Unrestricted Pure Call-by-Value Recursion

Johan Nordlander

Luleå University of Technology

johan.nordlander@ltu.se

Magnus Carlsson

Galois, Inc.

magnus@galois.com

Andy Gill

University of Kansas

andygill@ku.edu

Abstract

Call-by-value languages commonly restrict recursive definitions by only allowing functions and syntactically explicit values in the right-hand sides. As a consequence, some very appealing programming patterns that work well in lazy functional languages are hard to apply in a call-by-value setting, even though they might not be using laziness for any other purpose than to enable the desired form of recursion.

In this paper we present an operational semantics as well as a straightforward implementation technique for unrestricted recursion under pure call-by-value. On that basis we are able to demonstrate that highly recursive programming idioms such as combinator-based parsing are indeed compatible with call-by-value evaluation.

Categories and Subject Descriptors   D.3.3 [*Programming languages*]: Languages constructs and features—Recursion

General Terms   Languages, theory, experimentation

Keywords   Call-by-value, value recursion, semantics, implementation, combinator libraries.

1.   Introduction

One particularly attractive feature of lazy languages like Haskell is the ability to freely use recursion in the definition of data objects. Not only are recursive bindings in such languages available for objects of any type, but the right-hand sides of recursive bindings may also utilize whatever forms of computations that are appropriate, including function application to the variables being defined. Combinator libraries and domain-specific embedded languages written in Haskell make extensive use of both abilities in order to provide their desired abstraction boundaries. As an illustration, the following example of a combinator-based parser is definable by recursion, even though its type is abstract and its right-hand side contains multiple calls to abstract parser-constructing combinator functions.

$$exp :: Parser\ Exp$$
$$exp = choice\ [literal,\ seq\ [literal,\ op,\ exp],\ parens\ exp]$$

Call-by-value languages do not in general provide a comparable level of freedom. OCaml, for example, demands that the right-hand sides of recursive bindings are already of a form that are essentially values, and SML further restricts the type of recursive values to functions only. From a Haskell programmer's point of view, these limitations are unfortunate, as they mean that some of the most successful idioms developed in the Haskell community cannot easily be carried over if one would like to switch to a language with call-by-value semantics. In this paper we use Haskell syntax, but assume that we want to switch to a call-by-value semantics.

The debate over the relative merits of lazy and strict functional programming has lasted for decades, and will not be repeated here. Instead, we simply postulate that a strict but pure functional language would be a very valuable programming tool, with a potential for marrying the run-time efficiency and resource usage predictability of call-by-value evaluation with the declarative style of purely functional programming. However, a core question that follows from this assumption is of course to what extent programming idioms of the Haskell world are truly generic and not inherently tied to a lazy evaluation machinery. In this paper we demonstrate that as far as reliance on unrestricted recursion goes, Haskell-style programming can be comfortably applied in a suitably adapted call-by-value setting.

Our contributions are the following:

- We scrutinize a series of examples that motivate the need for unrestricted recursion and also build up an operational intuition of why computing recursive data structures under call-by-value is a much simpler problem than delaying and forcing arbitrary computations under a lazy evaluation strategy (Section 2).

- We provide an operational semantics for a call-by-value language with unrestricted recursion, whose clarity and simplicity is hardly hampered at all by the recursion support (Section 3). The semantics allows a degree of freedom in choosing whether to evaluate a variable to its bound value or not, so we prove that the evaluation relation is confluent, and as a corollary, referentially transparent. The system is further extended with algebraic datatypes, records and primitive datatypes in Section 4.

- We describe how the semantics of our language can be straightforwardly implemented, with a particular emphasis on making value inspection (i.e., pointer dereferencing) as cheap as in any call-by-value language not constrained by supporting value recursion (Section 5).

- We demonstrate the power of our approach by means of a call-by-value implementation of self-optimizing parser combinators, imitating some of the features of a combinator library originally written in Haskell by Swierstra and Duponcheel (Section 7). Our implementation shows – perhaps surprisingly – that programming in this style is not inherently tied to lazy evaluation semantics, simply to the availability of unrestricted recursion.

- We provide a general procedure for rewriting a certain class of programs that would otherwise be stuck according to our evaluation semantics, and we identify a new point in the design space between call-by-value and lazy evaluation characterized by the ability to perform these rewrites at run-time (Section 8).
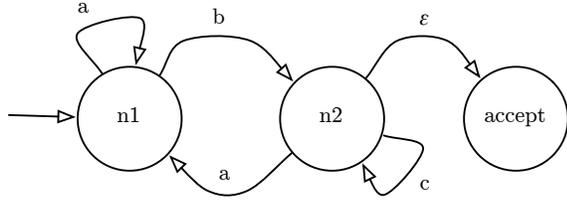
Figure 1. Example NFA

The reader should be aware upfront that this work is not concerned with the orthogonal issue of statically guaranteeing the absence of certain run-time errors, like the detection of ill-founded recursion. Our language does indeed allow ill-founded terms such as **let** $x = x$ **in** x to be written, and we do not attempt to provide any logical system for filtering them out. However, this is not a qualitatively different situation from being unable to statically detect non-terminating terms, or terms that terminate exceptionally because of division by zero. Most languages accept such terms anyway, with the argument that conservative filtering systems are too restrictive. We also note that even in Haskell, ill-founded recursion shows up as dynamic errors or non-termination, not as static type errors.

Related research on recursion under call-by-value, including some approaches to static error detection, are reviewed in Section 9.

2. Background and motivation

Consider the mutually recursive function definition

$$f = \lambda x. \ldots g \; x \ldots$$
$$g = \lambda y. \ldots f \; e \ldots$$

At run-time, f and g are represented as sequences of machine instructions that somehow must be able to refer to each other's locations in memory. On the program top-level, the most straightforward implementation generates code that just refers to the addresses of f and g symbolically, relying on an assembler to substitute real addresses for the symbols prior to execution. If the functions are represented as heap-allocated closures, however, the respective memory blocks must dynamically be set up to contain pointers to each other. This is still straightforward, though, as both blocks may be allocated and concrete addresses obtained before any memory initialization takes place.

