Inconsistencies of pure LISP

Andreas Eick Elfriede Fehr
Lehrstuhl für Informatik II
RWTH Aachen

Abstract

In a first informal section we shall discuss some problems with definitions of list functions. In particular we motivate the restrictions of pure LISP with respect to list expressions using the full lambda definability.

In sections two and three we formally define the lambda-semantics of the pure LISP language as well as the denotational description of the original semantics of pure LISP as defined by the interpreter EVALQUOTE. In the last section, we show that even for this restricted language the interpreter does not work correctly with respect to the laws of the λ-calculus.

These results sharpen previous results by Perrot [6], Simon [7] and others, which point out that the usage of functional arguments in extended LISP destroys the α- and β-convertibility of LISP-expressions.

1. Introduction

Many algorithms can be formulated as string or list transformations. Either the underlying data structure corresponds immediately to lists or a suitable list representation was found because most machines require sequential input and output. The question in which (programming) language to express a list transformation is easy to answer for mathematicans: Take the lambda notation for defining functions and use some basic list operations as additional atoms in your expressions! This clearly gives a powerful tool for defining programs, and the mathematical theory which is well developed provides a sound basis for verification systems. Furthermore, it is well known that all lambda-definable functions are computable and there exists a number of strategies to compute the value of a lambda-expression, which are based on the laws of α- and β-conversion. These are the ideas some people developing applicative programming languages have in mind and Mc Carthy et.al. were the first to design a real language, namely LISP, on this base.
Unfortunately it turns out that α- and β-reduction cannot be efficiently performed on a machine. The main reasons for this are: 1. Searching for free variables in the arguments

which are bound in the body, finding new names and renaming variables accordingly, is highly inefficient and 2. The literal substitution of arguments for formal parameters is expensive.

A first idea to overcome these problems is to introduce a stack which holds pairs of formal and actual parameters and then evaluate procedure bodys refering to the stack whenever a variable occurs. It is easy to see that this approach works satisfactory in some cases but causes parasitic bindings in others. Consider the following example, where A and B denote constants:

$$t = (\lambda x.(\lambda y.(\lambda x.\ y\ B)\ x\ )\ A) \underset{\beta}{\longrightarrow} (\lambda y.(\lambda x.\ y\ B)\ A\ ) \underset{\beta}{\longrightarrow} (\lambda x.\ A\ B) \underset{\beta}{\longrightarrow} A$$

Hence, the intended meaning of the expression t is the value A . On the other hand a simple stack implementation would work as follows:

To compute t compute the value of

$(\lambda y.(\lambda x.yB)x)$ with entry [x | A] on the stack. Next compute $(\lambda x.y\ B)$ with respect to the stack [x | A] . [y | x] and finally take the value of y with respect to the stack [x | A] . [y | x] . [x | B] which yields B .

This example shows, that when the value of x is evaluated in the innermost block, then the actual environment is used rather than the static one given by the programmer. This defect is called 'dynamic binding' or "most recent" error (Mc Gowan [5]) . It causes problems because it is not consistent with the mathematical theory underlying the lambda-calculus.

Another idea to overcome these problems is to disallow local declarations of globally defined variables. It would be an unpleasant restriction though, because it would distroy the full modularity. Anyhow it can be shown by an example that this method does not work in general either. Consider the following expression together with two reductions:

$$(\lambda z.(zz)\ \lambda x.\lambda y.(xy)) \underset{\beta}{\longrightarrow} (\lambda x.\lambda y.(xy)\ \lambda x.\lambda y.(xy)) \underset{\beta}{\longrightarrow} \lambda y.(\lambda x.\lambda y.(xy)y)\ .$$

Although there were no redefined variables in the original expression, we find that after two reductions y is globally and locally defined.

The next idea to solve the problem is trying to completely evaluate arguments before they are put onto the stack. Of course one knows that this corresponds to the call-by-value strategy and is thus not always defined when a safe strategy terminates, but it seems to be possible to program in a style which tolerates call-by-value execution.

One positive aspect comes in with this idea, namely call-by-value increases efficiency because arguments are evaluated exactly once, even if they occur more often.

