section we outline an approach to the interpretation of functional programs in combinatory algebras. We give an explicit algebraic construction of a model for functional programming in *cartesian closed universes*. A category-theoretic characterization of such models was given by Longo & Moggi (1984) and later by Curien (1986).

We start with a construction of a partial algebra of functions *Jaf* over a cartesian closed universe $Ccu(\mathcal{F}un, \mathcal{T}up)$, where $\mathcal{T}up$ is a set of arbitrarily long and nested tuples of *Ccu*, and $\mathcal{F}un$ is a set of functions of *Ccu* of arbitrary arity. We call tuples of length one that are not nested *atomic*. We regard some atomic tuples of $\mathcal{T}up$ as denoting functions of *Ccu* and certain non-atomic tuples as denoting parameterizefllnctions. We call a set of all such tuples Θ. Therefore let us define a pair of functions:

$$\mu: \Theta \rightarrow \Phi \quad \text{and} \quad \rho: \Phi \rightarrow \Theta$$

where $\Phi \subset \mathcal{F}un$, $\Theta \subset \mathcal{T}up$ such that

$$\mu \circ \rho = id_{\Phi},$$

and

$$\mu((f, a_1, ..., a_n)) = f_{a_1, ..., a_n}$$

where $f_{a_1, ..., a_n}$ is a function with parameters $a_1, ..., a_n$.

There are two classes of primitive functions in $\mathcal{F}un$: "projection" and "substitution" functions defined as follows:

$$\{ \mathcal{S}\{i,n\}: \mathcal{A}^n \to \mathcal{A};\ \mathcal{A} \subset \mathcal{T}up \}_{i,\,n,\,i \le n}$$

is the class of projection functions, where

(PROJ) $$\mathcal{S}\{i,n\}(a_1, ..., a_i, ..., a_n) = a_i.$$

$$\{ *\{n\}_{f,g}: \mathcal{A}^n \to \mathcal{A};\ \mathcal{A} \subset \mathcal{T}up \}_n$$

is the class of substitution functions, where

(SUBS) $$*\{n\}_{f,g}\,(a_1, ..., a_n) = f_{a_1, ..., a_n}(\,(g, a_1, ..., a_n)\,).$$

NOTATION. (i) Because of the complexity of notation, we will often avoid the explicit use of functions μ and ρ by adopting the convention that italic font will indicate functions of $\mathcal{F}un$, i.e. f will denote $\mu(f)$.
(ii) Let $S\{i,n\}$ denote $\rho(\mathcal{S}\{i,n\})$, $(f\ *\{n\}\ g)$ denote $\rho(*\{n\}_{f,g})$.
(iii) Let $\mathbf{T}, \mathbf{F} \in \Theta$ stand for $S\{1,2\}$, $S\{2,2\}$ respectively, and $\mathbf{I} \in \Theta$ stands for $S\{1,1\}$. T and F are often used in definitions by cases since

$$\mu(T)(a_1, a_2) = s\{1,2\}\,(a_1, a_2) = a_1,$$

and similarly

$$\mu(F)(a_1, a_2) = a_2.$$

The element T can also be used to define constant functions $\mathcal{T}_a$, i.e.

$$\mathcal{T}_a(b) = \mu(\,(T, a)\,)(\,b\,) = \mu(T)(a, b) = a.$$

The element id represents the identity function but only on atoms of $\mathcal{T}$.

The equality in $\mathcal{T}up$ is the identity relation and the equality between the elements of $\mathcal{F}$ is determined by the following "extensionality" rule: $f, g \in \mathcal{F}un$

(EXT) $$(\forall\, t \in \mathcal{T}up,\ f(t) = g(t)) \Rightarrow f = g.$$

DEFINITION. Let $\Phi \subset \mathcal{F}un$ and $\Theta \subset \mathcal{T}up$ be fixed and functions μ, ρ be given. The above definitions give rise to a (partial) extensional algebra of functions

$$\mathcal{A}f = < Ccu(\mathcal{F}un, \mathcal{T}up), :, *\{i\}, S\{i,j\}, I, T, F >$$

where ":" is a binary operation of application, if (PROJ), (SUBS), (EXT) and the following condition holds for all $t \in \mathcal{T}up$, $(a_1, ..., a_i, ..., a_n) \in \mathcal{A}^n$, $n \ge 1$, $\mathcal{A} \subset \mathcal{T}up$:

If $t \in \Theta$ *then*

$$\text{(APP1)}\quad t : (a_1, ..., a_i, ..., a_n) = \begin{cases} \mu(t)(a_1, ..., a_i, ..., a_n) & \text{if } \mu(t) \in [\,\mathcal{A}^n \to \mathcal{T}up\,] \\ \mu(t)(a_1, ..., a_i) : (a_{i+1}, ..., a_n) & \text{if } \mu(t) \in [\,\mathcal{A}^i \to \mathcal{T}up\,] \\ \mu(t)_{a_1, ..., a_i, ..., a_n} & \text{otherwise} \end{cases}$$

else

(APP2) $$t : (a_1, ..., a_i, ..., a_n) = (t, a_1, ..., a_i, ..., a_n).$$

NOTATION. (i) The application operation ":" is assumed to be left-associative, i.e.

$$t : a_1 : a_2 : \ldots : a_n = (\ldots((t : a_1) : a_2) \ldots : a_n).$$

(ii) We omit the index expression {1} after functions, e.g. * stands for *{1}.

REMARK. Let P = (I * I). *Jal* is only a partial algebra since, for example, (P : P) is indeterminate:

$$\begin{aligned} P : P &= \mu(P)(P) = *_{I,I}(P) = S\{1,1\}(P) : (I : P) = P : (I : P) \\ &= *_{I,I}(I : P) = S\{1,1\}(I : P) : (I : (I : P)) \\ &= S\{1,1\}(P) : (I : (I : P)) = P : (I : (I : P)) = \ldots \end{aligned}$$

The above definition of the *Jal* algebra was chosen as the simplest and most efficient one for an implementation in computer architecture (refer to section 4.4). A user, however, may wish to extend *Jaf* by adding new functions and establishing algebraic identities which can help deriving and proving properties of symbolic programs. Consider, for example, the following definitions of two functions, which we will find very useful in the following sections.

DEFINITION. Let T{n} denote S{1,n+1}. Then a "composition" operator •{n} can be defined as

$$f \bullet\{n\}\ g = (T\{n\}, f) *\{n\}\ g.$$

Some of the algebraic properties of the class of composition operators can be expressed, for example, by the following equivalences in *J*.

PROPOSITION. *For all* f, g, h $\in \Theta$ *and* $(a_1, \ldots, a_n) \in \mathcal{A}^n$,

(1) $(f \bullet g) : (a_1, \ldots, a_n) = f((g, a_1, \ldots, a_n))$ *where* $f = \mu(f)$,

(2) $(T, f) \bullet\{n\}\ g = (T\{n\}, f)$ *and* $(F, f) \bullet\{n\}\ g = g$,

(3) $(f \bullet\{n\}\ g) \bullet h = f \bullet\{n\}\ (g \bullet h)$,

(4) *If* m > n *then*

$$(f *\{m\}\ g) \bullet\{n\}\ h = (f \bullet\{n\}\ h) *\{m\}\ (g \bullet\{n\}\ h)$$

and also

$$(f \bullet\{m\}\ g) *\{n\}\ h = f \bullet\{m\}\ (g *\{n\}\ h).$$

PROOF. The proofs are straightforward verification and are omitted.