Operationally, mutually recursive data structures may be understood in exactly the same way, even though their denotational semantics would have to be quite different from recursive functions in a call-by-value setting. For example, consider the definition

$$x = 1 : y$$
$$y = 2 : x,$$

where : is the infix list constructor symbol. Both x and y thus represent infinite alternating sequences of the integers 1 and 2, starting one element apart. Their representation can be finite, though. On the top level, x and y would be initialized memory areas that contain symbolic references to each other's addresses, with symbol resolution delegated to the assembly pass during code generation. If represented as heap-allocated objects, the cross-referencing addresses would be set up dynamically after both blocks have been allocated, just like the initialization of mutually recursive function closures.

It is important to note that there is nothing fundamentally at odds with call-by-value evaluation going on here. One might reason like this: In a call-by-value language, all variables stand for values, and under this assumption expressions $1 : y$ and $2 : x$ are also values, which is consistent with the assumption. Let us see how such recursive value bindings can be put to good use.

Figure 1 shows an example of a non-deterministic finite automaton (NFA) that accepts a certain set of strings. With recursive value bindings we can express this automaton right away as a cyclic structure of *NFA* nodes.

$$\begin{aligned}
\textbf{data } NFA =\ & N\,[(Char, NFA)]\,[NFA] \\
\mid\ & Accept
\end{aligned}$$

$$\begin{aligned}
start &= n1 \\
n1 &= N\,[('a', n1), ('b', n2)]\,[\,] \\
n2 &= N\,[('c', n2), ('a', n1)]\,[Accept]
\end{aligned}$$

A function accepting or rejecting strings on basis of such an automaton could look like this:

$$\begin{aligned}
check\ Accept\,[\,] &= True \\
check\ Accept\ xs &= False \\
check\ (N\ arcs\ eps)\,[\,] &= try\ eps\,[\,] \\
check\ (N\ arcs\ eps)\,(x{:}xs) &= try\,[\,n\mid(y,n)\leftarrow arcs, y == x\,]\ xs \\
&\quad\textbf{orelse}\ try\ eps\,(x{:}xs)
\end{aligned}$$

$$\begin{aligned}
try\,[\,]\ xs &= False \\
try\,(n:ns)\ xs &= check\ n\ xs\ \textbf{orelse}\ try\ ns\ xs
\end{aligned}$$

Note that the presence of cycles means that a function cannot naively traverse an *NFA* graph and expect to always terminate. Instead the termination of *check* and *try* above must depend on the finiteness of the string to be checked (and the absence of cycles with only ε-edges in the graph). The short-circuiting **orelse** construct borrowed from SML is also crucial here, as it only evaluates its right expression in case the left one is false.

Now, if we accept the idea that variables should count as values in a call-by-value language, we might ask why computations on them should not be allowed too. For example, if variables are values, the expression $(\lambda z.z)\,y$ is a call-by-value redex that reduces to a value (the variable y) in just one step. Hence we might argue that even though the definition

$$x = 1 : (\lambda z.z)\,y$$
$$y = 2 : x$$

is not a value binding, it can be reduced to a value binding in one step despite the mutual dependencies between x and y.

The core of our work is about how to make this idea precise in a formal manner, and how to implement it efficiently. For now, however, let us just assume that recursive bindings can contain arbitrary expressions in their right-hand sides, and see what kind of programs this would allow us to express.

One thing that becomes possible is to define a translation from regular expressions to NFAs, using a function that itself is recursive. The code could look as follows:

$$\begin{aligned}
\textbf{data } RegExp =\ & Lit\ Char \\
\mid\ & Seq\ RegExp\ RegExp \\
\mid\ & Star\ RegExp
\end{aligned}$$

$$\begin{aligned}
toNFA\ (Lit\ c)\ n &= N\,[(c, n)]\,[\,] \\
toNFA\ (Seq\ r1\ r2)\ n &= toNFA\ r1\ (toNFA\ r2\ n) \\
toNFA\ (Star\ r)\ n &= \textbf{let}\ n' = N\,[\,]\,[toNFA\ r\ n', n] \\
&\quad\textbf{in}\ n'
\end{aligned}$$

Function *toNFA* takes two arguments, one is the regular expression to translate and one is the NFA that the translated regexp should enter as its accept state. Notice the local recursive binding of n', which turns into a value binding only after an NFA for the regular expression r has been computed.

The reason this code works in spite of the recursive definition of n' is that *toNFA* never attempts to *inspect* its second argument; it only uses its parameter n as an abstract value for storing inside the data structures it builds. For this reason it does not really matter

that, in some cases, this abstract value is actually being computed by the very same function call!

A natural question to ask is of course what it would mean if a function indeed would attempt to inspect such a yet undefined value, for example by pattern-matching or by an equality test. Our chosen design is that such usage should be equivalent to a run-time error, on par with other dynamic errors like division by zero. We call programs with such behaviors *ill-defined*, and in the operational semantics we provide in Section 3 they will be identified as stuck configurations.

The most obviously ill-defined binding is

$$z = z,$$

as it makes the assumption that z is a value depend on itself. For the same reason the following example must be wrong:

$$z = head\ z : [\ ]$$

However, one could argue that a binding like

$$x = 1 : y$$
$$y = 2 + head\ x : x$$

should be correct, on basis of a fairly natural assumption that bindings are evaluated top-to-bottom. That is, instead of insisting that every recursively bound variable is treated like an abstract token during evaluation of all right-hand sides, one could consider imposing this restriction to *forward* variable references only, thus making mutually recursive bindings sensitive to the order in which they appear.

The language semantics we propose follows this idea, on the grounds that it offers the programmer more expressive power, but also because it appears to blend better with the rest of our technical design choices. We will return to this topic after we have described our operational semantics in Section 3. The order dependency between bindings might be considered unfortunate, but it should be noted that it only applies to bindings whose right-hand sides are not fully evaluated – value bindings can still be reordered at will as in any purely functional language.