This method works correctly for our first example:

$(\lambda y.(\lambda x.\ y\ B)x)$ with entry $[x \mid A]$ would continue evaluating $(\lambda x.\ y\ B)$ with respect to the stack $[x \mid A] . [y \mid A]$ and then finish up by computing the value of y with respect to the stack $[x \mid A] . [y \mid A] . [x \mid B]$ which yields A .

Unfortunately, the suggested method does not work correctly for all expressions. Functional arguments can spoil the idea of reducing arguments completely before storing them. A stepwise reduction of our second example with corresponding stacks would work as follows:

	expression	stack
1.	$(\lambda z.(zz)\ \ \lambda x.\lambda y.(xy))$	$[\ ]$
2.	(zz)	$[z \mid \lambda x.\lambda y.(xy)]$
3.	$(\lambda x.\lambda y.(xy)z)$	$[z \mid \lambda x.\lambda y.(xy)]$
4.	$\lambda y.(xy)$	$[z \mid \lambda x.\lambda y.(xy)].[x \mid \lambda x.\lambda y.(xy)]$
5.	$\lambda y.(\lambda x.\lambda y.(xy)y)$	$[z \mid \lambda x.\lambda y.(xy)].[x \mid \lambda x.\lambda y.(xy)]$
6.	$\lambda y.\lambda y.(xy)$	$[z \mid \lambda x.\lambda y.(xy)].[x \mid \lambda x.\lambda y.(xy)].[x \mid y]$
7.	$\lambda y.\lambda y.(yy)$	

Observe that in step 2 the argument of z cannot be further evaluated. The conflict with β-reduction occurs in step 6, when x is associated to y which in the last step gets erroneously bound to the second declaration of y , instead of the intended first declaration.

The last idea we want to discuss here is to put a strict type system onto the language such that expressions are built up from functions which take a certain number of arguments of base type and produce a value of base type too. In order to obtain a sufficiently wide class of definable list functions a recursion operator is added to the language which is not necessary in the type-free case because then the Y-operator of the λ-calculus can be used instead. One can expect that the usage of base typed closed expressions allows a correct stack-implementation by the following considerations: If the expression is of base type (and closed) then the outermost arguments are also closed and of base type and hence one can inductively reduce the arguments to a constant value and then bind them on the stack to continue the execution. As constant values cannot be captured by other bindings, this seems to be a correct strategy.
The language we have arrived at is exactly the meta-language of pure LISP as introduced by Mc Carthy et.al. in [1] .
Now, in order to compare the operational semantics of LISP with the mathematically intended semantics of the LISP-expressions we shall develop a more formal framework in the next sections.

2. Syntax and static semantic specification of pure LISP

We shall use the method of denotational descriptions of programming language semantics in the Scott-Strachey style [8]. Our domains will be complete partial orders (cpo's), where a cpo is a partially ordered set containing a minimal element and least upper bounds of directed subsets.
The standard syntactic domains are flat cpo's for list structures and identifiers, where we obtain from an arbitrary set S the flat cpo $S_\perp$ by adjoining a minimal element $\perp$ and defining the order by $\perp \subseteq s$ for all $s \in S$.
We shall fix an infinite denumerable set X of variables and let f,x,y,z denote elements of X .

Definition 1 (the domains Id and L , and the set Σ)

(i) $Id := X_\perp$ is the syntactic domain of identifiers

(ii) $L := \underline{List}_\perp$ is the domain of lists, where the set $\underline{List}$ is given by the following grammer:

$\underline{List} ::= \underline{err} \mid \underline{Atom} \mid (\underline{List}\ *)$
$\underline{Atom} ::= \underline{Word} \mid \underline{Bool} \mid \underline{Integer}$
$\underline{Word} ::= \{A,\ldots,Z\}^+$
$\underline{Bool} ::= \underline{tt} \mid \underline{ff}$
$\underline{Integer} ::= \{0,\ldots,9\}^+$

(iii) Σ denotes the set of base functions, given explicitely by

$\Sigma := \{\underline{car}^{(1)}, \underline{cdr}^{(1)}, \underline{cons}^{(2)}, \underline{eq}^{(2)}, \underline{atom}^{(1)}, +^{(2)}, *^{(2)}, \dot{-}^{(2)}, \underline{Zero}^{(1)}, \ldots\}$,

where (n) denotes the functional arity n .

