Recursion and Circularity
—Extended Puzzle With Solution—

By

Matthias Felleisen
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 201

Recursion and Circularity

—Extended Puzzle With Solution—

by

Matthias Felleisen

October, 1986

This material is based on work supported in part by an IBM Fellowship and by the National Science Foundation grant number DCR 85-01277.

16 October 1986

Recursion and Circularity

—Extended Puzzle with Solution—

Matthias Felleisen

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405, USA

The λ_{sv}-calculus. The λ_{sv}-calculus [2] is an extension of Plotkin's [5] λ_v-calculus. It is an algebraic system for reasoning with assignments in higher-order languages. Its syntax and reduction rules are formalized in Figure 1.

The term language is an extension of the original λ-calculus-language, Λ. There are four new concepts. Firstly, we have added a set of assignable variables. Unlike (constant) variables, assignable variables can be rebound to a new value, that is, all subterms of a program that are traceable to the same bound variable may be replaced by a different value. In order to keep track of these arguments we have introduced the concept of a *labeled* value. All values with the same label originated from the same β-step. Our third new concept is the class of σ-abstractions. When applied to an argument, they cause the rebinding of a labeled value. Lastly, we introduce the notion of a delabeling application which transforms labeled values into values.

The basic notions of reduction are term rewriting rules like the ordinary β-relation. Besides the basic β_v-rule we have a β_σ-rule which is used to label replacable, labeled values. σ-abstractions that are applied to a value and $\mathcal{D}$-applications "bubble" their way up in a term until they reach the root. Once they are at the top of the term, computation rules may be applied to them. These computation rules implement the proper effect, *i.e.*, they delabel the value or they reassign a new value to all subterms with the same label. For both computation rules the label replacement $M[n := V]$ is needed. It parses the term M and puts V^n in all places where it finds a subterm U^n.

Definition 2: The λ_{sv}-calculus

The improper symbols are λ, (,), ., σ, and $\mathcal{D}$. $Vars = Var_\lambda \cup Var_\sigma$ is a countable set of variables. We use the set of natural numbers, ω, as labels. The set of values contains variables, abstractions, and σ-abstractions. V ranges over values, n over labels, and X over Var_σ and labeled values. Λ_S contains

— *variables:* x if $x \in Var_\lambda$;

— *λ-abstractions:* $(\lambda x.M)$ if $M \in \Lambda_S$ and $x \in Vars$;

— *applications:* (MN) if $M, N \in \Lambda_S$;

— *σ-abstractions:* $(\sigma x.\, M)$ and $(\sigma V^n.M)$ if $M, N \in \Lambda_S$ and $x \in Var_\sigma$;

— *$\mathcal{D}$-applications:* $(\mathcal{D}\, x\, M)$ and $(\mathcal{D}\, V^n\, M)$ if $M \in \Lambda_S$ and $x \in Var_\sigma$.

The notions of reduction are

$$(\lambda x.M)V \xrightarrow{\beta_v} M[x := V] \qquad (\beta_v)$$

where $x \in Var_\lambda$ and V is a value,

$$(\lambda x.M)V \xrightarrow{\beta_\sigma} M[x := V^n] \qquad (\beta_\sigma)$$

where $x \in Var_\sigma$, V is a value, and n is fresh.

$$N((\sigma X.M)V) \xrightarrow{\sigma_R} (\sigma X.(NM))V \qquad (\sigma_R)$$

where N and V are values

$$((\sigma X.M)V)N \xrightarrow{\sigma_L} (\sigma X.(MN))V \qquad (\sigma_L)$$

where V is a value.

$$N(\mathcal{D}\, X\, M) \xrightarrow{\mathcal{D}_R} (\mathcal{D}\, X\, (\lambda v.N(Mv))) \qquad (\mathcal{D}_R)$$

where N is a value

$$(\mathcal{D}\, X\, M)N \xrightarrow{\mathcal{D}_L} (\mathcal{D}\, X\, (\lambda v.MvN)). \qquad (\mathcal{D}_L)$$

The top-level computation rules are

$$(\sigma U^n.M)V \rhd_\sigma M[n := V] \text{ where } V \text{ is a value} \qquad (\rhd_\sigma)$$

$$(\mathcal{D}\, V^n\, M) \rhd_{\mathcal{D}} MV[n := V]. \qquad (\rhd_{\mathcal{D}})$$

Equipped with these basic definitions we can turn to the question of how to use the calculus.