3. Combinatorial Equations in *Jaf*

At the end of the previous section we have demonstrated how we can use the *Jaf* algebra to directly define new functions from the primitive ones. In the following we discuss an alternative function construction mechanism in

Jaf, recursion. Recursion can be used to write functional programs in the form of recursive definitions. We show that (partial) canonical solutions to a finite set of combinatorial equations are determined uniquely in *Jaf* and can be found in a uniform way by simple abstraction algorithms based on the following completeness and fixed point results.

DEFINITION. $G(v_1, \dots, v_m)$, denoting a *term* over *Jaf* in variables from a fixed set of variables $\{v_1, \dots, v_m\}$, is defined by induction as follows:

(i) $v_1, \dots, v_m$ and elements of *Tup* are all terms,

(ii) if G_1, G_2 are terms then $G_1 : G_2$ and $G_1 *\{i\} G_2$ are also terms.

DEFINITION. Let $\mathcal{A} \subset Tup$. An applicative function $\gamma: \mathcal{A}^n \to \mathcal{A}$ is called *representable over Jaf* if

$$\exists g \in \Theta, \forall (a_1, \dots, a_n) \in \mathcal{A}^n, \; g : (a_1, \dots, a_n) = \gamma(a_1, \dots, a_n).$$

DEFINITION. Let $\mathcal{A} \subset Tup$. An applicative function $\gamma: \mathcal{A}^n \to \mathcal{A}$ is called algebraic in *Jaf* if there is a term G over *Jaf* in variables from a fixed set of variables $\{v_1, \dots, v_n\}$ such that for all $(a_1, \dots, a_n) \in \mathcal{A}^n$,

$$\gamma(a_1, \dots, a_n) = G(v_1/a_1, \dots, v_n/a_n),$$

where v/a denotes the simultaneous replacement of all occurrences of the variable v by the element a.

If we postulate that all applicative algebraic functions are representable in a system then the system is called combinatory complete (a notion attributed to Curry (1930)).

THEOREM (Combinatory Completeness). *The Jaf algebra is combinatory complete, i.e. if* $G(v_1, \dots, v_n)$ *is a term over Jaf in variables from a fixed set of variables* $\{v_1, \dots, v_n\}$ *then there exists an (extensionally unique) element f in* Θ *such that for all* $(a_1, \dots, a_n) \in \mathcal{A}^n$,

$$G(v_1/a_1, \dots, v_n/a_n) = f : (a_1, \dots, a_n).$$

PROOF. By induction on the structure of a term.

(i) If G contains no free variables then $f = (T\{n\}, G)$, because

$$f : (v_1, \dots, v_n) = \mu(T\{n\})(G, v_1, \dots, v_n) = S\{1,n+1\}(G, v_1, \dots, v_n) = G.$$

(ii) If G is a variable, i.e. $G(v_1, \dots, v_i, \dots, v_n) = v_i$ then $f = S\{i,n\}$ because

$$f : (v_1, \dots, v_n) = \mu(S\{i,n\})(v_1, \dots, v_n) = S\{i,n\}(v_1, \dots, v_n) = v_i.$$

(iii) If $G(v_1, \dots, v_n) = G_1(v_1, \dots, v_n) : G_2(v_1, \dots, v_n)$ then by induction step

$$G_1(v_1, \dots, v_n) = f_1 : (v_1, \dots, v_n)$$
$$G_2(v_1, \dots, v_n) = f_2 : (v_1, \dots, v_n).$$

Hence $f = f_1 *\{n\} f_2$, because

$$\begin{aligned} f : (v_1, \dots, v_n) &= (f_1 : (v_1, \dots, v_n)) : (f_2 : (v_1, \dots, v_n)) \\ &= G_1(v_1, \dots, v_n) : G_2(v_1, \dots, v_n) \\ &= G(v_1, \dots, v_n). \end{aligned}$$

(iv) If $G(v_1, \dots, v_n) = G_1(v_1, \dots, v_n)\ *\{m\}\ G_2(v_1, \dots, v_n)$ then by induction step, in a similar way to above, we have $f = f_1\ *\{m+n\}\ f_2$.

The uniqueness of f follows directly from the (EXT) rule.

COROLLARY. *Jaf is an extensional combinatory algebra* (cf. Barendregt, 1981; Meyer, 1982).

A combinatory algebra can be constructed explicitly in *Jaf* because it is sufficient to isolate two different elements K and S of *Jaf* such that for all a, b, c $\in \mathcal{A}$, (K : a) : b = a and ((S : a) : b) : c = (a : c) : (b : c). The existence of elements K and S in an algebraic structure guarantees solutions to a finite set of combinatorial equations (Barendregt, 1981), but the process of finding these solutions in terms of S and K is unacceptable from a computational point of view, because it leads to a combinatorial growth in the size of resulting variable free expressions (Turner, 1979a; also Burton, 1983). The above proof of the completeness theorem demonstrates a practical algorithm which performs abstraction on combinatory terms with respect to all specified variables in a single step. This is possible because of the given definition of the *Jaf* algebra. The algorithm also retains information from functional terms within combinatory expressions that enables a parallel reduction of the expressions. Our algorithm is similar but more direct then the one presented by Abdali (1976), and it yields compact and algebraically structured combinatory expressions.

A special case of a combinatorial equation is recursive definition. One can find the representation of a recursively defined function in *Jaf* using fixed point functionals (Barendregt, 1981).

DEFINITION. *A recursive definition over Jaf* is a finite system of equations of the form:

$$v_i = G_i ,$$

where G_i is a term in n variables from the set of variables $\{v_1, ..., v_i, ..., v_n\}$.

DEFINITION. *A fixed point functional* is a functional $\mathcal{Y}$ such that for any function g

$$\mathcal{Y}(g) = g(\mathcal{Y}(g)),$$

i.e. $\mathcal{Y}(g)$ is a fixed point of g.

THEOREM (Fixed Point). *Let* Z = S{2,2} *{2} ((S{1,2} *{2} S{1,2}) *{2} S{2,2}). *Then the element* Y = (Z, Z) *of* Θ *represents a fixed point functional* $\mathcal{Y} \in \mathcal{F}un$.

PROOF. For any $f \in \Theta$

$$Y : f = \mu(Z)_Z(f) = S\{2,2\}_Z : (f)\,((S\{1,2\}_Z(f)) : (S\{1,2\}_Z(f))) : (S\{2,2\}_Z(f)))$$
$$= f : ((Z, Z) : f) = f : (Y : f).$$

PROPOSITION (Double Fixed Point, Barendregt (1981)). *Consider the following system of equations in Jaf:*

$$v_1 = G_1(v_1, v_2)$$
$$v_2 = G_2(v_1, v_2).$$

We show that there exists $f_1, f_2 \in \Theta$ *such that*

$$f_1 = G_1(v_1/f_1, v_2/f_2)$$
$$f_2 = G_2(v_1/f_1, v_2/f_2),$$

i.e.

