A LISP COMPILER FOR FP LANGUAGE AND ITS PROOF VIA ALGEBRAIC SEMANTICS

C. CHOPPY, G. GUIHO, S. KAPLAN

Laboratoire de Recherche en Informatique
Université de Paris-Sud
Bâtiment 490
91405 Orsay - Cedex, FRANCE

INTRODUCTION

Since Backus pioneer paper [BACKUS 78], much work has been devoted to the study of the Functional Programming (FP) approach. As a main reason for this success, FP environment is characterized by a clean algebraic framework for reliable program design. The general purpose of this paper is to describe a LISP computation system for a FP language and to provide a FP algebraic semantics in order to prove the correctness of the system.

Part I provides a modelization of FP algebra of programs within the framework of abstract data types. The purpose of this part is to describe and define FP in a totally algebraic way (which strongly relies on the algebraic structures of FP).

Part II is the description of an actual interactive FP computation system. FP expressions are compiled into LISP code that is evaluated by the LISP interpreter. The system includes some efficient mechanisms, such as rapid error transmission (*strictness* in FP sense), simple optimization for function composition.

Part III sketches the proof of the compiler w.r.t. semantics given in part I. More precisely, we show how to perform a complete proof (which consists in proving the *axioms* given in the first part - assuming reasonable lemmas about the LISP environment), and conduct typical axioms demonstrations.

We assume the reader has basic knowledge about Backus Functional Programming systems. This paper is a short version of [CHOPPY et al. 83].

I - ALGEBRAIC SEMANTICS OF FP LANGUAGE

I.1. - INTRODUCTION

The algebraic behaviour of FP languages has been considered as an essential asset, in

particular compared with more classical programming languages. This is investigated in [BACKUS 78], where the set of FP programs is viewed as an algebra, and some properties (*theorems*) of this algebra are stated, as :

$$\forall\ f,g,h \quad [f\ ,\ g]\ o\ h\ =[f\ o\ h\ ,\ g\ o\ h]\ .$$

In order to consider this aspect more systematically, the formalism of abstract data types [ADJ 78, ZILLES 79,...] is particularly well suited. In this part, we provide an extensive modelization of FP environment in that framework. This gives a mathematical semantics for FP language, according to which it is proven (part III) that the FP compiler described in part II is correct.

Conversely, there has been several attempts to modelize programming languages with abstract data types [GOGUEN et al. 79, GAUDEL 80, WIRSING et al. 81, BROY et al. 80, ...]. It clearly happened to be easier for FP, due to its strong natural algebraic structure.

I.2 - DEFINITION OF THE ABSTRACT TYPE

The general structure of the type is summarized as follows :

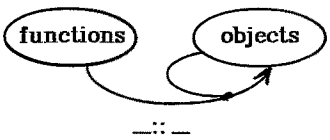


Figure 1 : FP type structure

[this denotes a function _:: _: **functions** x **objects** → **objects** , the "_" symbol standing for the position of the arguments.]

We now develop more precisely the structure of the two previous sorts.

OBJECTS AND LISTS OF OBJECTS

The structure of the sorts involved in the definition of FP base objects is the following :

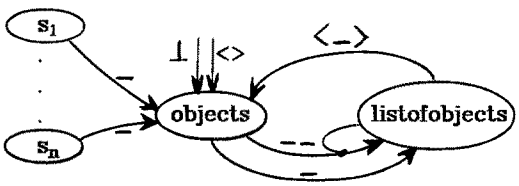


Figure 2 : General structure of objects sort

• The $\mathbf{s_i}$ are predefined sorts of disjoint atomic objects, with $\mathbf{s_1}$ and $\mathbf{s_2}$ being for instance the usual **boolean** and **integer** sorts.

We will require the existence of "equality predicates" for all $\mathbf{s_i}$:

$$\_?\ EQ_i\ ?\ \_:\ \mathbf{s_i}\ x\ \mathbf{s_i} \rightarrow \mathbf{s_1} = \mathbf{boolean}$$

that are sufficiently complete and hierarchically consistent with respect to **boolean** (cf. e.g. [GUTTAG et al. 78]).

The $\mathbf{s_i}$ are any types algebraically specified (i.e. with axioms and signatures). We will suppose that axioms about $\mathbf{s_i}$ (and $\_?\ EQ_i\ ?\ \_$) may be interpreted as canonical rewrite rule systems.