Reasoning with the λ_{sv}-calculus. For the reasoning with the λ_{sv}-calculus we define a series of equivalence relations. As usual the notions of reduction are the basis for a congruence relation over Λ_S-terms. We denote this relation with $=_m$. The equivalence closure of the computation rules together with $=_m$ form a non-compatible (non-congruent) equivalence $=_s$. This unconventional construction factors out all the points in a reduction sequence which must happen in a linear order. The two relations together are the core of the λ_{sv}-calculus.

Working with the basic notions of reduction and computation rules is sometimes cumbersome. We therefore introduce two meta-rules which shortcut recurring reduction sequences. These meta-rules are based on the notion of an *applicative context*. Contexts are terms with a hole. We use [ ] to denote holes. Applicative contexts are either a hole, a value applied to an applicative context, or an applicative context applied to an arbitrary term. The notation $C[\ ], C'[\ ], \ldots$ stands for an applicative context; this notation underlines that a context is a term which is a function of the contents of its hole. $C[M]$ accordingly represents a term which results from filling the hole in $C[\ ]$ with M. Given terms of the form $(\sigma U^n.M)V$ and $(\mathcal{D}U^nM)$ in an applicative context $C[\ ]$ we can show that the following two meta-rules hold:

$$C[(\mathcal{D}\,V^nN)] = C[NV[n := V^n]]$$
$$C[(\sigma V^n.N)V] = C[N][n := V^n].$$

These meta-rules are consistent with the standard reduction function and, hence, we can use them with the understanding that they do not lead us astray on the search for a value.

The two meta-rules play an important role in conjunction with the operational equivalence relation. We shall say that two terms are operationally equivalent, $M \approx N$, if there is no arbitrary context that can distinguish them hygienically. In other words, one can always substitute one for the other in a program context where neither term is in a label sharing relationship with the rest of the program. Clearly, $M =_m N$ implies $M \approx N$. It is furthermore possible to show that, if for all applicative contexts $C[\ ]$ $C[M] =_s C[N]$, then $M \approx N$. The next section illustrates an application of these equivalence relations to the problem of self-references.

Circular structures. Assignments introduce a peculiar effect into a higher-order programming language: they permit the explicit construction of self-referential

terms. On traditional machines this corresponds to circular structures. We investigate in this section how our term language deals with these circularities.

The essence of our treatment of self-referential assignments is a tricky interplay between the assignment abstraction and the delabeling application. For an example let us consider the expression

$$(\mathbf{let}\ (f\ulcorner 0\urcorner)\ ((\sigma f.\mathcal{D}f\mathbf{I})(\lambda x.(\mathcal{D}f\mathbf{I})x)), \qquad (*)$$

where the abbreviation $(\mathbf{let}\ (x\,v)\,b)$ stands for $(\lambda x.b)v$. Using the above meta-rules we get the following computation for any applicative context $C[\ ]$:

$$\begin{aligned} \ldots &= C[(\sigma\ulcorner 0\urcorner^n.\mathcal{D}\ulcorner 0\urcorner^n\mathbf{I})(\lambda x.(\mathcal{D}\ulcorner 0\urcorner^n\mathbf{I})x)] && (1)\\ &= C[\mathcal{D}(\lambda x.(\mathcal{D}\ulcorner 0\urcorner^n\mathbf{I})x)^n\mathbf{I}] && (2)\\ &= C[\lambda x.(\mathcal{D}(\lambda x.(\mathcal{D}\ulcorner 0\urcorner^n\mathbf{I})x)^n x)]. && (3)\end{aligned}$$