The meanings of base functions are exactly as they are defined for LISP, and can be looked up in Mc Carthy et. al [1]. For LISP-expressions one defines two syntactic categories, namely Form and Function with metavariables e and fn respectively. 'Forms' define objects of base type whereas 'Functions' define functional objects of type 1, i.e. they expect a finite number of arguments of base type and produce an answer of base type. These domains are specified by the following syntactic clauses:

Definition 2 (forms and functions)

$e ::= \underline{List} \mid X \mid fn[e_1;\ldots;e_n] \mid [e_{11} \to e_{12};\ldots;e_{n1} \to e_{n2}]$
$fn ::= \Sigma \mid X \mid \lambda\,[[x_1;\ldots;x_n];e] \mid \underline{label}\,[f;\ fn]$

Corresponding to the syntactic categories, there are two semantic functions for each denotational specification. Let us first formalize the static semantics, which is consistent with the conversion rules of the λ-calculus (see Fehr [3]) .

Let in general for a domain D with subdomain E $d_{|E}$ denote the projection of $d \in D$ onto E.

Defintion 3 ($[\![\ ]\!]^S$, the static semantics of pure LISP) :

(i) $U := [Id \to [L + [L^* \to L]]]$ is the set of environments with metavariable σ.

(ii) $E : \underline{Form} \to U \to L$ is the semantic function for 'Forms', where we denote $E(e)(\sigma)$ by $E[\![e]\!]^S\sigma$. E is given by the following equations:

a) $E[\![L]\!]^S\sigma = L$ for each $L \in \underline{List}$

b) $E[\![x]\!]^S\sigma = \sigma(x)_{|L}$ for each $x \in X$

c) $E[\![fn[e_1;\ldots;e_n]]\!]^S\sigma = F[\![fn]\!]^S\sigma(E[\![e_1]\!]^S\sigma,\ldots,E[\![e_n]\!]^S\sigma)$

d) $E[\![[e_{11} \to e_{12};\ldots;e_{n1} \to e_{n2}]]\!]^S\sigma = \underline{if}\ E[\![e_{11}]\!]^S\sigma\ \underline{then}\ E[\![e_{12}]\!]^S\sigma\ \underline{else\ if}\ \ldots\ \underline{else\ if}\ E[\![e_{n1}]\!]^S\sigma\ \underline{then}\ E[\![e_{n2}]\!]^S\sigma$.

(iii) $F : \underline{Function} \to U \to L^* \to L$ is the semantic function for 'Functions', where again we denote $F(fn)(\sigma)$ by $F[\![fn]\!]^S\sigma$. F is given by the following equations, where we use $\underline{\lambda}$ in the metalanguage to denote strict and rank free functions:

a) $F[\![f]\!]^S\sigma = \underline{\lambda}\, x_1 \ldots x_n .\ f(x_1,\ldots,x_n)$ for each $f \in \Sigma^{(n)}$

$= \sigma(f)_{|L^* \to L}$ for $f \in X$

b) $F[\![\lambda[[x_1;\ldots;x_n];e]]\!]^S\sigma = \underline{\lambda}\, l_1 \ldots l_n .\ E[\![e]\!]\sigma[x_1 \mid l_1]\ldots[x_n \mid l_n]$

c) $F[\![\underline{label}\ [f;\ fn]]\!]^S\sigma = \mu a.(\lambda F.F[\![fn]\!]^S\sigma[f / F])\ (a)$, where $\mu a.\ f(a)$ denotes the minimal fixed point of f.

We state now two results which derive from Scott's theory of lambda-calculus, and are formally proved for this restricted language over the list interpretation in Eick [2] .