• To keep Backus syntax for lists, the following set of constructors for the sort **listofobjects** is needed :

$\_$: **objects** → **listofobjects**,
$\_\,\_$: **objects** x **listofobjects** → **listofobjects**

To ensure the construction of list to be a strict operation, the following equations "between constructors" are given, similar to classical "error propagation" axioms :

$< \_\,\_(\bot, 1) > = \bot$ $\quad$ $< \_\,\_(o , \_(\bot)) > = \bot$
$< \_(\bot) > = \bot$ $\quad$ $< \bot > = \bot$

On the whole, there are exactly four kinds of constructors for the sort **objects** :
the constant $\bot$,
the constant <> corresponding to the object "empty-list",
the coercions $\_$: $\mathbf{s_i}$ → **objects** yielding atomic FP-objects, and
the operator < $\_$ > transforming a **listofobjects** into an **object**.
From now on, we present set of axioms that are sufficiently complete w.r.t. this family of constructors.

We now define a *definedness* predicate : $?\!\downarrow\!? \_$: **objects** → **bool**, with the (sufficiently complete set of) axioms :

For i ∈ [1..n] , ∀x ∈ $\mathbf{s_i}$, $?\!\downarrow\!? (\_(x)) = ff$
$?\!\downarrow\!? (\bot) = tt$
∀o ∈ **objects** , ∀l ∈ **listofobjects**
$?\!\downarrow\!? (<>) = ff$ $\quad$ $?\!\downarrow\!? (< \_(o) >) = ?\!\downarrow\!? (o)$
$?\!\downarrow\!? (< \_\,\_(o,l) >) = ?\!\downarrow\!? (o)$ OR $?\!\downarrow\!? (< l >)$

This achieves the description of the **FP base objects**. It is tedious, though easy to check the following facts :
- semantics of the $\mathbf{s_i}$ is not altered by the new axioms (more precisely, the global specification is sufficiently complete w.r.t. the former specifications of the $\mathbf{s_i}$.
- semantics of the **objects** and **listofobjects** part is correct w.r.t. the usual FP model.
- we still have a runnable (i.e. confluent and noetherian) rewrite rule system (when the specifications of the $\mathbf{s_i}$ are so themselves).

THE "FUNCTIONS" SORT

FP functions are elements of the sort **functions**. We consider successively the three classical classes of functions : primitive functions (becoming constants in our formalization), functions obtained by application of combinators (that are operators of the algebra with non null arity), and recursively defined functions .

In FP, functions need to be **strict**. We thus adopt the following general axiom :

$$\forall f \in \mathbf{functions}, \quad f :: \bot = \bot$$

• *Primitive functions*
We shall give the algebraic specification of the primitive functions head and tail ; the other functions (selectors, identity, reverse, ...) would be modelized in the same way (cf. [CHOPPY et al. 83]).

Head and tail functions

We define the two "constants" head and tail in the following way :

$\forall x \in$ **objects** , $\forall l \in$ **listofobjects**

$?\downarrow?(< x\ l >) = ff \Rightarrow head :: < x\ l > = x$

$?\downarrow?(< x\ l >) = ff \Rightarrow tail :: < x\ l > = < l >$

$\forall l \in$ **listofobjects**

$?\downarrow?(< l >) = ff \Rightarrow head :: < l > = l$

$?\downarrow?(< l >) = ff \Rightarrow tail :: < l > = <>$

$head :: <> = \bot \quad tail :: <> = \bot$

$\forall x \in \mathbf{s_i}$ (for $i \in [1..n]$) $head :: \_x = \bot \quad tail :: \_x = \bot$

• *Functionals (combinators)*

Combinators, in our framework, are just operators of range **functions**, and of a non-null arity.

Composition

We have $\_ o \_$: **functions** x **functions** → **functions**, and :

$\forall f,g \in$ **functions**, $\forall x \in$ **objects** $(f \circ g) :: x = f :: (g :: x)$

Construction

To modelize the FP-construction of functions

(which associates to $f_1,...,f_n$ the function $[f_1,...,f_n]$, s.t:

$$[f_1,...,f_n] :: x = < f_1 :: x,...,f_n :: x >)$$

we apply the pattern we used to construct lists of objects introducing a new sort **listoffunctions**, and new operators "_ _", "_", [], and [_].