To illustrate the power that follows from letting later bindings see the result of previous ones, we define a function *cap* :: *NFA* → *NFA*, that returns an NFA equal to its input except that all immediate edge labels are turned into upper case.

$$cap\ (N\ arcs\ eps) = N\ [\,(toUpper\ c, n)\,|\,(c, n) \leftarrow arcs\,]\ eps$$
$$cap\ Accept \qquad = Accept$$

We can now use this combinator to build an uppercase version of an NFA node *previously defined*.

$$a \quad = \ toNFA\ someRegExp\ start$$
$$b \quad = \ cap\ a$$
$$start = \ N\ [\,]\ [a, b, Accept]$$

Note the difference between how *toNFA* and *cap* treat their respective *NFA* arguments. Because *cap* really needs to look into the given automaton to find out which character labels it uses, the following reordering of a and b makes the definition ill-defined:

$$b \quad = \ cap\ a$$
$$a \quad = \ toNFA\ someRegExp\ start$$
$$start = \ N\ [\,]\ [a, b, Accept]$$

On the other hand, the binding for *start* has no such dependencies to other bindings, so it could have been placed at any position without changing the meaning of the recursive definition.

It should be emphasized here that the function *cap* is not able to capitalize *all* edge labels of the NFA it is given, only the labels of the explicit edges emanating from the root of the automaton. The reason *cap* is not recursively applied to the subsequent nodes it finds is that such a behavior would not terminate in case the given NFA contains cycles (which is exactly the kind of NFAs we are considering here). This complication is really an instance of a much more fundamental problem with unrestricted recursion: how to tell the difference between data structures that are not inductively defined and structures that actually are. While a solution to this problem would certainly be of value to programmers using our proposal, we consider it a topic outside the scope of our current work.

The fact that a function really needs to inspect the value of its argument – that it is *strict* in Haskell terminology – is a property that cannot be deduced from looking at its type, at least not if a standard Hindley/Milner type system is assumed. Programmers will thus need to use some caution when writing recursive bindings in our proposal, because well-typed programs might still terminate with a run-time error. However, this is certainly not a new problem; in fact a very similar problem exists when writing recursive bindings in Haskell.

Consider the program fragment

$$a = f\ b$$
$$b = g\ a$$

In Haskell this binding is ill-defined if both f and g are strict (assuming the values of a or b will actually be needed), and the program will either loop or stop with a message that a "black hole" has been detected. In our language it is ill-defined either when f is strict, or when the function composition $g \circ f$ is. Under both evaluation strategies, however, the programmer will have to consult information not captured in the types of f and g in order to conclude well-definedness.

There are many other examples of recursive value bindings being a clear and declarative mechanism to express behavior. For example, functional reactive programming (Elliott and Hudak 1997) builds cyclic structures, then interprets them, and could similarly be expressed in a strict language – if we had support for value recursion.

The recursive structures that may be computed by our proposal are cyclic, but they are not infinite – i.e., they cannot encode an infinite number of distinct nodes. A common term for data structures of this form is *regular*. In contrast, Haskell allows even irregular structures to be expressed, like the ones generated by the function *enumFrom*:

$$enumFrom = \lambda n.\, n : enumFrom\, (n + 1)$$

Although this recursive function definition is just as valid in call-by-value languages as it is in Haskell, any attempt to apply *enumFrom* in a call-by-value language would irrevocably lead to non-termination. This is of course just what to expect – *enumFrom* $(n + 1)$ is not a value even with our relaxed treatment of variables, and *enumFrom* obviously lacks the base case necessary for termination.

This observation emphasizes that our proposal for unrestricted call-by-value recursion is not a silver bullet that allows any kind of lazy programming pattern to be applied in a call-by-value setting. Our work concerns cyclic structures only, although this does indeed appear to be a feature that is particularly important to many purely functional idioms.

Another observation is that cyclic structures – at least in a call-by-value language – are not likely to be created only to be traversed once and then thrown away, simply because it is not possible to traverse them in finite time. A more likely role for these structures is to be used over and over; as in string recognizers and parsers, or perhaps even more importantly: in programs composed of mutually interacting components that encapsulate *monadic state*. Thus, if there are costs to be paid for the ability to form cyclic structures, it makes sense to take them up front at creation time rather than

$$
\begin{array}{lll}
e & ::= & \lambda x.e \mid x \mid e\,e' \mid \mathbf{let}\,b\,\mathbf{in}\,e \quad \text{Expressions} \\
b & ::= & x = e \mid b, b' \mid 0 \quad \text{Bindings}
\end{array}
$$

$$
\begin{array}{lll}
v & ::= & \lambda x.e \quad \text{Values} \\
\Gamma & ::= & x = v \mid \Gamma, \Gamma' \mid 0 \quad \text{Value bindings}
\end{array}
$$

$$
\begin{array}{lll}
w & ::= & v \mid x \quad \text{Weak values}
\end{array}
$$

Figure 2. Language grammar

$$
\Gamma \vdash (\lambda x.e)\,w \;\to\; [w/x]e \quad \textsc{Beta}
$$

$$
\Gamma, x = v, \Gamma' \vdash x \;\to\; v \quad \textsc{Var}
$$

$$
\frac{\Gamma + \mathcal{E} \vdash e \;\to\; e'}{\Gamma \vdash \mathcal{E}[e] \;\to\; \mathcal{E}[e']} \quad \textsc{Nest}
$$

$$
\Gamma \vdash \mathcal{E}[\mathbf{let}\,\Gamma'\,\mathbf{in}\,w] \;\to\; (\mathcal{E} + \Gamma')[w] \quad \textsc{Merge}
$$

Figure 3. The evaluation relation

paying them repeatedly each time such data is referenced. Our proposal is designed with this preference in mind, actually incurring no additional cost at all for accessing cyclic data compared to the acyclic case.

In the next section we will put these ideas on a firmer ground.

3. Semantics

Let x, y, z, f, g, h range over an enumerable set of variable identifiers. Our expression syntax is the lambda calculus extended with recursive let-bindings (algebraic datatypes, records and primitive values will be added in Section 4). The grammars for expressions and values are given in Figure 2.