$$f_1 = g_1 : (f_1, f_2)$$
$$f_2 = g_2 : (f_1, f_2).$$

PROOF. Define $\chi(f_1) = \gamma(g_2 : f_1) = (Y \bullet g_2) : f_1$ so that $f_2 = \chi(f_1)$. Thus

$$f_1 = g_1 : (f_1, \chi(f_1)) = (g_1 * (Y \bullet g_2)) : f_1 = \gamma(g_1 * (Y \bullet g_2))$$

and

$$f_2 = \chi(f_1) = (Y \bullet g_2) : \gamma(g_1 * (Y \bullet g_2)).$$

Now let us verify the above choice of f_1 and f_2

$$f_1 = \gamma(g_1 * (Y \bullet g_2)) = (g_1 * (Y \bullet g_2)) : \gamma(g_1 * (Y \bullet g_2)) = (g_1 * (Y \bullet g_2)) : f_1$$
$$= g_1(f_1)\,((Y \bullet g_2)(f_1)) = g_1 : (f_1, \chi(f_1)) = g_1 : (f_1, f_2)$$

$$f_2 = (Y \bullet g_2) : \gamma(g_1 * (Y \bullet g_2)) = (Y \bullet g_2) : f_1 = \gamma((g_2, f_1)) = g_2 : (f_1, \gamma((g_2, f_1)))$$
$$= g_2 : (f_1, f_2).$$

THEOREM (Multiple Fixed Point). *There exists* $f_1, f_2, \ldots, f_n \in \Theta$ *such that*

$$f_1 = g_1 : (f_1, f_2, \ldots, f_n)$$
$$f_2 = g_2 : (f_1, f_2, \ldots, f_n)$$
$$\ldots$$
$$f_n = g_n : (f_1, f_2, \ldots, f_n),$$

where $g_i \in \Theta$ *corresponds to a term* $G_i(v_1, \ldots, v_i, \ldots, v_n)$ *over Jaf.*

PROOF. Let us define

$$\chi(f_1, f_2, \ldots, f_{n-1}) = \gamma(g_n : (f_1, f_2, \ldots, f_n)) = (Y \bullet\{n-1\} g_n) : (f_1, f_2, \ldots, f_{n-1})$$
$$\chi(f_1, f_2, \ldots, f_{n-2}) = \gamma((g_n *\{n-1\} (Y \bullet\{n-1\} g_n)) : (f_1, f_2, \ldots, f_{n-2}))$$
$$= (Y \bullet\{n-2\} (g_{n-1} *\{n-1\} (Y \bullet\{n-1\} g_n))) : (f_1, f_2, \ldots, f_{n-2})$$
$$\ldots$$
$$\chi(f_1) = \gamma((g_2 *\{2\} (Y \bullet\{3\} (\ldots (Y \bullet\{n-2\} (g_{n-1} *\{n-1\} (Y \bullet\{n-1\} g_n)))\ldots))) : f_1)$$
$$= (Y \bullet (g_2 *\{2\} (Y \bullet\{3\} (\ldots (Y \bullet\{n-2\} (g_{n-1} *\{n-1\} (Y \bullet\{n-1\} g_n)))\ldots))) : f_1$$

so that

$$f_n = g_n : (f_1, f_2, ..., f_{n-1}, f_n) = \mathcal{X}(f_1, f_2, ..., f_{n-1})$$

$$f_{n-1} = g_{n-1} : (f_1, f_2, ..., f_{n-1}, \mathcal{X}(f_1, f_2, ..., f_{n-1}))$$
$$= (g_{n-1} *\{n-1\} (Y \bullet\{n-1\} g_n)) : (f_1, f_2, ..., f_{n-2}, f_{n-1})$$
$$= \mathcal{X}(f_1, f_2, ..., f_{n-2})$$

...

$$f_2 = \mathcal{X}(f_1)$$

$$f_1 = \mathcal{Y}(g_1 * (Y \bullet (... (Y \bullet\{n-2\} (g_{n-1} *\{n-1\} (Y \bullet\{n-1\} g_n)))...))).$$

The proofs of the above theorem demonstrate a simple mechanical method for finding uniform representations (fixed points) of functions defined by mutually recursive definitions. This method is particularly suitable for implementations of symbolic computation because expressions which represent the fixed points are obtained by symbolic manipulations without using any auxiliary functions such as tupling (Barendregt, 1981) or environment (Williams, 1981). They can be computed while recursive definitions are partially supplied, and the full representations are obtained when the last mutually recursive definition is given. Therefore, any of the mutually recursive functions can be used and evaluated independently of the others from its full fixed point representation in *Jaf*.

COROLLARY. *Recursive definitions over Jaf have (extensionally unique) solutions (fixed points) in Jaf which can be found uniformly.*

The canonical fixed points correspond to least fixed points of a structure ordered by approximations (c.p.o.) (Barendregt, 1981). Therefore, simple computational induction and algebraic manipulations can be combined to infer properties of recursively defined functions.

Now, let us enrich *Tup* by numerals without worrying about the representation of a numeral system in *Jaf*. The existence of a numeral system in *Jaf* is guaranteed by combinatory completeness (Barendregt, 1981) and it can be constructed in a number of different ways (Bunder, 1981; van der Poel *et al.*, 1980). We simply add new elements called numerals, $\mathcal{N}um \subset Tup$, and arithmetic functions, elements of *Fun*.

DEFINITION. Let *numerals*, $\mathcal{N}um \subset Tup$, be represented by distinct new elements $n \in Tup$ for $n = 0,-1,1,-2,2,...$. Let $\mathcal{ADD}\{i\}$ (addition functions), $\mathcal{SUB}\{i\}$ (subtraction functions), $\mathcal{EQU}$ (the equality test function) be new elements of *Fun* defined for all $(n_1, ..., n_i) \in \mathcal{N}um^i$, $i \geq 1$, as follows:

(NUM1) $\mathcal{ADD}\{i\}(n_1, ..., n_i) = n_1 + ... + n_i$

(NUM2) $\mathcal{SUB}\{i\}(n_1, ..., n_i) = n_1 - ... - n_i$

(NUM3) $\mathcal{EQU}(n_1, n_2) = T$ if $n_1 = n_2$, otherwise F

We can now define, using recursive definitions, and represent, using the fixed point operator $\mathcal{Y}$, all computable functions in *Jaf* (Kleene, 1936).

4. Implementation of JAM

This section describes an interpreter and functional simulator of a combinator machine called *JAM*achine. JAM is based on *Jaf* algebra defined in section 2 and it is an algebraically structured architecture (Thatcher *et al.*, 1981). It supports both serial and parallel processing for functional programs that manipulate symbolic streams. A functional program is first translated into a combinator code using our abstraction algorithms, section 3. Combinator code is further translated into a directed graph for execution on JAM. The execution of a graph takes place by communicating streams of graph pointers, which represent argument and result subexpressions, along graph arcs representing data dependencies between the combinators within the combinator code. The memory organization of JAM supports sharing of subgraphs through graph pointers, which in turn avoids re-evaluation of separate copies of common arguments. JAM executes functional programs in a data flow fashion but it applies sequential graph reduction for graphs, or subgraphs, which are inherently sequential. This makes JAM a useful cross between a data flow machine and graph reduction machine (Treleaven, 1984) for the purpose of efficient symbolic computation.

4.1. PROGRAMMING ON JAM

JAM is programmed using a reduction language called *Jal*, which is founded on functional expressions that are formed using recursive definitions or recurrence relations (viewed as simple iterations) over the domain of arbitrarily long tuples (viewed as streams).

The expressions are interpreted in the cartesian closed universe *Ccu*(*Tup*, *Fun*). The elements of *Tup* are called *functional objects*, or simply *objects*, and the elements of *Fun* are functions which manipulate objects. *Constant* objects such as numerals and quoted character strings ("abc"), and *simple names*, which are non-empty strings of symbols, are distinguished from the other objects and they are called *atomic objects*. Simple names that start with alphabetic characters may represent functions or objects and simple names that start with non-alphabetic characters are reserved for representing operators. A simple name may represent a functions/objects/operators (1) directly, e.g. the name s represents the *s* functions, (2) through a definition, e.g. DEF True = T, or (3) by combining both previous methods, e.g. (S{1,2}, true).