Apply to all

Let alpha : **functions** → **functions** with

$\forall f \in$ **functions** alpha f :: <> = <>

$\forall f \in$ **functions**, $\forall x \in$ **objects**, $\forall l \in$ **listofobjects**

alpha f :: < x l > = <(f :: x) (alpha f :: <l>)>

$\forall f \in$ **functions**, $\forall t \in \mathbf{s_i}$ (for $i \in [1..n]$) alpha f :: $\_t = \bot$

• *Recursively defined functions*

For recursively defined functions, a new sort **identifiers** is given, with a coercion from **identifiers** into **functions**. Because there is no notion of environment, and thus no global binding between function identifiers and their possible semantics, we use the following axiom :

$$\forall f \in \textbf{identifiers}, \forall x \in \textbf{objects}, \quad f :: x = \bot$$

(Applying the name of a function without body to an object gives bottom.)

We now introduce the fixpoint operator:

$$\text{fix}\_ \equiv \_ : \textbf{identifiers} \text{ x } \textbf{functions} \rightarrow \textbf{functions},$$

taking a name and a body, and producing a (well-defined) function.

Let us call *context* any term $\mathbf{T} \in T_{S,\Sigma}[X]$ of range **functions**, X being a variable of sort **functions** too. For instance,

$$\mathbf{T}_{FACT}[X] = eg0 \rightarrow 1 \text{ or } mult \circ [id , X \circ sub1]$$

is a context.

We now adopt the following *meta*-axiom :

for any context **T**, $\forall id \in$ **identifiers**, $\forall f \in$ **functions**

$[\text{fix id} \equiv \mathbf{T}[id]] :: x = \mathbf{T}[\text{fix id} \equiv \mathbf{T}[id]] :: x$

It just states that we can **unfold** the definition of a function.

Example :
Let t_{FACT} be fix idfunct $\equiv$ $\mathbf{T_{FACT}}$ [idfunct] ; one proves by induction on n=0 that

$$t_{FACT} :: n = n!$$

NOTE :
In the meta-axiom hereabove, the expression " fix id $\equiv$ **T**[id] " must be substituted to the formal argument of **T**. We shall suppose that the substitution is total (i.e. performed at every occurence in **T**). This is discussed in the next part.

I.3 - INITIAL ALGEBRA SEMANTICS

Having modelized FP with abstract data types (ADTs), we have provided a mathematical semantics for FP (which is used in the last part to prove the correctness of the compiler).
Actually, it is often considered that the semantics of an abstract data type is its initial model (when it exists). This can been done here, the initial model being :

$$\hat{I} = T_{S,\Sigma}/\equiv_{\{Ax\}},$$

where {Ax} is the set of (positive) axioms previously given.

This approach naturally provides a *least fixed point semantics* for recursively defined functions, in the following sense.
Suppose a classical recursive equation scheme f $\equiv$ **F**[f] is given, where **F** is a continuous functional.
Let f_o be the least fixed point of this equation. f_o may be partial. Let $x \in DOM(f_o)$.
It is possible, accordingly to [BACKUS 81b], to "lift" (which means approximately "to translate into FP") the definition of **F** , yielding a corresponding Φ , which is clearly a context in our sense. We then have :

THEOREM

$$(\text{fix id} \equiv \Phi[\text{id}]) :: \text{lift}[x] = \text{lift}[f_o(x)] \quad \textbf{in } \hat{I}.$$

This means that in $\hat{I}$, (fix id $\equiv$ **T**[id]) :: x *computes* the least fixed point of **T** (considered as the *lifted* of a classical recursive functional)
The proof of the theorem is clear, using the fact that successively applying the meta-axiom to a scheme corresponds to its computation through the *full-substitution computation* rule, which is safe (i.e. computes the least fixed point [MANNA 74]).

Notes :
- As shown in [ADJ 77], initial algebra semantics for an abstract data type is naturally equivalent to denotational semantics. Nevertheless, when applied to our previous modelization, this does not provide denotational semantics for FP *considered as a language*. On the other hand, it is possible to directly deduce denotational semantics from our framework in the following way :
  - the domain of the objects is just the **objects** sort of the initial algebra $\hat{I}$, ordered by $x < y$ iff $x = class_{\hat{I}}(\bot)$
  - FP functions are interpreted by their action in $\hat{I}$, through their application by $(\_ :: \_)^{\hat{I}}$ Functionals are treated analogously.
- The axioms that have been given may be interpreted as rewrite rules, including the meta-

axiom (applied by full-substition). This provides a confluent system, but which clearly does not always terminate ; for instance, the meta-axiom may be applied infinitely often to any operator fixpoint definition.
Nevertheless, it is a simple way of interpreting FP languages, which is correct w.r.t. to the previous initial algebra semantics (according to classical results on rewriting).

II. - LISP Computation of FP

This FP computation system is available both in MULTICS/MACLISP/EMACS and in UNIX/FRANZLISP/WINNIE [AMAR 83] environments (i.e. user interface is done through full-page editor). It makes use of a grammar generator and of a specialized parser [VOISIN 84].