Lemma 1 (α-consistency)
Let fn be an expression of the form $\lambda[[x_1;\ldots;x_n];e]$. Assume that the variable z does not occur in fn. Then $F[\![fn]\!]^S = F[\![\lambda[[x_1;\ldots;x_{i-1};z;x_{i+1};\ldots;x_n];\ \$_z^{x_i} e]]\!]^S$ holds for all $1 \leq i \leq n$, where $\$_t^x e$ denotes the result of substituting t for all free occurences of x in e.
This Lemma is the semantic pendant to the α-conversion in λ-calculus. Analoguous theorems can be proved for the β-conversion and value reduction:

Theorem 2 (β-consistency)
Let e be an expression of the form $\lambda[[x_1;\ldots;x_n];e'][e_1;\ldots;e_n]$. Assume that no variable occurs both free in $e_1;\ldots;e_n$ and bound in e'. Then

$E[\![e]\!]^s = [\![\$^{x_1 \ldots x_n}_{e_1 \ldots e_n} e']\!]^s$ holds.

Theorem 3 (δ-consistency) :
Let e be an expression of the form $f[L_1;\ldots;L_n]$, where $L_i \in \underline{List}$ for $1 \leqslant i \leqslant n$ and $f \in \Sigma$.
Then $E[\![f[L_1;\ldots;L_n]]\!]^s = L$, where $L = f(L_1,\ldots,L_n)$ holds.

The next theorem formalizes an important property of this semantic defintion:

Theorem 4 (substitutivity of equivalence):
Let $e_1, e_2 \in \underline{Form}$ be expressions such that $E[\![e_1]\!]^s = E[\![e_2]\!]^s$. If e is another form which contains e_1 as a subexpression, then $E[\![e]\!]^s = E[\![\$^{e_1}_{e_2} e]\!]^s$.

The essential idea of the proof of this theorem strongly depends on the fact that closed expressions are independant from the environment.

Theorem 5 (expansion rule):
$F[\![\underline{label}[f;fn]]\!]^s\sigma = F[\![fn]\!]^s\sigma[f \mid [\![fn]\!]^s\sigma]$

3. Dynamic semantic specification

In this section we shall present the original LISP-semantics in a denotational style as suggested by Gordon in [4]. He also proved in the same paper that this semantic description is equivalent to the operational definition given in Mc Carthy et. al. [1] by the interpreter function EVALQUOTE.

Definition 4 ($[\![\ ]\!]^d$ the dynamic (original) semantics of LISP):

(i) Env : Id → [Env → [L+[L* → L]]] is the set of recursive environments with meta-variable ρ . This is needed to model the fact that the meaning of free variables is not explicitely determined by an environment but rather as a function from (the calling) environment into values.

(ii) E' : Form → Env → L is the dynamic semantic function for forms, where we denote $E'(e)(\rho)$ by $[\![e]\!]^d\rho$. E' is given by the following equations:

a) $[\![L]\!]^d\rho = L$

b) $[\![x]\!]^d\rho = \rho(x)\rho_{|L}$

c) $[\![fn[e_1;\ldots;e_n]]\!]^d\rho = [\![fn]\!]^d\rho([\![e_1]\!]^d\rho,\ldots,[\![e_n]\!]^d\rho)$

d) $[\![[e_{11} \to e_{12};\ldots;e_{n1} \to e_{n2}]]\!]^d\rho =$ <u>if</u> $[\![e_{11}]\!]^d\rho$ <u>then</u> $[\![e_{12}]\!]^d\rho$ <u>else if</u> ... <u>else if</u> $[\![e_{n1}]\!]^d\rho$ <u>then</u> $[\![e_{n2}]\!]^d\rho$

(iii) F' : <u>function</u> → Env → L^* → L is the dynamic semantic function for 'functions' where again we denote $F'(fn)(\rho)$ by $[\![fn]\!]^d\rho$. F' is given by the following equations:

a) $$[\![f]\!]^d\rho = \begin{cases} \underline{\lambda}\ x_1\ldots x_n.\ f(x_1,\ldots,x_n) & \text{for each } f \in \Sigma^{(n)} \\ \rho(f)\rho \mid_{L^* \to L} & \text{for } f \in X \end{cases}$$