In words, in (*) f is first bound to some arbitrary value, the label n at the value indicating that this is a replacable value. Then, in line 1 this labeled value is assigned a value which contains an n-labeled value inside, *i.e.*, the assignment is self-referential. But, as can be seen from the resulting expression in line 2, the inner reference survives the assignment: it is not replaced. The way our calculus is set up, this inner replacement is not needed until it is truly delabeled. It is precisely for this reason that delabeling applications were introduced. They protect a labeled value from being delabeled too early. Once a value is delabeled we must make sure that inner references to the same label are updated such that they reflect the true state of the program. This is achieved by the delabeling computation rule. The effect can be observed in line 3 of our example. In some sense, the calculus maintains circular structures in a by-need manner.

Our above example has another interesting property. When the resulting function is invoked, it will go into an infinite loop. This is seen more easily if we use the following abbreviation for the expression in (*):

$$(\mathbf{rec}\ f\ (\lambda x.(\mathcal{D}f\mathbf{I})x)).$$

Recast in terms of operational equivalence, we have just demonstrated that

$$\begin{aligned}(\mathbf{rec}\ f\ (\lambda x.(\mathcal{D}f\mathbf{I})x)) &\approx \underline{\mathcal{D}(\lambda x.(\mathcal{D}\ulcorner 0\urcorner^n\mathbf{I})x)\mathbf{I}}\\ &\approx \lambda x.\underline{(\mathcal{D}(\lambda x.(\mathcal{D}\ulcorner 0\urcorner^n\mathbf{I})x)^n x)}\\ &\approx \lambda x.(\mathbf{rec}\ f\ (\lambda x.(\mathcal{D}f\mathbf{I})x))x,\end{aligned}$$

where the last step is justified by the definition of operational equivalence and the two preceding lines. Put informally, we have shown that when this last function is invoked it first regenerates itself and then uses the argument. We have constructed an infinite loop without using a recursion operator. Before we take a closer look at this phenomena, we briefly recall in the next section the basic principles of recursion in the traditional λ-calculus and λ_v-calculus.

Recursion. Recursive definitions are syntactic sugar for the λ-calculus of the form:

$$f = \ldots f \ldots f \ldots \ .$$

The self-reference on the right-hand side of the equation to the defined quantity is the characteristic of recursive definitions. In the λ-calculus the self-reference on the right-hand side can be restricted to a single occurrence of f:

$$f = (\lambda f. \ldots . f \ldots f \ldots) f \equiv (\lambda g. \ldots . g \ldots g \ldots) f.$$

From this reformulation it is clear why recursive definitions are syntactic sugar to the λ-calculus: a recursively defined object is the[1] fixpoint of some function and the $\mathbf{Y}$-combinator, $\mathbf{Y} \equiv \lambda f.(\lambda g.f(gg))(\lambda g.f(gg))$, of the λ-calculus constructs fixpoints for all functions F. In other words, given F, $F(\mathbf{Y}F) =_\beta \mathbf{Y}F$. Hence, the above equation is an abbreviation for

$$f \equiv \mathbf{Y}(\lambda g. \ldots . g \ldots g \ldots).$$

In the λ_v-calculus this construction does not work. Since $\mathbf{Y}F$ is not a value in $F(\mathbf{Y}F)$, it must be reduced further before F can absorb its argument. But this yields $F(F(\mathbf{Y}F))$ and leads obviously into an infinite loop. The dilemma is caused by the immediate evaluation of the fixpoint value $\mathbf{Y}F$. Assuming that we actually want to define a recursive function, the way out of this dilemma is an application of a standard trick in a by-value system, namely, delaying the action by abstraction until it is needed. In other words, since $\mathbf{Y}F$ represents a function, it is feasible to wrap it into $\lambda x. \ldots . x$. Therefore, we claim that a by-value recursion operator, $\mathbf{Y}_\mathrm{v}$, should satisfy

$$\mathbf{Y}_\mathrm{v}F =_{\beta_v} F(\lambda x.\mathbf{Y}_\mathrm{v}Fx).$$

[1] The fixpoint is indeed unique and is the *minimal* fixpoint. This is the result of the Scott-Stratchey denotational semantics for the λ-calculus.