A name that represents a function/object/operator is called function/object/operator name, or just *name* for short. A name can be associated with any valid *Jal* expression by the definition of the form:

DEF Func_name_expression = expression;

where *Func_name_expression* is a *Jal* expression which starts with the name of a function or an object, and than may be followed by variables. All *Jal* operators are infix operators, i.e. (a & b), which are defined and named in the following general way:

OPR Opr_name_expression = expression;

where *Opr_name_expression* is a *Jal* expression involving the name of an operator and possibly variables.

A definition in *Jal* can be *simple*, *functional* or *indexed*. A simple definition associates a *Jal* expression directly to a name. If variables are used to define a function, object or operator then the definition is called a functional definition. For example, the following definition

DEF (Const, x) : y = x;

specifies that *Const* represents a functional of arity one which takes its first argument, x, to produce a constant function. An indexed definition may take the form of either simple or functional definition. The difference from the other two types of definitions is that an indexed definition is an inductive definition which defines an indexed class rather then a single function, operator or object.

A class is indexed by *index expressions* in variables ranging over natural numbers. An index expression can only be a simple arithmetic expression over natural numbers. A simple arithmetic expression is an addition or subtraction of a constant to/from a indexed variable (e.g. i+1 or j-5). However, there are the following restrictions: (1) the same index variable name cannot occur more then once on the left hand side of the indexed definition, (2) simple arithmetic expressions involving index variables cannot occur on the left hand side of the definition. A name immediately followed by an index expressions, which are separated by commas and enclosed in a pair of braces is called *index name* (e.g. Ind_name{i+1,j-5}).

An indexed definition is usually combined with a simple definition to specify the initial elements of the class. An example of an indexed definition is a definition of an indexed class of functional objects, *Fib_number*{i} (i=1,2,..), which corresponds to the chain of Fibonacci numbers as they increase with the index i:

```
DEF Fib_number{i} = ADD{2} : (Fib_number{i-1}), Fib_number{i-2});
DEF Fib_number{2} = 1;
DEF Fib_number{1} = 1;
```

Because the *Jal* language has an associated partial algebra of functions *Jaf* it is possible to model a partial equivalence predicate in *Jal*. The partial equivalence function $\mathcal{EQV}$ applied to two functions results: T if the functions can be proved equivalent in *Jaf*, F if they can be disproved equivalent in *Jaf*, and it is indeterminate otherwise.

The *undefined value* can be simply represented in *Jaf* by the element UV = (Y, T). The element UV has a dual purpose, for it represents the undefined value as well as the everywhere-undefined function $\mathcal{UV}$, because

$$\mathcal{UV}(a) = \mathcal{Y}_T\,(a) = S\{1,2\}(\,(Y, T), a) = (Y, T) = UV.$$

A strict function *f*, i.e. *f* (UV) = UV, can now be expressed in *Jaf* using $\mathcal{EQV}$ function which tests an argument to be UV. For example, a strict version of fixed point functional $\mathcal{Y}$, denoted $\mathcal{YUV}$, can be expressed in *Jaf* as

YUV = (T, (EQV, UV)) *{2} Y *{2} UV;

A function can also be defined strict by the *strict definition* in *Jal*:

STRICT function_name;

This concludes this short introduction to the *Jal* language. Section 4.4 describes *Jal* and JAM in more details.

We end this section by demonstrating how to define recursive lists in *Jal* which we will extend in the next section to the complete embedding of Backus' FP into *Jal*, see Figure 1 and 2.

We define an indexed class of pairing operators (Church, 1941) ^{n} in the following way (Gibert, 1983b):

```
OPR a ^{n} b : x  = (a:x) ^{n-1} (b:x);
OPR a ^ b : x = x : a : b;
```

One can use these pairing operators to inductively define finite lists and r.e. infinite lists. For example, if (1 ^ (2 ^ (3 ^ UV))) is an object that represents the list of three numbers "<1,2,3>" (UV represents the empty list) then the list manipulating functions *head* and *tail* (Hd, Tl in Figure 1) can be constructed using the function *swap* defined as follows:

```
DEF Swap = S{2,2} * S{1,2};
```

i.e. Swap : (a, b) = (b, a). Then

```
Hd : (a ^ b) = (Swap, T) : (a ^ b) = a
```

and

```
Tl : (a ^ b) = (Swap, F) : (a ^ b) = b.
```

An indexed class of constructors for recursive lists, List{i}, i = 1,2,..., can be now defined as

```
DEF List{n}  = s{1,n} ^{n+1} (T, List{n-1});
DEF List{1}  = I ^{2} UV;
```

i.e. List{3} : (a, b, c) = (1 ^ (2 ^ (3 ^ UV))).

4.2. FUNCTIONAL PROGRAMMING ON JAM

The universality and the expressive power of the *Jal* language guarantees that one can run efficiently any "high level" functional programming system based on lambda calculus on JAM. This will be exemplified by the following simple embedding of Backus' FP (Backus, 1978) into *Jal*. The embedding of FP into *Jal* preserves the algebraic structure of FP and at the same time it allows us to extend FP by (i) permitting programs to manipulate infinite sequences (this may make some applications easier to program), (ii) including infinite expansions for recursively defined functions into the algebra of programs.

Firstly consider the data structures of FP. A set Ob of objects is built recursively from atoms, At, $\perp$ (the undefined object) and objects by the n-ary list constructor, '< ... >', i.e.

$$Ob = At \cup \{\perp\} \cup \{<ob_1, ..., ob_n> \mid ob_1, ..., ob_n \in Ob\}.$$

We define the embedding ε to be a bijection from *Ob* to *Tup* as follows

$$\varepsilon(\perp) = \mathrm{UV} \text{ and } \varepsilon(<>) = \mathrm{UV},$$

$$\varepsilon(\mathit{true}) = \mathrm{T} \text{ and } \varepsilon(\mathit{false}) = \mathrm{F},$$

$$\varepsilon(n) = \mathrm{n} \text{ where } n \in \mathit{Int} \text{ and } \mathrm{n} \in \mathit{Num},$$

$$\varepsilon(a) = \mathrm{a} \text{ where } a \in \mathit{At} \text{ and } \mathrm{a} \in \mathit{Tup} \text{ is atomic},$$

$$\varepsilon(<ob_1, ..., ob_n>) = \mathrm{List}\{\mathrm{n}\} : (\varepsilon(ob_1), ..., \varepsilon(ob_n)) = \varepsilon(ob_1) \wedge (... \wedge \varepsilon(ob_n)...).$$

We extend Ob to $Oinf$ by adding infinite lists of objects, i.e. $Oinf = Ob \cup Oinf^{\omega}$ where $Oinf^{\omega}$ can be represented by total recursive functions $Int \rightarrow Oinf$ which are mapped on functional objects defined in *Jal* which correspond to r.e. lists. For example, an infinite list of all Fibonacci numbers can be constructed using the following recursive definition in *Jal*

DEF (Fib_list, n) : m = n ^ ((Fib_list, m) : ((ADD, n) : m));

which can be represented by a generator, *Fib_gen*, directly defined in *Jal* as

DEF Fib_gen = (Y, S{2,3} ^{4} (T *{3} (T, ADD)) * (T, 0));

i.e. Fib_gen : 1 = <0, 1, 1, 2, 3, 5, ...>.

Now, consider an application operation in FP, ": ", which applies a FP function *f* to an object *ob*. We define $\varepsilon(f : ob) = \varepsilon(f) : \varepsilon(ob)$ where $\varepsilon(f)$ is shown in Figure 1 and 2 and $\varepsilon(ob)$ is defined above.