The system is given the definition of the FP language that we deal with ; semantic attributes are attached to each FP symbol :

- the type of the symbol, used during the parsing process (which uses strong typing)
- a LISP expression representing semantics for the symbol, and used during the code generation step.

The computation itself simply works by parsing a given FP expression, realizing semantic attribute attachment (i.e. the FP expression is compiled into LISP code), and evaluating the resulting tree by mere LISP evaluation process.
This compiling process being simple, it will be shortly described with an example ; we shall give more details on features concerning strictness and function definition.

II.1 - EXAMPLE

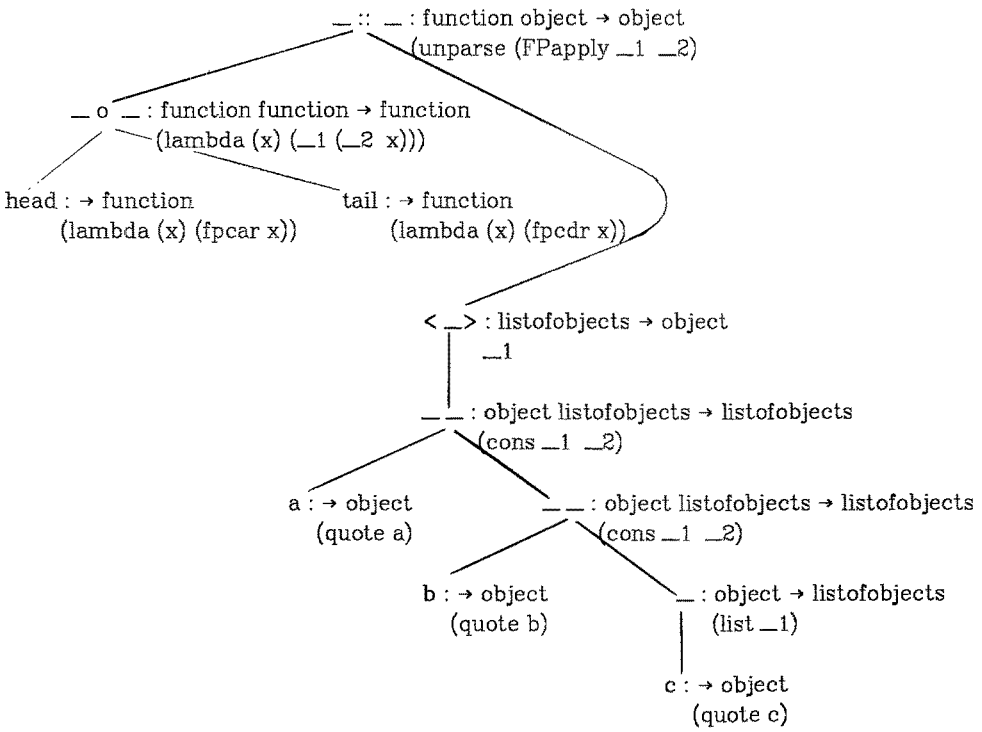
In order to provide some feeling for the system overall use and processing, and its evaluation mechanism, let us take an example : the figure (next page) is the resulting parse tree (after semantic attachment) of the FP expression :

head o tail :: < a b c >

Let us comment on this figure :

- the "_" signs in the operator name indicate the argument positions ("mixfixed" operators are allowed), therefore coercion is denoted by an operator name reduced to "_" ; in the operator semantics, the arguments are referred to by "_1", "_2",... (this is a macro provided by the system)
- semantics of the application "_:: _" : FPapply is application preserving strictness (see next paragraph), unparse translates a LISP expression into FP syntax
- semantics of _o _, which is simplified for sake of readability, actually involves beta-reduction
- primitive function semantics are expressed by LISP lambda expressions ; see for instance, semantics of head and tail : fpcar and fpcdr return (by "throw" mechanism) bottom when the argument is an atom
- objects are atoms (a, b, c,..., bottom, also booleans and integers via coercion) and sequences of atoms constructed by means of a sort "listofobjects" and operators : <> (empty sequence), < _>, _ _, _.

To compute this FP expression, one has to generate the code (i.e. the S-expression) corresponding to it, by traversing the tree, and evaluate, through the LISP interpreter, the resulting S-expression.