We furthermore know that the inner application $\mathbf{Y}_v F$ is generated by the self-application gg in $\mathbf{Y}$'s defining equation. Hence, $\mathbf{Y}_v$ is defined by indirecting this self-application

$$\mathbf{Y}_v \equiv \lambda f.(\lambda g.f(\lambda x.ggx))(\lambda g.f(\lambda x.ggx)).$$

It is straightforward to verify that this definition of $\mathbf{Y}_v$ satisfies the above equation.

Recursion without self-application. Computer scientists realized early that recursion is a powerful strategy for realistic problem-solving. However, implementing recursion with the Y-operators as shown in the previous section is too costly. They use function application in abundance and function application is expensive on traditional as well as advanced machine architectures.

The usual improvement is the implementation of a recursive function F by a circular structure on the machine level [1, 6]. This is justified by the above equations which say that $\mathbf{Y}F$ regenerates itself when the recursive function is used. The self-reference is not eliminated but mapped onto a circular structure.

Landin [3] was the first one to realize that in a higher-order language with assignments this trick is available in the language itself. Since Landin did not have a calculus for his extended language, he could not prove any properties of this construction but had to rely on his intuition.

In the λ_{sv}-calculus recursive functions can—as indicated above—be defined with the syntactic form $(\mathbf{rec}\ f\ M_f)$, where f is the name by which M_f may refer to itself. The disadvantage of this approach is the introduction of a new syntactic form with binding character. It is desirable to build a function $\mathbf{Y}_{rec}$ which uses **rec**, but is more like $\mathbf{Y}$ or $\mathbf{Y}_v$.

A first approximation to the definition of $\mathbf{Y}_{rec}$ is

$$\mathbf{Y}_{rec} \equiv \lambda f.(\mathbf{rec}\ g\ f(\mathcal{D}g\mathbf{I})).$$

The intuition behind this definition is that the variable g stands for the pseudo-fixpoint of f and that f must be applied to this pseudo-fixpoint. Unfortunately, this does not work in a by-value system. The expression $(\mathbf{rec}\ g\ f(\mathcal{D}g\mathbf{I}))$ must be evaluated completely before g becomes bound to the proper recursive value. Hence, $f(\mathcal{D}g\mathbf{I})$ must be evaluated and this means that g is immediately delabeled. But the current value of g is arbitrary and, since it is passed by-value, it is not reassignable once f has returned a result.

The problem with this first approximation is again the problem of early evaluation. Assuming that we use $\mathbf{Y}_{rec}$ to define recursive functions we can use the

delaying technique to get the timing right. Since $f(\mathcal{D}g\mathbf{I})$ causes the problem we define $\mathbf{Y}_{\text{rec}}$ in our second approximation as

$$\mathbf{Y}_{\text{rec}} \equiv \lambda f.(\mathbf{rec}\ g\ \lambda x.f(\mathcal{D}g\mathbf{I})x).$$

Without syntactic sugar this becomes:

$$\mathbf{Y}_{\text{rec}} \equiv \lambda f.(\lambda g.(\sigma g.\mathcal{D}g\mathbf{I})(\lambda x.f(\mathcal{D}g\mathbf{I})x))\ulcorner 0\urcorner.$$