4.3. ALGEBRA OF FUNCTIONAL PROGRAMS ON JAM

As in general mathematics, the distributivity of one function over another is an important notion of the *Jal* language. It permits convenient representations of infinite expansions for recursively defined functions (e.g. conditional expansions, see below), or it provides optimization transformations for functional programs. For instance, it immediately follows

```
DEF Id = I;
DEF Hd = (Swap, S{1,2});
DEF Tl = (Swap, S{2,2});
DEF Select{i} = Hd • Selectl{i};
DEF Selectl{i} = Tl • Selectl{i-1};
DEF Selectl{1} = Id;
DEF Appendl = Hd ^{2} (Hd • tl);
DEF Not = Id * (T, F) * (T, T);
DEF Eq = (T, EQU) * Hd * ( Hd • Tl );
/* recursive reviota */
DEF Rrev = Id ^{2} ( Rrev • (SUB, 1) );
/* fixed point reviota */
DEF Rrev = (YUV, (T, Id) ^{3} (S{1,2} *{2} (T, (SUB, 1))) );
```

Fig. 1. Some FP functions in *Jal*

```
DEF (Const, f) = (T, f);

DEF (Bu, f, x) = f • ( (T, x) ^{2} Id);
/*
    (p -> q ; r) is denoted by (if, p, q, r) where
    (if, p, q, r) : x = q : x   if (p : x) evaluates to T,
    (if, p, q, r) : x = r : x   if (p : x) evaluates to F,
    (if, p, q, r) : x = UV      if (p : x) evaluates to UV
*/
DEF (If, p, q, r) = p * q * r;
/*
    [f1,..., fn] is denoted by (confun{n}, f1,..., fn)
*/
DEF Confun{n} = S{1,n-1} ^{n+1} (T, Confun{n-1});
DEF Confun{1} = (T, UV);
/*
    alfa f is denoted by (Apply_all, f)
*/
DEF (Apply_all, f) = (f • Hd) ^{2} ((Apply_all, f) • Tl);
STRICT Apply_all;
/*
    /f is denoted by (Insertl, f)
*/
DEF (Insertl, f) = f • (hd ^{2} ( (Insertl, f) • Tl));
STRICT Insertl;
```

Fig. 2. Some FP functionals in *Jal*

from the proposition in section 2. that the following two important distributivity laws of Backus' Algebra of Programs (Backus, 1978) are preserved in *Jaf*:

$$(\text{Confun}\{n\}, f_1, ..., f_n) \bullet g = (\text{Confun}\{n\}, f_1 \bullet g, ..., f_n \bullet g)$$

$$((p * q) * g) \bullet h = ((p \bullet h) * (q \bullet h)) * (g \bullet h).$$

It is a straightforward procedure to verify that the whole of Backus' Algebra of Programs is preserved in *Jaf*. In fact we have carried out most of the proofs with the assistance of JAM (Gibert, 1984b). The algebraic roots of *Jal* allow *Jaf* to be used as the meta-level to prove properties of *Jal* programs. The $\mathcal{EQV}$ function, which is used to model the equality predicate of FP, adds extra power to FP since it uses *Jaf* to verify program equivalences.

Now we demonstrate how to extend the laws of Backus' Algebra of Programs involving a conditional functional to include infinite conditional expansions (Williams, 1982):

$$f = (p_0 * q_0) * (\mathcal{H}(f)) = (p_0 * q_0) * ((p_1 * q_1) * ((p_2 * q_2) * ...)),$$

where $\mathcal{H}$ is a functional in *Jaf* such that for all h

$$\mathcal{H}((p_i * q_i) * h) = (p_{i+1} * q_{i+1}) * (\mathcal{H}(h)).$$

For example, one of the laws noted above can be rewritten as follows:

$$((p_0 * q_0) * ((p_1 * q_1) * \ldots)) \bullet h = ((p_0 \bullet h) * (q_0 \bullet h)) * (((p_1 \bullet h) * (q_1 \bullet h)) * \ldots),$$

because

$$\begin{aligned} f \bullet h &= (Y, (T, p * q) *\{2\}\ H \bullet h) \\ &= ((p * q) * (\mathcal{H}(Y, (T, p * q) *\{2\}\ H))) \bullet h \\ &= ((p \bullet h) * (q \bullet h)) * (((p_1 * q_1) * (\mathcal{H}(\mathcal{Y}((T, p * q) *\{2\}\ H)))) \bullet h) \\ &= ((p \bullet h) * (q \bullet h)) * (((p_1 \bullet h) * (q_1 \bullet h)) * \ldots] \\ &= ((p \bullet h) * (q \bullet h)) * (\mathcal{H}(Y, T \bullet ((T, p * q) *\{2\}\ (T \bullet H))) *\{2\}\ (T, h)) \\ &= (Y, T \bullet ((T, p * q) *\{2\}\ T \bullet H)) *\{2\}\ (T,h)). \end{aligned}$$

4.4. IMPLEMENTATION OF JAM

JAM(achine) is principally a combinatory graph reduction machine. *Jal* expressions are translated into "free-from-bound-variables" (pure) *combinatory code* through an implementation of our abstraction algorithms described in section 3. Ordinary variables are abstracted using the combinatory completeness result, and simple recursive and mutually recursive variables are abstracted using the fixed point results. The execution of the code takes place through a sequence of graph reductions that transform the *Jal* expressions into their meaning (the elements of $Ccu(\mathcal{F}un, \mathcal{T}up)$).

4.4.1. EVALUATION MODES

The standard evaluation mode for a *Jal* expression is *lazy* (or *normal order*), i.e. evaluation of an expression is leftmost in respect to the application operation, and evaluation of a subexpression is delayed until it becomes the *leftmost*. The leftmost component of an expression determines the reduction rule or the definition simplification to be applied. All required arguments are consumed and a new expression which represents the result of applying the particular rule or simplification is substituted in the place of the old expression. If no reduction or simplification can be applied the lazy evaluation terminates.

However, the value of a non-leftmost subexpression may be required by JAM either during the execution of a program (e.g. by a primitive arithmetic function) or after the execution terminates (for example, to print a result). In the first case, the normal order evaluation of an expression is postponed while the argument subexpressions are executed. If the results of the evaluation of subexpressions are of the required type (e.g. a number) then the execution proceeds, otherwise the execution terminates. This type of evaluation is called *eager* (or *innermost order*) and is known to be not safe semantically when used exclusively (Eick & Fehr, 1983). However, if we restrict eager evaluation to the following two cases, we are able to preserve the simple algebraic model and maintain efficiency of execution of functional programs: (1) the execution of *strict* functions, e.g. arithmetic or relational functions, (2) the execution of the *pipe-apply* operator.