The free variables of an expression e and the variables bound in a binding b are written $fv(e)$ and $bv(b)$ respectively. The comma operator used to combine bindings is associative and has the empty binding 0 as both right and left unit. We capture this fact as a structural equivalence relation $\equiv$ on bindings:

$$
\begin{array}{rcl}
b, (b', b'') & \equiv & (b, b'), b'' \\
b, 0 & \equiv & b \\
0, b & \equiv & b
\end{array}
$$

We treat terms as syntactically identical if they are alpha equivalent or equal up to $\equiv$ on bindings. Moreover, a binding b, b' is only considered syntactically correct if b and b' bind disjoint sets of variables. In the following we will implicitly assume that all terms are syntactically correct.

Evaluation in our system will potentially take place in the presence of free variables. Such variables can preferably be understood as *pointers* in a concrete implementation, and a value binding is thus equivalent to a *heap* mapping pointers to their associated values in a mutually recursive fashion. Small step evaluation is a ternary relation $\Gamma \vdash e \to e'$, which should be read "e may evaluate to e' in heap Γ". The evaluation rules are given in Figure 3.

The core technical detail in our semantics is the notion of *weak values*, which include variables in addition to the ordinary values of the lambda calculus. Weak values are ranged over by meta variable w, so rule BETA is thus standard call-by-value beta reduction extended to weak values. Note in particular that while weak values are accepted as arguments to functions, the range of value bindings (ranged over by Γ) is restricted to proper values v.

Rule VAR captures heap lookup, or "pointer dereferencing". Because variables count as weak values, applying the VAR rule is not strictly necessary for variables occurring in the argument position of a BETA redex. The freedom to take or not take VAR steps thus introduces an element of non-determinism in our semantics, a subject we will be returning to shortly.

Rules NEST and MERGE use an evaluation context $\mathcal{E}$ to capture the position in an expression where evaluation is next about to take place. A context is a term with a hole $[\,]$ in place of one of its subterms, as defined by the following grammar.

$$
\mathcal{E} \quad ::= \quad [\,]\,e \mid (\lambda x.e)\,[\,] \mid \mathbf{let}\,\Gamma, x = [\,], b\,\mathbf{in}\,e \mid \mathbf{let}\,\Gamma\,\mathbf{in}\,[\,]
$$

We write $bv(\mathcal{E})$ to denote the variables bound in $\mathcal{E}$ and also visible at its hole.

Rule NEST allows evaluation in a subexpression to count as evaluation of the enclosing expression as a whole. Here the heap environment is temporarily extended with the possible value bindings in scope at the hole of the context. This is written as an operation $\Gamma + \mathcal{E}$, whose result is an extended heap. $\Gamma + \mathcal{E}$ is defined by pattern-matching over the different forms of $\mathcal{E}$ as follows.

$$
\begin{array}{rcl}
\Gamma + (\mathbf{let}\,\Gamma', x = [\,], b\,\mathbf{in}\,e) & = & \Gamma, \Gamma' \\
\Gamma + (\mathbf{let}\,\Gamma'\,\mathbf{in}\,[\,]) & = & \Gamma, \Gamma' \\
\Gamma + ([\,]\,e) & = & \Gamma \\
\Gamma + ((\lambda x.e)\,[\,]) & = & \Gamma
\end{array}
$$

To avoid accidental name capture, $\Gamma + \mathcal{E}$ has a side-condition requiring that $bv(\Gamma) \cap bv(\mathcal{E}) = \emptyset$. It is however always possible to meet this requirement by means of alpha-renaming.

Finally, rule MERGE lets a value binding in a subexpression float outwards and merge with any bindings in the surrounding context. Again we encapsulate the details in a separate operation, writing $\mathcal{E} + \Gamma'$ for the context that results when context $\mathcal{E}$ is extended with value binding Γ'. This operation is also defined by pattern-matching over $\mathcal{E}$, assuming no overlap between Γ' and the bindings in $\mathcal{E}$. That is, applicability of this operator is restricted to the case where the sets $bv(\mathcal{E}) \cap bv(\Gamma')$ and $fv(\mathcal{E}) \cap bv(\Gamma')$ are both empty.

$$
\begin{array}{rcl}
(\mathbf{let}\,\Gamma, x = [\,], b\,\mathbf{in}\,e) + \Gamma' & = & \mathbf{let}\,\Gamma, \Gamma', x = [\,], b\,\mathbf{in}\,e \\
(\mathbf{let}\,\Gamma\,\mathbf{in}\,[\,]) + \Gamma' & = & \mathbf{let}\,\Gamma, \Gamma'\,\mathbf{in}\,[\,] \\
([\,]\,e) + \Gamma' & = & \mathbf{let}\,\Gamma'\,\mathbf{in}\,[\,]\,e \\
((\lambda x.e)\,[\,]) + \Gamma' & = & \mathbf{let}\,\Gamma'\,\mathbf{in}\,(\lambda x.e)\,[\,]
\end{array}
$$

Again we note that the side conditions assumed can always be met by alpha renaming the offending let-expression.

An evaluation context of the form $\mathbf{let}\,\Gamma', x = [\,], b\,\mathbf{in}\,e$ is noteworthy because the variables x and $bv(b)$ are in scope at the hole, even though the values of the corresponding right-hand sides have not yet been determined. When such a context extends the given heap Γ in rule NEST, the side-condition on $+$ enforces that Γ must be free from any existing bindings for x and $bv(b)$. Consequently, rule VAR is prevented from use on these variables during evaluation of the expression in the hole, as only the fully evaluated binding fragment of the context (Γ' that is) ends up in the extended heap. The implication of this scheme is that although a right-hand side may very well contain forward references, it must be able to reduce to a value without the need to treat the forward references as anything but abstract constants. In contrast, backward references (i.e., any references to $bv(\Gamma')$ from within the hole) may indeed be looked up using rule VAR, should it be required.