When applied to a function F in an arbitrary applicative context $C[\ ]$, the computation proceeds as follows:

$$\begin{aligned} C[\mathbf{Y}_{\text{rec}}F] &=_s C[(\lambda g.(\sigma g.(\mathcal{D}g\mathbf{I}))(\lambda x.F(\mathcal{D}g\mathbf{I})x))\ulcorner 0\urcorner)] \\ &=_s C[(\sigma\ulcorner 0\urcorner^m.(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I}))(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)] \\ &=_s C[\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)^m\mathbf{I}] \\ &=_s C[\lambda x.F(\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)^m\mathbf{I})x]. \end{aligned}$$

With the operational equivalence relation this can be expressed more succinctly:

$$\mathbf{Y}_{\text{rec}}F \approx \mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)^m\mathbf{I} \approx \lambda x.F(\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)^m\mathbf{I})x.$$

Since, by the definition of operational equivalence, indistinguishable terms can be replaced in a hygienic program contexts, we can deduce that, if $\lambda x.F[\ ]x$ and $(\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)^m\mathbf{I})$ do not share a label, then

$$\mathbf{Y}_{\text{rec}}F \approx \lambda x.F(\mathbf{Y}_{\text{rec}}F)x.$$

The precondition of this step is true if and only if the two instances of F do not share labels, *i.e.*, if F does not contain labels.

The derived operational equivalence for $\mathbf{Y}_{\text{rec}}$ nicely captures the fact that the defined quantity is a function, but it also stipulates that the $\mathbf{Y}_{\text{rec}}$-operator is different from the $\mathbf{Y}_{\text{v}}$-operator. In particular, the equations suggests that the two behave differently if the defined quantity is *not* a function. Since $\mathbf{Y}_{\text{rec}}$ insists on returning an abstraction, we expect to get a bad result back. It is not difficult to validate our suspicion: when applied to $\lambda x.0$ the two operators give different answers.

Given that $\mathbf{Y}_{\text{rec}}$ does not correspond to $\mathbf{Y}_{\text{v}}$, we can ask whether there is a variation of $\mathbf{Y}_{\text{v}}$ which satisifies the above operational equivalence of $\mathbf{Y}_{\text{rec}}$. Based on the extra indirection around $F(\mathbf{Y}_{\text{rec}}F)$ it is easy to see that we need a similar indirection in $\mathbf{Y}_{\text{v}}$ and thus we get:

$$\mathbf{Y}_{\text{v}}' \equiv \lambda fx.(\lambda g.f(\lambda x.ggx))(\lambda g.f(\lambda x.ggx))x.$$

A short calculation shows that this variation has the desired property.

A more interesting question is whether there is another approximation of $\mathbf{Y}_{\text{rec}}$ which is more like $\mathbf{Y}_{\text{v}}$. For the second approximation we had delayed the entire application $f(\mathcal{D}g\mathbf{I})$. This is unnecessary because it is only the delabeling of g which causes the problem. A delay of the $\mathcal{D}$-application alone should be sufficient since the recursive function is only needed when it is invoked. Thus we get a third approximation:[2]

$$\begin{aligned}\mathbf{Y}_{\text{rec}} &\equiv \lambda f.(\mathbf{rec}\ g\ (f(\lambda x.(\mathcal{D}g\mathbf{I})x))\\ &\equiv \lambda f.(\mathbf{let}\ (g\ulcorner 0\urcorner)\ ((\sigma g.\mathcal{D}g\mathbf{I})(f(\lambda x.(\mathcal{D}g\mathbf{I})x))))\\ &\equiv \lambda f.(\lambda g.((\sigma g.\mathcal{D}g\mathbf{I})(f(\lambda x.(\mathcal{D}g\mathbf{I})x))))\ulcorner 0\urcorner.\end{aligned}$$