The *pipe-apply* operator "%" is a functional (module) composition operation naturally supported by the architecture of JAM which is brought to

the language level to allow a functional program to be built out of simple combinations of interconnecting subprograms, cf. Figure 3. The pipe-apply operator is defined as follows

$$(f_1 \,\%\, f_2) : (a_1, ..., a_n) = f_1\ (f_2\,(a_1, ..., a_n)).$$

In other words, the pipe-apply operator applies a function f_2 to a tuple $(a_1, ..., a_n)$ and at the same time routes the output of f_2 to the input of f_1. In particular, if f_2 denotes a functional object (e.g. Fib_number in section 4.1) then the pipe-apply operator computes f_2 before it is applied to f_1.

```
    J-Machine 1.3 (MAY 86) - type "help;" for help
 1> /*
 1>     A program which sorts stream of n numbers
 1>
 1>     Max_to_left shifts greatest number of n
 1>     numbers of a tuple to the left of the tuple
 1> */
 1> DEF Max_to_left{n} =
 1>            GT
 1>              *{2} ((Max_to_left{n-1} % S{1,2}) *{n} S{2,n})
 1>              *{2} ((Max_to_left{n-1} % S{2,2}) *{n} S{1,n})
 1> ;
 2> DEF Max_to_left{1} = I;
 3> /*
 3>     Many_swap shifts first element of a n-tuple
 3>     to the last position in the tuple
 3> */
 3> DEF Many_swap{n} = S{2,2} *{n} S{1,n};
 4> DEF Many_swap{1} = I;
 5> /*
 5>     Main program
 5> */
 5> DEF Sort{n} = Sort{n-1}
 5>                          % Many_swap{n}
 5>                          % Max_to_left{n}
 5> ;
 6> DEF Sort{1} = I;
 7> ARGS Sort{n} = n;
 8> /*
 8>     Execution
 8> */
 8> Sort{10} : (3,1,6,5,9,2,4,8,7,0);

    (0,1,2,3,4,5,6,7,8,9)
 9> ^D
    End of session.
```

Fig. 3. Example program.

JAM transforms a *Jal* expression into its full (not lazy) meaning in *Ccu(Fun, Tup)*. An expression is normal order evaluated to produce a lazy resul. If the evaluation terminates, then the result is interpreted in *Ccu(Fun, Tup)*. The interpretation (meaning) in *Ccu(Fun, Tup)* is obtained by recursively applying normal order evaluation to each immediate subexpression of the result of the evaluation.

4.4.2. EQUIVALENCE PROOFS

JAM uses lazy evaluation when proving equivalences between *Jal* expressions. Lazy evaluation guarantees correctness of a proof. A proof is performed through the analysis of symbolic execution of functional expressions during which comparisons and evaluations are applied alternately. Before any evaluation takes place, the expressions are compared for identity. If they are not identical, they are lazily evaluated. Then, the proof continues recursively while the resultant expressions are compared subexpression by subexpression.

If a disagreement occurs between the evaluations at any stage of the proof, the data base of equivalences is consulted. If, after using all possible equivalences, the evaluations are still not equivalent then the proof fails, otherwise the proof continues until the evaluations fully terminate.

```
    J-Machine 1.3 (MAY 86) - type "help;" for help
1>  /*
1>      A simple inductive equivalence
1>  */
1>  DEF A{n} = A{n-1};
2>  DEF A{1} = B;
3>
3>  TRACE PROVE ON{i}   A{i} = B;
    Induction basis:
              LHS:   Start sub-reduction of    ( A )
            ( A )    definition    =>          ( B )
              RHS:   Start sub-reduction of    ( B )
    Proof completed.
    Induction hypothesis:
              LHS:   Start sub-reduction of    ( A{i} )
              RHS:   Start sub-reduction of    ( B )
    Induction step:
              LHS:   Start sub-reduction of    ( A{i+1} )
            ( A )    definition    =>          ( A{i} )
              RHS:   Start sub-reduction of    ( B )
    Proof completed using induction hypothesis.
          ( A{i} )   is equivalent to          ( B )
9>  ^D
    End of session.
```

Fig. 4. Example of an induction proof.

An inductive equivalence is proved in two separate stages: the base stage and the induction stage. In the base stage the index variables of a proof are assumed to have the value 1 on both sides of the equivalence and the proof proceeds as described before. In the induction stage, the induction hypothesis is assumed as one of equivalences, and the index variable of a proof is substituted by its successor on both sides of the equivalence. Formal indices are changed after the first reduction to *inductive formal indices* which cannot be further used in the execution.

For example, consider the induction proof in Figure 4. When JAM attempts to reduce A{i+1}, the simplification by definition is applied once and it results in A{i}. However, now the index {i} is an inductive formal index, the execution terminates without going into an infinite loop, and the proof proceeds as described before.

4.4.2. CODE GENERATION

The *Jal* combinatory code representing a functional program can be executed on a sequential reduction machine such as the GMD machine (Berkling, 1975) or it can be further translated to low level code capable of running on other reduction architectures, e.g. SKIM (Clarke *at. el.*, 1980) or NORMA SASL (Turner, 1981).

For execution on JAM, a *Jal* expression is translated into a directed graph composed of two types of nodes. The first type of node is called *function node* and it represents a primitive or defined *Jal* function, see Figure 5. The second type of node is a *subexp node* for representing *Jal* subexpressions.

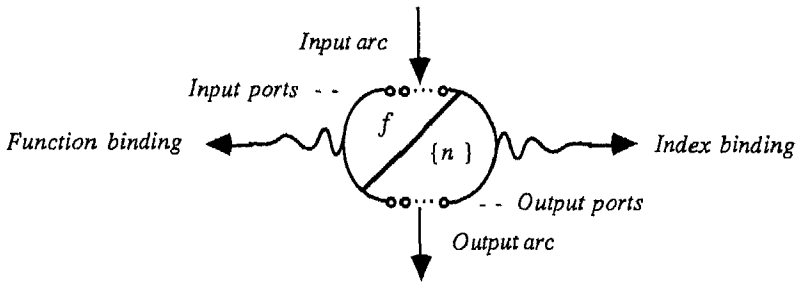

Fig. 5. Node representation of a function

Both types of nodes contain the following two pointer fields (Figure 5):

(1) a function or subexpression graph binding which points to the graph containing the body of the corresponding function or subexpression,

(2) an index binding which points to the global *definition table* entry for a function or subexpression, where actual indices and other local data are kept.

Arcs correspond to application operations through which arguments and results are passed between functions. Therefore a graph node corresponding to a *Jal* function has a single arc which delivers an argument graph pointer

or an argument stream, and a single arc for the result graph pointer or the result stream, Figure 5. However, there may be more than one input or output port to a node. Nodes for functions with parameters possess a single value input port for each parameter and a multiple value input port for other arguments. Ports for a node are ordered from "left to right". Arguments are gathered and assigned to parameter ports in the order of arrival on the input arc. A node can start consuming other arguments only after all parameter ports are filled. The order of results on the output arc is determined by the order of the output ports.

The ARGS definition of *Jal* specifies the number of parameters (arity) for a function, which in turn specifies the number of ports to be used in the corresponding function node. For example, an indexed class of *Jal* functions, f{n} (n=1,2,...), with the following ARGS definition

$$\text{ARGS } f\{n\} = n;$$

corresponds to the node templets with n-1 parameter ports plus one argument port. This feature gives a very fine control over the running of JAM, because it makes possible to condition the execution of function on availability of all parameters, cf. Figure 3. Infix operators of *Jal*, which are single nodes, have two extra parameter ports for the operands. If operands are present then the operator node is an ordinary function nodes, cf. *{2} node in Figure 6.