b) $[\![\lambda[[x_1;\ldots;x_n];e]]\!]^d\rho = \underline{\lambda}\ l_1\ldots l_n\ .\ [\![e]\!]^d\rho[^{x_1}/_{\lambda\rho.l_1}]\ldots[^{x_n}/_{\lambda\rho.l_n}]$

c) $[\![\underline{label}\,[f;fn]]\!]^d\rho = [\![fn]\!]^d\rho[f/[\![fn]\!]^d]$

This dynamic semantics looks quite similar to the static one: In iii(b) we see that abstraction yields bindings of constant functions to variables, together with definition ii(b) for the semantics of variables this yields exact equivalence for the λ-abstraction to the static case.
Furthermore, the definition iii(c) for <u>label</u> looks like one expansion of the recursive definition of f . Note however that in the updated environment f is not bound to $[\![fn]\!]^d\rho$, but to the function $[\![fn]\!]^d$ which expects a new environment first. How this fact can cause danger is demonstrated in the final section.

4. The difference between static and dynamic binding for pure LISP

Consider the following LISP-expression:

$e := \lambda[[x];\ \underline{label}\,[f;\ \lambda[[z];[z = 0 \to x;\ \underline{tt} \to \lambda[[x];f[x \dot{-} 1]][1]]]][1]][0]$

This example is a definition of the constant function 0 , when considered with static semantics, applied to the argument 0 . In the sequel we shall omit E and F , when it is clear from the context.

<u>Fact 1</u> $[\![e]\!]^s\sigma = 0$ for all environments σ

<u>Proof</u>

$[\![\lambda[[x];\ldots][0]]\!]^s\sigma = [\![\underline{label}\,[\ldots][1]]\!]^s\sigma[^x/_0]$ by Def. 3.ii(c) and iii(b)

$= [\![\lambda[[z];\ldots]]\!]^s\sigma[^x/_0][^f/_{[\ldots]^s\sigma[^x/_0]}](1)$ by theorem 5 and 3.ii(c)

$= [\![\lambda[[x];f[x \dot{-} 1]][1]]\!]^s\sigma[^x/_0][^f/_{\ldots}][^z/_1]$ by Def. 3.ii(c), iii(b), ii(d)

$= [\![f[x \dot{-} 1]]\!]^s\sigma[^x/_0][^{f/}_{\ldots}][^z/_1][^x/_1]$ by Def. 3.ii(c) and iii(b)

$= [\![\lambda[[z];\ldots]]\!]^s\sigma[^x/_0](0)$ by Def. 3.ii(c)

$= [\![[z = 0 \rightarrow x; tt \rightarrow \ldots]]\!]^{S}\sigma[^{x}/0][^{z}/0]$ by Def. 3.ii(c) and iii(b)

$= 0$ by Def. 3.ii(b)

Fact 2 $[\![e]\!]^{d}\rho = 1$ for all environments ρ

Proof $[\![\lambda[[x];\ldots][0]]\!]^{d}\rho = [\![\underline{\text{label}}\,[f;\ldots][1]]\!]^{d}\rho[^{x}/\lambda\rho.0]$ by 4.iii(b), ii(c), ii(a)

$= [\![\lambda[[z]\ldots]]\!]^{d}\rho[^{x}/\lambda\sigma.0][^{f}/[\![\lambda[[z]\ldots]]\!]^{d}](1)$ by Def. 4.iii(c), ii(c)

$= [\![\lambda[[x];f[x \dot{-} 1]][1]]\!]^{d}\sigma[^{x}/\ldots][^{f}/\ldots][^{z}/\lambda\rho.1]$ as above

$= [\![f[x \dot{-} 1]]\!]^{d}\rho[^{x}/\ldots][^{f}/\ldots][^{z}/\ldots][^{x}/\lambda\rho.1]$ as above

$= [\![\lambda[[z];\ldots]]\!]^{d}\rho[^{x}/\ldots][^{f}/\ldots][^{z}/\ldots][^{x}/\lambda\rho.1](0)$ by Def. 4.ii(c) et. al.