Here are some examples of reductions defined by our evaluation relation. For convenience we write $\Gamma \vdash e_1 \to e_2 \to e_3$ as a short-

hand for $\Gamma \vdash e_1 \to e_2 \wedge \Gamma \vdash e_2 \to e_3$.

$$\Gamma, x = v \vdash (\lambda y.y)\, x \; \to \; (\lambda y.y)\, v \; \to \; v \qquad (1)$$

$$\Gamma, x = v \vdash (\lambda y.y)\, x \; \to \; x \; \to \; v \qquad (2)$$

$$\Gamma \vdash (\lambda y.y)\, (\textbf{let}\, x = v\, \textbf{in}\, x) \; \to \; \textbf{let}\, x = v\, \textbf{in}\, (\lambda y.y)\, x \qquad (3)$$

$$\Gamma, f = \lambda x.f\, x \vdash f\, w \; \to \; (\lambda x.f\, x)\, w \; \to \; f\, w \; \to \; \ldots \qquad (4)$$

$$\Gamma, g = \lambda h.\lambda x.h\, x \vdash \textbf{let}\, f = g\, f\, \textbf{in}\, f\, w \; \to \qquad (5)$$
$$\textbf{let}\, f = (\lambda h.\lambda x.h\, x)\, f\, \textbf{in}\, f\, w \; \to$$
$$\textbf{let}\, f = \lambda x.f\, x\, \textbf{in}\, f\, w \; \to$$
$$\textbf{let}\, f = \lambda x.f\, x\, \textbf{in}\, (\lambda x.f\, x)\, w \; \to$$
$$\ldots$$

Examples (1) and (2) illustrate the different paths evaluation can take depending on whether a variable in argument position is reduced to its bound value or not. Example (1) shows the standard call-by-value case (nested VAR followed by BETA), whereas example (2) captures BETA reduction with a variable as argument (followed by a VAR step).

In example (3) the binding for x appears in a local let-expression that stands in the way for BETA reduction. This is where a MERGE step becomes necessary in order to move the local value binding outwards and expose the BETA redex.

Example (4) shows the beginning of the non-terminating reduction of a call to a trivially recursive function. The fact that it is non-terminating is not the interesting point here, only the unfolding of a typical recursive binding.

Example (5) essentially expresses the same non-terminating reduction chain, only this time the recursive function is formulated in terms of a non-recursive functional g, that is applied in the right-hand side of a recursive binding $f = g\, f$. Since this is not a value binding, the first steps of example (5) amount to reducing the right-hand side $g\, f$ to to the value $\lambda x.f\, x$; that is, a VAR step followed by a BETA. The derivation of the nested BETA step is illustrative, so we write it out in detail:

$$\frac{\Gamma' \vdash (\lambda h.\lambda x.h\, x)\, f \; \to \; \lambda x.f\, x}{\Gamma' \vdash \mathcal{E}[(\lambda h.\lambda x.h\, x)\, f] \; \to \; \mathcal{E}[\lambda x.f\, x]}$$

where $\mathcal{E} = (\textbf{let}\, f = [\,]\, \textbf{in}\, f\, w), \Gamma' = (\Gamma, g = \lambda h.\lambda x.h\, x)$

Note especially the underlying side-condition of heap extension, which ensures that Γ' cannot contain any prior bindings for f. This requirement effectively turns f into a constant while its right-hand side is being evaluated.

For contrast, we also show a couple of examples of recursive bindings that result in stuck evaluation:

$$\frac{\Gamma \vdash x \; \not\to}{\Gamma \vdash \textbf{let}\, x = x\, \textbf{in}\, e \; \not\to} \qquad (6)$$

$$\frac{\dfrac{\Gamma \vdash f \; \not\to}{\Gamma \vdash f\, y \; \not\to}}{\Gamma \vdash \textbf{let}\, f = f\, y\, \textbf{in}\, e \; \not\to} \qquad (7)$$

Example (7) is actually equal to the stuck configuration that would result if g in example (5) had been defined as $\lambda h.h\, y$ instead. As before, it is the side-condition on heap extension that prohibits the recursively bound variables from being captured by an outer scope.

Examples (1) and (2) above show an expression that may be evaluated in two different ways and still yield the same result. In general that is unfortunately not the case, however. Consider a slight variation of the referenced examples:

$$\Gamma, x = v \vdash (\lambda y.\lambda z.y)\, x \; \to \; (\lambda y.\lambda z.y)\, v \; \to \; \lambda z.v \qquad (8)$$

$$\Gamma, x = v \vdash (\lambda y.\lambda z.y)\, x \; \to \; \lambda z.x \qquad (9)$$

$$\Gamma \vdash e = e \quad \text{EQREFL}$$

$$\frac{\Gamma \vdash e_2 = e_1}{\Gamma \vdash e_1 = e_2} \quad \text{EQSYM}$$

$$\frac{\Gamma \vdash e_1 = e_2 \qquad \Gamma \vdash e_2 = e_3}{\Gamma \vdash e_1 = e_3} \quad \text{EQTRANS}$$

$$\Gamma, x = v, \Gamma' \vdash x = v \quad \text{EQVAR}$$

$$\frac{\Gamma\backslash_x \vdash e_1 = e_2}{\Gamma \vdash \lambda x.e_1 = \lambda x.e_2} \quad \text{EQLAM}$$

$$\frac{\Gamma \vdash e_1 = e_2 \qquad \Gamma \vdash e_1' = e_2'}{\Gamma \vdash e_1\, e_1' = e_2\, e_2'} \quad \text{EQAPP}$$

$$\frac{\Gamma\backslash_{bv(b_1)} \vdash b_1 = b_2 \qquad \Gamma\backslash_{bv(b_1)} \vdash e_1 = e_2}{\Gamma \vdash \textbf{let}\, b_1\, \textbf{in}\, e_1 = \textbf{let}\, b_2\, \textbf{in}\, e_2} \quad \text{EQLET1}$$

$$\frac{\Gamma\backslash_{bv(\Gamma_1)} \vdash \Gamma_1 = \Gamma_2 \qquad \Gamma, \Gamma_1 \vdash e_1 = e_2}{\Gamma \vdash \textbf{let}\, \Gamma_1\, \textbf{in}\, e_1 = \textbf{let}\, \Gamma_2\, \textbf{in}\, e_2} \quad \text{EQLET2}$$

$$\Gamma \vdash 0 = 0 \quad \text{EQNULL}$$

$$\frac{\Gamma \vdash e_1 = e_2 \qquad \Gamma \vdash b_1 = b_2}{\Gamma \vdash (x{=}e_1, b_1) = (x{=}e_2, b_2)} \quad \text{EQBIND1}$$

$$\frac{\Gamma \vdash v_1 = v_2 \qquad \Gamma, x = v_1 \vdash b_1 = b_2}{\Gamma \vdash (x{=}v_1, b_1) = (x{=}v_2, b_2)} \quad \text{EQBIND2}$$