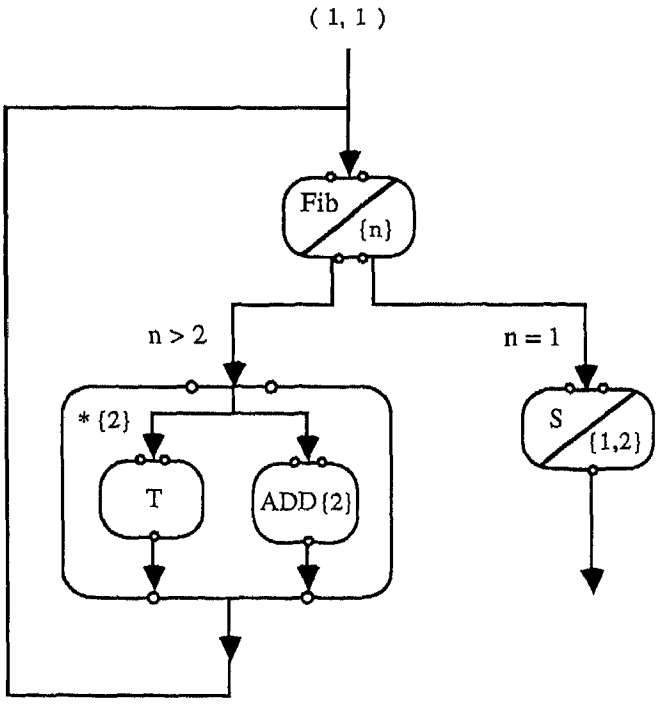

Fig. 6. Graph Representation of Fibonacci Function

EXAMPLE. We will illustrate code generation on the following single recursive definition of Fibonacci numbers:

```
DEF Fib_number{n} = Fib{n} : ( 1, 1 );
DEF Fib{n} = (T, Fib{n-1}) *{2} ADD{2};
DEF Fib{1} = S{1,2};
```

A graph which is produced from this definition is shown in Figure 6. All nodes are marked redex, and with the index n being a positive integer, e.g. Fib{5}, this is a pure data flow graph.

4.4.3. EXECUTION

The execution of a *Jal* expression on JAM is a combination of data flow and reduction, cf. (Treleaven, 1984). That is, the flow of data itself does not transfer control from one function node to another, as is the case in standard data flow (Some nodes, e.g. functions such as Fib_number{n} defined in section 4.1, might not have input or output arcs). Instead an independent control transfer takes place according to a global definition table. Availability of data arguments in a function node is not in itself sufficient to initiate execution of this node and the function needs to be marked *redex* in the global definition table entry for this function for the execution to proceed. Therefore, JAM treats a graph as the specification of the partial ordering of reduction sequences.

Function nodes or subexpression graphs marked in the global definition table as redexes are executed, the intermediate results are stored as pointers to new nodes or graphs, or as data values or index values in the global definition table. The execution of functions and subexpressions, and the change to corresponding indices from formal to actual in functions and subexpressions is delayed until the corresponding nodes are required to produce values. This technique results in considerable space and time improvements over pure data flow as it permits the immediate distribution of arguments to any destination in the graph, and at practically no cost since most reductions of *Jal* expressions are simply link manipulations. Consider, for example, the "composition" operator "•" which has been defined in section 2. The reduction of the *Jal* expression (f • g) : x is as follows

```
(1)   ( f • g ) : x  →  ( (T,f) * g ) : x
(2)                  →  ( (T,f) : x' ) ( g (x) )
(3)                  →  f ( g (x) )
```

Execution of the "*" operator in the first step creates a pointer node x' which points to the value of the data argument x. Then the reduction of the T combinator in the second step destroys the link x'. Therefore the whole reduction (steps 1 to 3) does not incur any overhead of coping, transmitting and possibly re-evaluated of x unnecessarily.

4.4.4. PERFORMANCE

We have made a prototype implementation of JAM interpreter and functional simulator in the C language under the UNIX 4.2BSD operating

system on VAX 11/780 at the University of Melbourne (Gibert, 1984b). The interpreter runs as a single system process but uses a fixed number of child subprocesses and pipelines with which it emulates parallel execution.

The preliminary performance analysis of JAM interpreter has been very encouraging. We have run a number of simple benchmarks to compare the time taken by our interpreter (total elapsed time under a light system load) to execute simple functional programs, e.g. Fibonacci function, with the time taken to execute the same programs by a conventional LISP interpreter (Foderaro *at. el.*, 1983). We considered only the execution time, as the translation time in our system is negligible compared to the time taken to perform evaluations. In all cases our interpreter, using only four subprocesses, was more than one order of magnitude faster than the LISP interpreter. Since our interpreter used very simple data structures and used the standard system-supplied process and memory management package which forms a bottleneck in the interpreter (up to 65% of the processing time was spent on process and memory management), it compares even more favorably with the LISP interpreter which utilizes sophisticated memory management facilities. However, it remains to be seen what performance can be achieved for larger programs as we would need to implement a complete compiler for the figures to be meaningful.

Conclusions

In this paper we aimed to construct an optimal algebraic system, which would have the power to accommodate any functional programming language inspired by the lambda calculus approach to the treatment of computable functions, and which could be used as a basis for an efficient machine architecture to implement symbolic computation.

JAM(achine) is designed in accordance with the algebraic construction of our combinatory system, the *Jaf* algebra. This avoids an additional metalevel needed for proving properties of programs and allows JAM to assist a programmer in carrying out proofs and program transformations. The combinators provide for machine instructions of a possible architecture but, at the same time, they are easily accessible to programmers.

One issue that has not been covered in the paper is that of finer-grain parallelism. The combinator code of a *Jaf* expression already possesses properties which make it possible to use very fine-grained parallelism in execution of functional programs, i.e. distributivity and freedom from bound variables, but there are at least three ways to further enhance inherent concurrency of the combinatory code. Firstly, it is possible to reduce all argument subexpressions in parallel, which could be considered as separate processes, after the analysis of argument dependencies between combinators in the code. For example, arithmetic (strict) operations already demand the evaluation of their arguments in parallel. Secondly, our graph, which represents the structure of a program, can be easily spread across many parallel processors similarly to data flow approach. Lastly, explicit parallel control operators such as *fork* could be introduced to the *Jaf* language. It is an interesting question as to which of these methods would best lead to the highest degree of concurrency in functional/symbolic computation. Some

work in this direction has already been started (Hudak & Goldberg, 1985; Halstead, 1985; Burton, 1984; Maurer & Oldehoeft, 1983; Maurer & Oberhauser, 1985) and parallel machines have been built (Mago, 1982; Kluge, 1983; Buchberger, 1984), but so far there is no parallel reduction machine that uses combinators to implement symbolic computations.

Work is currently in progress aimed at emulating a fine-grained parallel JAM on a data flow computer. Furthermore, recent developments in the hardware of data flow and parallel reduction machines are making a practical hardware implementation of a combinator based symbolic computation system such as ours feasible and very appealing.

Acknowledgements

Valuable suggestions by Bruno Buchberger and thoughtful comments from the referees have materially improved the presentation and the content of this work from the initial manuscript, and are gratefully acknowledged. Thanks to Dianne McKerrow who has helped greatly with preparation of the manuscript.

References

Abdali, S. K. (1976). An Abstraction Algorithm for Combinatory Logic. *Journal of Symbolic Logic* **41**,. 1, pp. 222-224.

Amamiya, M., Hasegawa, R. (1984). Dataflow Computing and Eager and Lazy Evaluations. *New Generation Computing* **2**, pp. 105-129.

Backus, J. (1978). Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *CACM* **21**, 8, pp. 613-641.

Backus, J. (1981). The Algebra of Functional Programs: Function level reasoning, linear equations and extended definitions. *Proceedings of the International Symposium on the Formalization of Programming*, Peniscola, Spain, April 1981. *Lecture Notes in Computer Science* **107**, Springer Verlag, pp. 1-43.

Barendregt, H. P. (1981). The Lambda Calculus, Its Syntax and Semantics. *Studies in Logic* **103**, North-Holland.