Nodes are labelled in the following way :

operator-name : operator-profile
operator-semantics

Figure 3 : Parse tree of head o tail :: < a b c >

II.2 - STRICTNESS

A rough way for implementing strictness would be to implement a ?⊥? predicate (as described in I.2) and have the application and the functionals use it. There are essentially two cases to consider :

(i) a mere application of a function to bottom :

head :: ⊥ = ⊥

(ii) bottom coming up at some point while evaluating an expression :

head o tail o head :: <a l> = ⊥

where a ∈ objects, l ∈ listofobjects. In this case, one wants bottom to "bubble up" through the several steps of evaluation. The LISP catch-throw mechanism provides an elegant way to by-pass evaluation steps. Along with the two cases considered above, catch-throws are used in the system :

(i) at the parsing level, the corresponding throw coming from the bottom semantics

(ii) at the application level, the corresponding throw coming from the primitive function semantics

II.3 - FIXPOINT SEMANTICS

Backus fixpoint definition is :

$$\text{fix}\ \{f \equiv \mathbf{E}(f\ g1\ \ldots\ gn)\} = E\ [\text{fix}\ \{f \equiv \mathbf{E}(f\ g1\ \ldots\ gn)\}\ /\ f\ ]$$

where / stands for substition symbol.
It is associated the following operator declaration :

(op fix _ ≡ _: identifiers functions → functions // (subst '(fix_≡ _ _1 _2) _1 _2))

where :

- (subst pattern1 tree pattern2) yields tree, in which each occurence of pattern2 is replaced by pattern1,
- fix_≡ _is the internal name of the fixpoint operator (with semantics hereabove).

Notice that the semantics exactly simulates the definition, and that it naturally corresponds with the semantics given in I.3 (the proof of the adequacy is done in part III).
We could also chose, instead of substituting everywhere, to substitute at some given occurences. It would be done by redefining the LISP function subst. This corresponds to giving computation rules for fixpoint computation, and is equivalent to implement the different semantics evoked in I.3.

II.4 - FUNCTION DEFINITION

Up to this point, we described within this system the only clean algebraic features of FP, as given for instance in part I. We now wish to be able to bind operator names to their bodies.
Backus [BACKUS 78] denotes by : Def l ≡ r definition of function l, where l is an unbound function symbol and r is a functional form (which may rely on l).
In our system, Def l ≡ r is an expression of type **defin**, where l is of type **identifiers** (function identifier), and r is of type **functions**. If the definition is recursive, then l will appear in the right hand side r in "call l", call being an operator with an argument l of type identifiers and a result l of type functions having the semantics of the function associated to the identifiers l.
The declaration of this operator is :

(op Def _ ≡ _: identifiers functions → defin // (eval '(op _1 : → functions // _2)))

Parsing and evaluating the expression Def l ≡ <body> causes the following actions :

- at a pre-parsing level :

declaration of l as an identifiers : (op l : identifiers) with default semantics (quote l). This pre-parsing action is somewhat inelegant, for it is generates side-effects during the parsing process; it allows the user to create new names without having to introduce the corresponding declaration at the grammar building level (in particular, one has to get out of FP environment to do this).

- when evaluating Def l ≡ <body> , the declaration :

(op l : → functions // (quote <body>)) is evaluated, thus realizing the binding.

In conclusion, the system described above for FP computation is efficient and works within a nice user interface. Its caracteristics are : the use of a strongly typed parser and fast and elegant processing for strictness and fixpoint computation.

III. - PROOF OF THE COMPILER

In this section, we prove that the FP computation system described in the second part is *valid* w.r.t. the semantics given in the first part. Actually, it is shown that *the FP compiler* (in its LISP environment) *is a model* [an algebra] *of the abstract data type defined in part I.*

In the following, the compiler (within its LISP environement) will be denoted by Ξ, and the abstract data type FP by $\overline{FP}$.
In order to prove that Ξ is a $\overline{FP}$-algebra, one has to
1) assign to each operator of $\overline{FP}$'s signature its *interpretation* in Ξ.
This is the easy part : each operator is interpreted as described in the second section, with its LISP action. For instance, the interpretation $(\_:: \_)_{\Xi}$ of the operator :

$\_:: \_$: **functions** x **objects** -> **objects**

is the function associating to its arguments _1 and _2 the LISP-result : (unparse (catch (apply _1 _2))) [cf. part II].