Figure 4. Referential equivalence

The resulting lambda expressions are not identical, yet they are both values, so no further reductions are possible. This demonstrates that our calculus is not confluent.

However, it is clear that the resulting values are still related – in fact they only differ in subterms that are assumed equal in the given value binding environment. This observation leads us to a useful equivalence which we call *referential equivalence*; i.e., the relation obtained when equality between a variable and its bound value is lifted to an equivalence relation on expressions. Referential equivalence is formalized in Figure 4.

We are now able to prove a slightly weaker confluence property, which states that if an expression can evaluate in two different directions, the results can always be made referentially equivalent by evaluating them a bit further.

THEOREM 1 (Confluence up to referential equivalence). *If*

$$\Gamma \vdash e \to e_1 \qquad and$$
$$\Gamma \vdash e \to e_2$$

then there exist expressions e_1' and e_2' such that

$$\Gamma \vdash e_1 \to^* e_1' \qquad and$$
$$\Gamma \vdash e_2 \to^* e_2' \qquad and$$
$$\Gamma \vdash e_1' = e_2'.$$

Proof By structural induction on e.

Furthermore, if two terms are referentially equivalent, continued evaluation cannot destroy that relationship. That is, referential equivalence is preserved by evaluation. We call this property *referential transparency*, as it captures the ability to identify a variable

with its bound value without having to worry about the effect this might have on the result of repeated evaluation.

THEOREM 2 (Referential transparency). *If*

$$\Gamma \vdash e_1 \rightarrow e_1' \qquad and$$
$$\Gamma \vdash e_2 \rightarrow e_2' \qquad and$$
$$\Gamma \vdash e_1 = e_2$$

then there exist expressions e_1'' and e_2'' such that

$$\Gamma \vdash e_1' \rightarrow^* e_1'' \qquad and$$
$$\Gamma \vdash e_2' \rightarrow^* e_2'' \qquad and$$
$$\Gamma \vdash e_1'' = e_2''.$$

Proof By induction on the derivation of $\Gamma \vdash e_1 = e_2$, applying Theorem 1 in the EQREFL case.

It should be noted that let-expressions are never fully eliminated in this calculus; all that may happen is that local value bindings get merged with the bindings of an enclosing let-expression in a MERGE step. The result of successful evaluation is thus in general a term of the form **let** Γ **in** w, irrespective of whether w refers to the variables bound in Γ or not. To let value bindings disappear from a term when they are no longer needed, one could consider adding *garbage collection* to the system; for example in the form of the following evaluation rules:

$$\Gamma \vdash \textbf{let } 0 \textbf{ in } e \rightarrow e \quad \text{GC1}$$

$$\frac{bv(\Gamma_2) \cap fv(\Gamma_1, \Gamma_3, e) = \emptyset}{\Gamma \vdash \textbf{let } \Gamma_1, \Gamma_2, \Gamma_3 \textbf{ in } e \rightarrow \textbf{let } \Gamma_1, \Gamma_3 \textbf{ in } e} \quad \text{GC2}$$

However, doing so would necessarily introduce another source of non-determinism in the calculus, and a need for a notion of equivalence in the presence of garbage collection. While both interesting and important, this is a topic that falls outside the scope of the current paper.

4. Extensions

Algebraic datatypes, records, as well as integers and other primitive types can easily be added to our system by simply extending the grammars for expressions, values and evaluation contexts, and adding the corresponding reduction axioms.

Here is a datatype extension supporting constructor applications and case analysis, with k ranging over some enumerable set of constructor names.

$$
\begin{array}{lcl}
e & ::= & \dots \mid k\,\overline{e} \mid \textbf{case } e \textbf{ of } \{k_i\,\overline{x}_i \rightarrow e_i\} \\
\mathcal{E} & ::= & \dots \mid k\,\overline{[]} \mid \textbf{case } [] \textbf{ of } \{k_i\,\overline{x}_i \rightarrow e_i\} \\
v & ::= & \dots \mid k\,\overline{w}
\end{array}
$$

$$\Gamma \vdash \textbf{case } k_j\,\overline{w} \textbf{ of } \{k_i\,\overline{x}_i \rightarrow e_i\} \rightarrow [\overline{w}/\overline{x}_j]e_j \quad \text{CASE}$$

The notation $\overline{[]}$ should here be read as a vector of expressions of which *one* is a hole. The actual order in which constructor arguments are evaluated can of course be fixed, but it is not important. Notice that constructor values $k\,\overline{w}$ accept weak values as arguments; this is the technical detail that allows cyclic datatype values to be constructed.

Record construction and selection is added in a similar way, assuming an enumerable set of label identifiers ranged over by l.

$$
\begin{array}{lcl}
e & ::= & \dots \mid \{l_i = e_i\} \mid e.l \\
\mathcal{E} & ::= & \dots \mid \{l_i = []_i\} \mid [].l \\
v & ::= & \dots \mid \{l_i = w_i\}
\end{array}
$$

$$\Gamma \vdash \{l_i = w_i\}.l_j \rightarrow w_j \quad \text{SEL}$$

Once more we abuse notation and let $\{l_i = []_i\}$ stand for a record term with a hole in one of its labeled fields, and again it is the

presence of weak values in the syntax for record values that allow cyclic record structures to be defined.