$= [\![x]\!]^{d}\rho[^{x}/\ldots][^{f}/\ldots][^{z}/\ldots][^{x}/\lambda\rho.1]$ by Def. 4.iii(b) and ii(d)

$= 1$ by 4.ii(b)

Fact 3 For any LISP-function fn and arguments arg 1,...,arg n , the following holds: $[\![\text{fn}[\text{arg 1};\ldots;\text{arg n}]]\!]^{d}\rho = \text{EVALQUOTE}[\text{fn};(\text{arg 1}\ldots\text{arg n})]$ for arbitrary ρ .

Proof see Gordon [4]

From these facts we can easily conclude the following theorems:

Theorem 6 The semantics of pure LISP is inconsistent with the static semantics.

Theorem 7 The semantics of pure LISP is inconsistent with α-conversion.

Proof Rename the second declaration of x in the above example into y . Compute the value analoguously as before. The result is 0 , which is different from 1.

Theorem 8 The semantics of pure LISP is inconsistent with β-conversion.

Proof Reduce in the above example the innermost redex. You obtain the form: $\lambda[[x]\ ;\ \underline{\text{label}}\ [f;\lambda[[z];[z = 0 \rightarrow x\ ;\ tt \rightarrow f[0]]]][1]][0]$.

This equals again to 0 , contradicting the value of e which was 1 .

Simon claims ([7] page 20):
"We have the strong opinion that the Church-Rosser property still holds for λ-calculus if the only operation of λ-terms is β-reduction on proper redexes", where a λ-term $(\lambda z.MN)$ is called a proper redex, if N is closed. He further claims that LISP fits to this restricted calculus, in the sense that the LISP interpreter works correctly with respect to this restriction. A corollary of theorem 8 contradicts his conjecture:

Corollary The semantics of pure LISP is inconsistent with β-reduction of proper redexes.

Proof Remark that the contracted redex in the proof of theorem 8 was proper.

These results are not only surprising, but also violate the mathematical theory underlying the lambda notation. Mc Carthy et. al. were wrong when they claimed in [1] p. 7:
"The variables in a lambda expression are dummy or bound variables because systematically changing them does not alter the meaning of the expression."

Proving properties of programs works usually by applying reduction rules. No valid reduction rules can be formulated for pure LISP on the expressions alone, but at least a simulation of the a-list has to be developed together with the reduction of expressions, as Gordon shows in [4] .
One question that arises now is, can pure LISP be further restricted in order to be safely evaluated by EVALQUOTE ?
The answer is not formally presented in this paper, but we show in Eick [2] that the answer can be positive if redeclaration of globally declared identifiers is forbidden. This restriction is equivalent to Mc Gowans sufficient condition for the correctness of his Interpreter I_{MR} , which performs the "most recent" evaluation strategy for block structured languages [5] . His condition is the following:
"A program must have no potentially recursive procedure R which contains the declaration of a procedure P(a label L) and a call of some procedure Q having P(L) as one of its arguments, where Q can call-*R".

References

[1] Mc Carthy, J. et. al.: "LISP 1.5 Programmer's Manual"
M.I.T. Press (1965)

[2] Eick, A.: "Semantische Analyse des LISP-Interpreters"
Diplomarbeit RWTH Aachen (1982)

[3] Fehr, E.: "The lambda-semantics of LISP"
Schriften zur Informatik und Angewandten Mathematik,
Bericht Nr. 72, RWTH Aachen (1981)

[4] Gordon, M.: "Operational reasoning and denotational semantics"
Stanford Artificial Intelligence Laboratory, Memo AIM-264, (1975)

[5] Mc Gowan, C.L.: "The "most recent" error: its causes and correction' SIGPLAN Notices, Vol. 7, Nr. 1 (1972)

[6] Perrot, J.-F.: "LISP et λ-Calcul" Ecole de Printemps d'Informatique Théorique, Le Châtre 1978

[7] Simon, F.: "Lambda Calculus and LISP", Bericht Nr. 8006, Institut für Informatik und Praktische Mathematik, Universität Kiel (1980)

[8] Strachey, C.: "Towards a Formal Semantics" in Formal Language Description Languages for Computer Programming (ed. Steele, T.B.), North-Holland (1966)