2) Now, Ξ is viewed as a sig($\overline{FP}$)-algebra. To prove that it is a $\overline{FP}$-algebra, it still has to be shown that Ξ satisfies the axioms of $\overline{FP}$, given in part I.
To do so, we shall avoid giving a formal description of the LISP system itself (the correctness of which would have to be proved) and then of the compiler Ξ, and finally prove the correctness of this formal model w.r.t. the algebraic semantics. It would be a large work, which is not absolutely necessary for our purpose.
The proofs are performed, assuming some reasonable *axioms* about the system, and some natural *deduction rules*. For instance, one will usually agree with the following axiom :

(car (cons a b)) = $_{\Xi}$ a (LISP-axiom)

provided that the evaluation of *a* yields no side-effect.

III.1. - THE PROOFS

In this part, we prove that Ξ satisfies the axioms of part I, i.e.

- the axioms concerning the object sorts. It is easy to check them in our implementation, and this wil not be developed here
- the general axiom : $f :: \perp = \perp$ (proven below)
- the axioms defining the behaviour of FP "constants" (in our sense) and "combinators". We will just prove one of them, the other proofs being similar
- the meta-axiom concerned with the definition of recursively defined functions (proved below).

Strictness of functions
The axiom to be proven is : $\forall f \in$ **functions** $f :: \perp = \perp$ **in** Ξ
or : $\forall f \in$ **functions** $(f :: \perp)_{\Xi} = (\perp)_{\Xi}$.
But : $(f :: \perp)_{\Xi} = f_{\Xi} ::_{\Xi} \perp_{\Xi}$ = (unparse (catch (apply 'f$_{\Xi}$ $\perp_{\Xi}$))).
Since : $\perp_{\Xi}$ = (throw 'bottom) ,
then : $(f :: \perp)_{\Xi}$ = (unparse 'bottom) = 'bottom = $\perp_{\Xi}$ QED

Proof of the correction of "head"
The three axioms defining head are :

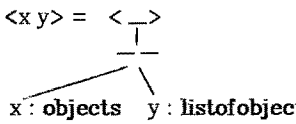
- head :: <> = $\perp$
- $\forall x \in \mathbf{s}_i$ (for i $\in$ [1..n]) head :: _x = $\perp$
- $\forall x \in$ **objects**, $\forall y \in$ **listofobjects**
  ?$\perp$?(< x y >) = ff $\Longrightarrow$ head :: < x y > = x

1) head :: <> = $\perp$
(head :: <>)$_{\Xi}$ = (unparse (catch (apply 'head$_{\Xi}$ <>$_{\Xi}$)))
= (unparse (catch (apply 'fpcar nil)))

But recall that :
fpcar = (lambda (x) (cond ((atom x) (throw 'bottom))) (t (car x))))
So : $(\text{head} :: \langle\rangle)_{\Xi} = \text{bottom} = \bot_{\Xi}$

2) head :: _x = ⊥
Let x be from any $\mathbf{s}_i$. We have : $(\text{head} :: \_x)_{\Xi}$ = (unparse (catch (fpcar $[\_x]_{\Xi}$)))
Since : $(\text{atom } [\_x]_{\Xi}) = \text{t}$, therefore : $(\text{head} :: \_x)_{\Xi}$ = bottom = $\bot_{\Xi}$

3) ?⊥?(<x y>) = ff ⟹ *head :: <x y> = x*
$(\text{head} :: \langle x\ y\rangle)_{\Xi}$ = (unparse (catch (fpcar $\langle x\ y\rangle_{\Xi}$)))
Now, recall that :



and that : $(\langle\_\rangle)_{\Xi}$ = (lambda (x) x) and $(\_\ \_)_{\Xi}$ = (lambda (x y) (cons x y))
So : $(\text{head} :: \langle x\ y\rangle)_{\Xi}$ = (unparse (catch (fpcar (cons x_{Ξ} y_{Ξ}))))
Suppose the following condition holds :

$$?\bot?(\langle x\ y\rangle)_{\Xi} = \text{ff}_{\Xi}$$

or $?\bot?_{\Xi}(\text{cons } x_{\Xi}\ y_{\Xi}) = \text{nil}$
Note that the exact meaning of $?\bot?_{\Xi}$ has not yet been defined. We shall suppose that it returns "t" whenever the Ξ-evaluation of its argument yields "bottom", and "nil" otherwise. Actually, it is a so-called *hidden function*, that does not physically exist in Ξ nor needs to be implemented, but that has to be supplied for the proofs.
Here, because $?\bot?_{\Xi}(\text{cons } x_{\Xi}\ y_{\Xi}) = \text{nil}$, we can apply the LISP-theorem :
(fpcar (cons a b)) = a when no side effect occurs.
Hence : $?\bot?_{\Xi}(\text{cons } x_{\Xi}\ y_{\Xi}) = \text{nil} \Longrightarrow (\text{head} :: \langle x\ y\rangle)_{\Xi} = x_{\Xi}$

QED

This achieves the proof that head in the system Ξ is correct w.r.t. $\overline{FP}$ algebraic semantics.