Primitive types and operations can be added as follows:

$$
\begin{array}{lcl}
e & ::= & \dots \mid n \mid e \oplus e' \\
\mathcal{E} & ::= & \dots \mid [] \oplus e \mid n \oplus [] \\
v & ::= & \dots \mid n
\end{array}
$$

$$\frac{n_1 + n_2 = v}{\Gamma \vdash n_1 \oplus n_2 \rightarrow v} \quad \text{PRIM}$$

Here n ranges over all kinds of primitive values, whereas $\oplus$ acts as a syntactic placeholder for all possible binary operations on these, with $+$ being the corresponding semantic operation. The shown inclusion of primitive values among the full values v makes them *boxed*; i.e., storable in the heap. This is not always desirable in a call-by-value language, so an alternative approach would be to only include them among the weak values instead. That would of course also rule out the possibility of defining primitive values by recursion, but one can argue that such definitions would not be particularly meaningful anyway, at least not for the common numeric types.

A few more evaluation examples illustrate the construction and use of cyclic data structures. To save space we use cyclic records here, but the same patterns apply to algebraic datatypes as well. The labels l_h and l_t can preferably be read as *head* and *tail*, respectively.

$$
\begin{aligned}
\Gamma, x = \{l_h=1, l_t=x\} \vdash x.l_t.l_h \rightarrow \\
\{l_h=1, l_t=x\}.l_t.l_h \rightarrow x.l_h \rightarrow \\
\{l_h=1, l_t=x\}.l_h \rightarrow 1 \qquad (10)
\end{aligned}
$$

$$
\begin{aligned}
\Gamma, f = \lambda y.\lambda z.\{l_h=y, l_t=z\} \vdash \textbf{let } x = f\,1\,x \textbf{ in } e \rightarrow^* \\
\textbf{let } x = \{l_h=1, l_t=x\} \textbf{ in } e \qquad (11)
\end{aligned}
$$

$$
\begin{aligned}
\Gamma \vdash \textbf{let } x = \{l_h=1, l_t=y\}, y = \{l_h=x.l_h, l_t=x\} \textbf{ in } e \rightarrow^* \\
\textbf{let } x = \{l_h=1, l_t=y\}, y = \{l_h=1, l_t=x\} \textbf{ in } e \qquad (12)
\end{aligned}
$$

$$\Gamma \vdash \textbf{let } x = \{l_h=y.l_h, l_t=y\}, y = \{l_h=2, l_t=x\} \textbf{ in } e \nrightarrow \quad (13)$$

Example (10) shows the unfolding of a cyclic data structure, whereas example (11) illustrates how the right-hand side of a cyclic binding can be abstracted out as a function. These reductions are data structure analogs of examples (4) and (5) of the previous section.

The reduction steps in example (12) demonstrate that field selection from a recursively bound variable is acceptable as long as the variable is a backward reference. An attempt to select from a forward reference results in a stuck term, as shown in example (13). The asymmetry between these two cases is further illustrated in Figure 5, where thick arrows depict the kind of references that are well-defined, and the dotted one marks the failed attempt to extract a subterm from a yet undefined data structure.

A framework for circumventing this problem by rewriting the failed programs will be discussed in Section 8.

5. Implementation

One of our foremost goals has been to be able to implement unrestricted call-by-value recursion without imposing any additional run-time penalties on either function calls or data structure accesses. That goal has immediately outruled many implementation techniques for value recursion that have been suggested elsewhere, such as representing recursively defined values via pointer indirections (Boudol and Zimmer 2002), or making selective use of
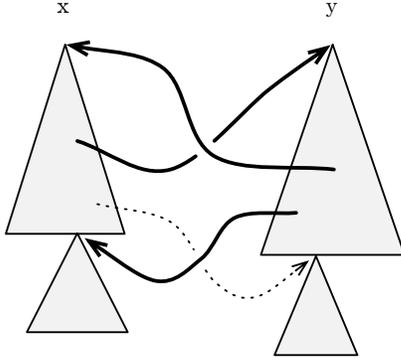
Figure 5. Mutually recursive data structures

lazy evaluation in the right-hand sides of recursive bindings (Syme 2006).

Our implementation is instead based upon the use of *illegal pointers* as the run-time representation of recursive values that are yet undefined, and a *substitution mechanism* that replaces those illegal pointers with the corresponding valid heap addresses as they become available. But before we go into detail about this technique, let us set the scene by a brief description on how the various other pieces of our operational semantics map onto a concrete implementation.

A value binding Γ corresponds to a heap, either implemented as statically initialized memory if that is possible, or dynamically created and garbage collected in the general case. The bound variables of a value binding correspond to heap addresses. Lambda-bound variables do not map to pointers, though; they are instead realized as locations within the stack frames and register files that lazily implement the argument/parameter substitutions prescribed by the BETA rule.

A VAR step is the equivalent of pointer dereferencing. The freedom to take or not take VAR steps before BETA reduction corresponds to the freedom the implementor of a referentially transparent language has when it comes to passing function parameters by value or by reference. Likewise, the presence of weak values in the syntax for record and constructor values imply a corresponding freedom to inline subterms of data structures. It should be noted, though, that the possibility of cycles in heap-allocated data makes it necessary to apply a more conservative inlining strategy than eagerly inlining everywhere. Moreover, inlining of arguments and subterms is only possible in cases where the compiler can prove that a variable actually stands for real data and not some heap value that is currently being constructed. The simplest strategy for inlining is thus to avoid it wherever a weak value is sufficient, and only dereference pointers where the semantics actually demands a value.

Heap allocation is the run-time equivalent of evaluating the bindings of a let-expression. The semantics actually suggests that each let-expression defines its own local heap fragment, that temporarily extends the global environment in rule NEST, and then gets merged with heap fragments of any surrounding contexts by means of MERGE steps. Uniqueness of the allocated pointers is implicitly guaranteed by the syntactic constraints on binding composition, and facilitated by the ability to alpha-rename local bindings in case of name clashes.