Berkling, K. J. (1975) Reduction Languages for Reduction Machines. *Proceedings of the IEEE International Symposium on Computer Architecture*, January 1975, pp. 133-140.

Bohm, C. (1982) Combinatory Foundation of Functional Programming. *ACM Symposium on Lisp and Functional Programming*, June 1982.

Buchberger, B. (1984) The Present State of the L-Network Project. *Proceedings of Mini and Microcomputers and their Applications 84*, Acta Press, Anaheim, pp. 178-181.

Bunder, M. W. (1981) Natural Numbers in Illative Combinatory Logic. *Proceedings of the 5th Latin American Symposium on Mathematical Logic*, Bogota, *Lecture Notes in Pure and Applied Mathematics*, Springer-Verlag.

Burton, F. W. (1983) A Linear Space Translation of Functional Programs to Turner Combinators. *IPL* **14**, 5, pp. 201-204.

Burton, F. W. (1984) Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs. *TOPLAS* **6**, 2, pp. 159-174.

Church, A. (1941) *The Calculi of Lambda Conversion to Metamathematics*, Princeton University Press, Princeton.

Clarke, T. J. W., Gladstone, P. J. S., MacLean, C. D., Norman, A. C. (1980) SKIM - The S,K,I Reduction Machine. *Proceedings of the 1980 ACM Symposium on Lisp and Functional*

Programming, August 1980, pp. 128-135.

Curien, P. (1986) Categorical Combinators. *Information and Control* **69**, pp. 188-254.

Curry, H. B. (1930) Grundlagen der kombinatorishen Logik. *American Journal of Mathematics* **52**.

Eick, A., Fehr, E. (1983) Inconsistencies of Pure LISP. *Proceedings of 6th GI-Conference on Theoretical Computer Science, Lecture Notes in Computer Science* **145**, Springer Verlag, pp. 101-110.

Engeler, E. (1977) A New Type of Models of Computation. *Proceedings of the Sixth Symposium on the Mathematical Foundations of Computer Science*, Tatranska Lominca, Czechoslovakia, September 1977, *Lecture Notes in Computer Science* **53**, Springer Verlag, pp. 52-58.

Engeler, E. (1981) Algebras and Combinators. *Algebra Universalis* **13**, pp. 398-392.

Engeler, E. (1981) Equations in Combinatory Algebras. *Lecture Notes in Computer Science* **164**, Springer Verlag, pp. 193-205.

Foderaro, J. K., Sklower, K., Layer, K. (1983) *The Franz Lisp Manual.* Unix Distribution, University of California.

Friedman, D. P., Wise, D. S. (1976) CONS Should not Evaluate its Arguments. *Proceedings of the Third International Colloqium on Automata, Languages and Programming*, July 1976, Edinburgh University Press, Scotland, pp. 257-284.

Gibert, J., Shepherd, J. A. (1983) From Algebra to Compiler: A Combinator-Based Implementation of Functional Programming. *Proceedings of the Third Conference on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, December 1983, pp. 290-314.

Gibert, J. (1983) An Elementary Model for Functional Programming with Infinite Objects. Technical Report 37, Monash University, Clayton, Australia, May 1983.

Gibert, J. (1984a) Functional Programming with Combinators. *Proceedings of the Logic and Computation Conference*, Monash University, Australia, January 1984.

Gibert, J. (1984b) J-Machine Users' Manual. Technical Report 84/10, University of Melbourne, Parkville, Australia, September 1984.

Halstead, R. H. (1985) Multilisp: A Language for Concurrent Symbolic Computation. *TOPLAS* **7**, 4, pp. 501-538.

Hudak, P., Goldberg, B. (1985) Serial Combinators: "Optimal" Grains of Parallelism. *Proceedings of Functional Programming Languages and Computer Architectures*, September 1985, *Lecture Notes in Computer Science* **201**, Springer Verlag.

Hughes, R. J. M. (1982) Super-Combinators: A New Implementation Technique for Applicative Languages. *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, Pittsburg, PA, June 1982, pp. 1-10.

Jones, N. D., Muchnick, S. S. (1982) A Fixed-Program Machine for Combinator Expression Evaluation. *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, Pittsburg, PA, June 1982, pp. 11-20.

Kleene, S. C. (1936) Lambda Definability and Recursiveness. *Duke Mathematical Journal* **2**.

Kluge, W. E. (1983) Cooperating Reductions Machines. *IEEE Transactions on Computers* **32**, 11, November 1983.

Longo, G., Moggi, E. (1984) Godel Numberings, Principal Morphisms, Combinatory Algebras. *Lecture Notes in Computer Science* **176**, pp. 397-406.

Mago, G. (1982) Data Sharing in an FFP Machine. *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, Pittsburg, PA, June 1982, pp. 201-207.

Maurer, P. M., Oldehoeft, A. E. (1983) The Use of Combinators in Translating a Purely Functional Language to Low Level Data Flow Graphs. *Journal of Computer Languages* **8**, 1, pp. 27-45, June 1983.

Maurer, D., Oberhauser, H. (1985) Ein Simulator fur die parallele Reduktion von Kombinatorcode. SFB 124-C1, Universitat Saarbruecken,September 1985.

McCarthy, J. (1960) Recursive Functions of Symbol Expressions and their Computation by Machine. *Communications of the ACM* **3**, 4, PP. 184-194.

Meyer, A. R. (1982) What is a Model of the Lambda Calculus? *Information and Control* **52**, 1.

Obtulowicz, A., Wiweger, A. (1981) Functional Interpretation of Lambda Terms. *Colloquia Mathematica Societatis Janos Bolyai* **26**, Mathematical Logic in Computer Science, Hungary, 1978, North-Holland.

Peyton-Jones, S. L. (1982) An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions. *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, Pittsburg, PA, June 1982, PP. 150-158.

Sleep, M. R. (1980) Applicative Languages, Dataflow and Pure Combinatory Code. *Proceedings of COMPCON*, Spring 1980, pp.112-115.

Stoye, W. R., Clarke, T. J. W., Norman, A. C. (1984) Some Practical Methods for Rapid Combinator Reduction. *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, pp.159-166.

Thatcher, J. W., Wagner, E. G., Wright, J. B. (1981) More on Advice on Structuring Compilers and Proving Them Correct. *Theoretical Computer Science* **15**.

Treleaven, P. C. (1984) Decentralised Computer Architecture. *New Computer Architectures, International Lecture Series in Computer Science*, Academic Press, pp. 1-55.

Turner, D. A. (1979a) Another Algorithm for Bracket Abstraction. *Journal of Symbolic Logic* **44**, 2.

Turner, D. A. (1979b) A New Implementation Technique for Applicative Languages. *Software Practice and Experience* **9**.

Turner, D. A. (1981) Recursion Equations as a Programming Language. Functional *Programming and its Applications (An Advanced Course)*, July 1981, University of Newcastle-upon-Tyne, Cambridge University Press, pp. 1-28 ; *NORMA SASL*, Burroughs Corp., October 1985.

van der Poel, W. L., Schaap, C. E., van der Mey, G. (1980) New Arithmetical Operators in the Theory of Combinators (Parts I,II,III). *Indag. Math.* **42**, pp. 271-325.

Williams, J. H. (1981) Formal Representations for Recursively Defined Functional Programs. *Proceedings of the International Symposium on the Formalization of Programming*, Peniscola, Spain, April 1981, *Lecture Notes in Computer Science* **107**, Springer Verlag, pp. 460-470.

Williams, J. H. (1982) On the Development of the Algebra of Functional Programs. *ACM Transactions on Programming Languages and Systems* **4**.