Proof of the meta-axiom
To prove : [fix id ≡ **T**[id]] :: x = **T**[fix id ≡ **T**[id]] :: x .
Recall that :
$(\text{fix}\_\equiv\_)_{\Xi}$ = (lambda (name body) (subst '($\text{fix}\_\equiv\_)_{\Xi}$ name body) name body))
where (subst x y z) is the result of substituting x for y everywhere in z.
Hence :
([fix id ≡ **T**[id]] :: x)$_{\Xi}$ =
(catch (unparse (apply (subst '($\text{fix}\_\equiv\_)_{\Xi}$ id_{Ξ} $\mathbf{T}_{\Xi}[\text{id}_{\Xi}]$) x_{Ξ}))) =
(catch (unparse (apply $\mathbf{T}_{\Xi}$ [($\text{fix}\_\equiv\_)_{\Xi}$ id_{Ξ} $\mathbf{T}_{\Xi}[\text{id}_{\Xi}]$] x_{Ξ}))) =
(catch (unparse (apply (**T**[fix id ≡ **T**[id]])$_{\Xi}$ x_{Ξ}))) =
(**T**[[fix id ≡ **T**[id]] :: x)$_{\Xi}$

QED

Thus, the compiler Ξ satisfies the meta-axiom. Note that the unfolding is performed in Ξ at every occurence of the identifier, as specified in the standard interpretation of the meta-axiom ("full-substitution" strategy). Another choice would be immediatly implemented by changing the action of the LISP-function "subst".
This achieves the proof of the computation system (recalling that the missing demonstrations are identical, in their principle, to the previous ones).

Remark : More precisely, the compiler belongs to the class $\overline{FP}$-alg* of the $\overline{FP}$-algebras satisfying :

$$tt \neq ff .$$

We may notice that the terminal model $T^*_{\overline{FP}}$ is characterized by the property of extensional equivalence : two objects f and g of sort **functions** are equal in $T^*_{\overline{FP}}$ iff
$\forall x \in$ **objects**, f :: x = g :: x . This is because the only *observator* acting on the sort **functions** is the application (_:: _), and that its range is the sort **objects** which is monomorphic in $\overline{FP}$-alg* (this stems from the completeness of the predicates _?EQ?$_i$_ w.r.t. the **booleans**).

• Now, suppose we decide to consider that two **functions** in Ξ are equal if they are extensionally equivalent **in** Ξ, then : Ξ **is the terminal model of** $T^*_{\overline{FP}}$.

Proof
We just have to prove that, if f,g $\in T_{S,\Sigma}$ of sort **functions**

$$f =_{ext} g \quad \text{iff} \quad f_\Xi =_{ext} g_\Xi$$

This is a simple consequence of the fact that the mechanism for computing recursive functions in Ξ *and* the formal evaluation process are similar, having both the least fixed point semantics property, and acting both by full-substition.
Note that getting extensional equivalence between functions through the consideration of terminal models is quite classical [KAMIN 80, WIRSING et al. 81].

• On the other hand, we could consider that two **functions** in Ξ are equal if they have the same definition (LISP genuine notion of equality). Then, we now claim that :

Ξ **is the initial model of** $T_{\overline{FP}}$,

Proof
We just need to prove that, if f,g $\in$ **functions**

$$f_\Xi = g_\Xi \quad \text{iff} \quad \{Ax\} \models f = g$$

$\Rightarrow$ If f and g are (LISP-)equal, they must have exactly the same definition. Then, f = g , and, a fortiori, {Ax} = f = g.
$\Leftarrow$ If {Ax} = f = g , then, in fact, f = g. This is because there is no equalities between objects of **functions** in {Ax}. QED
In conclusion, Ξ may be considered as the initial or the terminal model of $\overline{FP}$* , according to which philosophical meaning we agree to give to the notion of equality between functions in Ξ.
But anyhow, the choice corresponds exactly to the different notions of "meanings" we gave to the abstract data type describing FP.
Thus, we showed that Ξ is a correct FP compiler , with respect to the abstract type FP , **and** to the semantics we agreed to give to the type.

CONCLUSION

The aim of this paper was to describe a FP computation system, and to prove it.
This led to consider FP semantics using two different approaches : an axiomatic approach using the abstract data type theory, and a lambda-calculus and operational approach using a LISP environment. A connection was then established between them proving the validity of the latter w.r.t. the former.