A couple of calculation steps reveal that the application $\mathbf{Y}_{\text{rec}}F$ leads to an application of F to some function, but unlike in the case of $\mathbf{Y}_{\text{v}}F$ the result of this application is needed for the completion of the recursive definition:

$$C[\mathbf{Y}_{\text{rec}}F] =_s C[(\mathbf{let}\ (g\ulcorner 0\urcorner)\ ((\sigma g.\mathcal{D}g\mathbf{I})(F(\lambda x.(\mathcal{D}g\mathbf{I})x))))] \quad (1)$$
$$=_s C[((\sigma\ulcorner 0\urcorner^m.\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})(F(\lambda x.(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x)))]. \quad (2)$$

If we assume for the moment that in this context FV is operationally indistinguishable from some value $G[f \leftarrow V]$, then we can continue with

$$\ldots =_s C[((\sigma\ulcorner 0\urcorner^m.\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})G[f \leftarrow \lambda x.(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x]] \quad (3)$$
$$\ldots =_s C[\mathcal{D}G[f \leftarrow \lambda x.(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x]^m\mathbf{I})] \quad (4)$$
$$\ldots =_s C[G[f \leftarrow \underline{\lambda x.(\mathcal{D}G[f \leftarrow \lambda x.(\mathcal{D}\ulcorner 0\urcorner^m\mathbf{I})x]^m\mathbf{I})x}]]. \quad (5)$$

At this point we know from line (1) and (4) of this proof that the underlined subterm in (5) and $\mathbf{Y}_{\text{rec}}F$ are operationally indistinguishable. Thus, if we knew that the context of the underlined term was hygienic, we could fill in $\mathbf{Y}_{\text{rec}}F$. As above, this condition translates into the requirement that there are no labels in G. The label m is exempted because the replacement removes all of its instances. Assuming this additional condition we get:

$$\ldots \approx C[G[f \leftarrow \lambda x.\mathbf{Y}_{\text{rec}}Fx]]$$

[2] This variation is due to Bruce Duba, 1984. It comes closest in spirit to a transliteration of Landin's original $\mathbf{Y}_{\text{rec}}$-proposal for a by-reference language like Λ_S [3].

and by our first assumption and the fact that the context did not change

$$\ldots \approx C[F(\lambda x.\mathbf{Y}_{\mathrm{rec}} F x)].$$

In summary we have shown that $\mathbf{Y}_{\mathrm{rec}}$ satisfies the same operational equivalence relation as $\mathbf{Y}_{\mathrm{v}}$ provided the defining function F and its value guarantee two conditions:

- for all values V, $FV \approx G[f \leftarrow V]$ where $G[f \leftarrow V]$ is a value, and
- the value $G[f \leftarrow V]$ does not contain any other label than the ones introduced by V.

The second condition is quite restrictive. It rules out sensible programs such as loops which use imperative accumulators. We do not know how to avoid this. The first condition is a more viable restriction. It essentially requires that upon invocation F terminates without visible imperative actions or that it loops forever. Since the function F must be invoked by any recursion operator in order to yield the defined function, the termination condition causes no problem. The imperative-action part leads us back to the question posed at the outset of this question. Depending on how many times the defining function F is truly invoked, the imperative actions are performed a varying number of times. In other words, we really ask how many function applications are used to implement recursion with the various Y-operators.

Analysis. In the two preceding sections we have seen four recursion operators all of which allow the definition of recursive functions, but with slightly different operational properties:

$$\begin{aligned}
\mathbf{Y}_{\mathrm{v}} &\equiv \lambda f.(\lambda g.f(\lambda x.ggx))(\lambda g.f(\lambda x.ggx)) \\
\mathbf{Y}_{\mathrm{v}}' &\equiv \lambda f x.(\lambda g.f(\lambda x.ggx))(\lambda g.f(\lambda x.ggx))x \\
\mathbf{Y}_{\mathrm{rec}} &\equiv \lambda f.(\mathbf{rec}\ g\ f(\lambda x.gx)) \\
\mathbf{Y}_{\mathrm{rec}}' &\equiv \lambda f.(\mathbf{rec}\ g\ (\lambda x.fgx)).
\end{aligned}$$