In the first part, a complete formalization for a FP environment is given in the framework of algebraic abstract data types. This led straightforwardly to different semantics for FP. In particular, a thorough treatment for fixpoint definition of functions, parameterized by evaluation mechanisms was provided.

In the second part, a FP computation system is described within LISP environment. The organization of the system remained close to FP philosophy, even for the very conception of the compiler : for instance, FP functions are implemented as functions constants. An original way of implementing strictness is also provided.

Finally, this system is proven to be a model of the FP type. Notice that, the proof really dealt with the physical aspect of the compiler (not trying to further modelize it).

Ackowledgments
We want to thank Frèderic VOISIN for help in using and interfacing the parser with the compiler. We thank the members of the ASSPRO project for helpfull discussions.

We thank John WILLIAMS for fruitfull discussion and remarks.

REFERENCES

[ADJ 78]
Goguen, J.A., J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, correctness, and implementation of abstract data types, in Current Trends in Programming Methodology, vol. IV Data Structuring, R. Yeh ed., Prentice-Hall, 1978.

[AMAR 83]
Amar, P., Winnie : un editeur de textes multifenetres extensible, Actes des Journees BIGRE, Le Cap d'Agde, France, 1983.

[BACKUS 78]
Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, C.A.C.M., **21**, 8, pp. 613-639, August 1978.

[BACKUS 81a]
Backus, J., The algebra of functional programs : function level reasoning, linear equations and extended definitions, Proc. Int. Coll. on the Formalization of programming concepts, L.N.C.S. No 107, Peniscola, 1981.

[BACKUS 81b]
Backus, J., Function level programs as mathematical objects, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture, Wentworth-by-the-Sea, New Hampshire, October 1981.

[BIDOIT 81]
Bidoit, M., Une méthode de présentation des types abstraits : applications, Thèse de 3^{eme} cycle, Université Paris-Sud, Juin 1981.

[BROY et al. 80]
Broy, M., M. Wirsing, Algebraic definitions of a functional programming language and its semantical models, Institute fur Informatik der TU

[CHOPPY et al. 83]

Choppy, C., G. Guiho, S. Kaplan, Algebraic semantics for FP langguages, a lisp compiler and its proof, Rapport LRI N° 133, Orsay, 1983.

[GAUDEL 80]
Gaudel, M.C., Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation, Thèse d'Etat, Nancy, 1980.

[GOGUEN et al. 79]
Goguen, J.A., J.J. Tardo, An introduction to OBJ : a language for writing and testing formal algebraic program specifications, Specifications of Reliable Software Conf. Proc., Cambridge MA, April 1979.

[GUTTAG et al. 78]
Guttag, J., J. Horning, The algebraic specification of abstract data types, Acta Informatica, 10, pp. 27-52, 1978.

[GUTTAG et al. 81]
Guttag, J., J. Horning, and J. Williams, FP with data abstraction and strong typing, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture, Wentworth-by-the-Sea, New Hampshire, October 1981.

[HUET et al. 80]
Huet, G., J.M. Hullot, Proof by induction in equational theories with constructors, 21st IEEE Symp. on Foundations of Computer Science, 1980.

[KAMIN 80]
Kamin, S., Final data type specifications : a new data type specification method, 7th ACM Symp. on Principles of Programmong Languages, Las Vegas, 1980.

[KAPLAN 83]
Kaplan, S., Un langage de spécification de types abstraits algébriques, Thèse de 3^{eme} cycle, Orsay, Février 1983.

[MANNA 74]
Manna, Z., Mathematical theory of computation, Mc Graw Hill, 1974.

[VOISIN 84]
Voisin, F., CIGALE : Construction Interactive de Grammaire et Analyse Libérale d'expressions, Thèse de 3^{eme} cycle, Université d'Orsay, France.

[WILLIAMS 80]
Williams, J.H., On the development of the algebra of functional programs, Report RJ2983, I.B.M. Research Laboratory, San Jose, 1980.

[WILLIAMS 81]
Williams, J.H., Notes on the FP style of functional programing, Lecture Notes for the Course "FP and its applications", Newcastle-upon-Tyne, July 1981.

[WIRSING et al. 81]
Wirsing, M., M. Broy, An analysis of semantic models for algebraic specifications, Marktoberdof Summer School on Theor. Found. of Progr. Methodology, 1981.

[ZILLES 79]
Zilles, S.N., An introduction to data algebras, L.N.C.S. No 86, Springer Verlag, 1979.