For a comparison of these operators we adopt the standard reduction strategy as the basis of our measure. The important question is how many applications the defined recursive function needs according to this evaluation in order to achieve recursion. Hence, we first determine the functions which realize the recursive effects:

$$\begin{aligned}
\mathbf{Y}_{\mathrm{v}} F &= F\underline{(\lambda x.(\lambda g.F(\lambda x.ggx))(\lambda g.F(\lambda x.ggx)))} \\
\mathbf{Y}_{\mathrm{v}}' F &= \underline{(\lambda x.(\lambda g.F(\lambda x.ggx))(\lambda g.F(\lambda x.ggx)))} \\
\mathbf{Y}_{\mathrm{rec}} F &= G[f \leftarrow \underline{\lambda x.(\mathcal{D}G[f \leftarrow \lambda x.(\mathcal{D}\ulcorner 0 \urcorner^m \mathbf{I})x]^m \mathbf{I})x}] \\
\mathbf{Y}_{\mathrm{rec}}' F &= \underline{(\lambda x.F(\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0 \urcorner^m \mathbf{I})x)^m \mathbf{I})x)}.
\end{aligned}$$

The underlined parts are the relevant functions. The first two turn out to be the same, but they actually perform some of the work at different times. In principle, both of them reconstruct the recursive function from scratch once they are invoked. This happens by the inner self-application and the invocation of the defining function F on that result. When the recursive function is reconstructed, it is passed the recursion argument. This amounts to four applications per recursion step.

The first point to notice about the imperative versions constructed with $\mathbf{Y}_{\text{rec}}$ is that they do not contain a copy of the function F. This means that F is only invoked once; the rest of the work is done by the function f. When the function f is used to perform a recursion step, it simply reconstructs itself by a delabeling step and avoids any other applications, *i.e.*, f only uses one application per recursive invocation.

Recursion according to $\mathbf{Y}_{\text{rec}}'$ is more expensive than the one by $\mathbf{Y}_{\text{rec}}$ but still cheaper than a purely functional implementation. Instead of reconstructing the defined recursive function by a self-application, it is formed by a delabeling step and then applied to the functional F. Otherwise, recursion works like in the first two cases. It needs three applications per step.

Conclusion. In the preceding sections we have discussed four different recursion operators in an imperative, by-value, higher-order programming language. If they are used to define purely functional recursive functions, the four (probably) implement the same behavior, but are quite different in efficiency. If they are used to define recursive functions with globally visible side-effects or if they are applied to non-function defining functions, then the four operators possibly exhibit essential differences.

By lack of a model, the satisfaction of operational equivalence relations does not imply uniqueness and necessitates the conditional language in the preceding paragraph. Neverthelees, the discussion has illustrated that a formal system like the λ_{sv}-calculus can point to behavioral differences of functions. An interesting research question is to analyze the meaning of $\approx$ and related relations in a syntactic manner à la Morris [4] in order to get a better (minimal) characterization of our operators.

References.

1. BURGE, W. *Recursive Programming Techniques*, Addison-Wesley, 1975.

2. FELLEISEN, M., D.P. FRIEDMAN. A calculus for assignments in higher-order languages, *Proc. 14th ACM Symp. Principles of Programming Languages*, 1987, to appear.

3. LANDIN, P.J. A correspondence between ALGOL 60 and Church's lambda notation, *Comm. ACM,* **8**(2), 1965, 89–101; 158–165.

4. MORRIS, J.H. *Lambda-Calculus Models of Programming Languages*, Ph.D. Thesis, Project MAC, MAC-TR-57, MIT, 1968.

5. PLOTKIN, G. D. Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science* **1**, 1975, 125–159.

6. TURNER, D.A. A new implementation technique for applicative languages, *Software—Practice and Experience* **9**, 1979, 31–49.