A real implementation will of course only maintain one global heap, that offers allocation of memory blocks whose addresses are unique by construction. The primary reason our semantics does not model this global heap directly – by viewing evaluation as a relation between heap/expression pairs, for example – is that such an approach would render the calculus less compositional. Moreover,

$$[\![\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e_{n+1}]\!] =$$

$$
\begin{aligned}
&\tau_1 \; x_1 = \xi_1; \\
&\ldots \\
&\tau_n \; x_n = \xi_n; \\
&x_1 = [\![e_1]\!]; \; subst(\theta_1, x_1); \\
&\ldots \\
&x_n = [\![e_n]\!]; \; subst(\theta_n, x_1, \ldots, x_n); \\
&\textbf{return } [\![e_{n+1}]\!];
\end{aligned}
$$

where $\xi_1 \ldots \xi_n$ are unique illegal addresses
and $\theta_i = [x_1/\xi_1, \ldots, x_i/\xi_i]$

Figure 6. Implementing evaluation of recursive bindings

a call-by-value semantics would also have to make inner bindings appear in the global heap before any outer bindings do, and thus require rather involved scoping rules for heap/expression pairs in order to capture the evaluation of nested let-bindings correctly.

Now, consider the let expression

$$\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e_{n+1}$$

If it were not for the complications of recursion, this expression could be represented in some C-like implementation language as follows (using τ_i to stand for the run-time representation type of x_i, and the notation $[\![e_i]\!]$ to denote the translated code for e_i):

$$
\begin{aligned}
&\tau_1 \; x_1 = [\![e_1]\!]; \\
&\ldots \\
&\tau_n \; x_n = [\![e_n]\!]; \\
&\textbf{return } [\![e_{n+1}]\!];
\end{aligned}
$$

However, because of recursion, each of the e_i may refer to *any* of the x_i. Some of these references will be backward references (i.e., an e_i referring to some x_j with $j < i$), which are acceptable as they stand. But we also need to be able to provide adequate values for the variables that are not backward references. Here is where our technique of using illegal addresses as initial pointer values comes into play.

An illegal address can be passed around as an argument and stored in data structures just like a regular pointer; although any attempt to dereference it – i.e., to read from the memory it points to – will result in a hardware-detected exception. We will use such an exception as the run-time equivalent of getting stuck due to ill-defined recursion. Of course, this is only going to be possible on architectures with a sufficiently large address space to allow for some addresses to be invalid (the odd ones, for example, or simply addresses outside the allocated virtual memory space), but we will blatantly ignore the perspective of really resource-constrained computing systems in the scope of this current paper.

Now, assuming that we can obtain a set of distinct illegal addresses $\xi_1, \ldots, \xi_n$, we propose to implement the evaluation of generic let-expressions according to Figure 6.

That is, we first initialize each let-bound variable with its corresponding illegal address before we compute the actual right-hand values in sequence, with each computation step followed by a *substitution* operation that replaces the illegal addresses it finds with the corresponding pointer into the heap.

The specification of function $subst(\theta, x_1, \ldots, x_k)$ is to destructively apply substitution θ to the data structures reachable from each of the roots $x_1 \ldots x_k$. It must thus be able to traverse arbitrary heap-allocated structures and find all contained pointers, just like a garbage collector does. Fortunately, because we are assuming a garbage collected heap implementation, the infrastructure for doing so (pointer/scalar distinction, node layout information) must already be in place.

Dynamic state in the form of one *visited* bit per node must also be available to *subst*, in order to ensure terminating traversals of the possibly cyclic heap structures. However, because garbage-collection activities may possibly interrupt recursive binding evaluations – and thus intersperse the calls to *subst* – this bit must be distinct from any dynamic state used by the garbage collector proper.

Let-expressions may be semantically nested, thus *subst* might encounter illegal addresses that actually belong to some other let-expression already under evaluation (i.e., illegal addresses not in the domain of the given θ). To make sure even the illegal addresses are unique, a global counter pointing out the next free illegal address should be used, that gets incremented and decremented in a stack-like fashion.

In principle, the *visited* bits should also be reset after each call to *subst*, in order to not confuse subsequent calls, or ongoing binding evaluations on an outer level. That work can be limited to just one final traversal of only those structures that are visited an odd number of times by *subst*, though, if both the memory allocator and *subst* are set up to toggle the logic interpretation of the *visited* bit between each call.

Several optimizations to this implementation scheme are possible. Dependency analysis should of course be applied prior to compilation to ensure that binding groups are really recursive and as small as possible. Ordinary function bindings may be lambda-lifted to the top level and need not participate in this kind of value recursion. Bindings that are record- or datatype constructions only referenced from weak value contexts can be allocated before any initializations, thus removing the need to do any substitutions on their addresses. And many *subst* calls can furthermore be safely skipped if the calls are guided by the following statically computable conditions:

$$
\begin{aligned}
skipsubst_i &= fragile_i \setminus substituted_i = \emptyset \\
fragile_i &= \bigcup \{ fwrefs_j \cap upto_i \mid x_j \in bwrefs_{i+1} \} \\
bwrefs_i &= fv(e_i) \cap \{x_1, \ldots, x_{i-1}\} \\
fwrefs_i &= fv(e_i) \cap \{x_i, \ldots, x_n\} \\
upto_i &= \{x_1, \ldots, x_i\} \\
substituted_i &= \bigcup \{ upto_j \mid 1 \le j < i, \neg skipsubst_j \}
\end{aligned}
$$

The *fragile* variables at position i are the forward referencing variables potentially reachable from the next right-hand side to be evaluated (position $i+1$), but which according to the semantics can be assumed to be fully evaluated at that point (i.e., should no longer be represented by illegal pointers). If this set – which typically is smaller than the complete set of variables defined up to point i – is fully contained in the set of variables already updated by previous *subst* calls, the call at point i can be safely skipped.

It can be noted that purity of expression evaluation plays an important role in the described implementation scheme. If evaluation of right-hand sides could cause arbitrary mutations, it would not in general be possible to limit the search for illegal addresses to the structures defined in the current recursive binding – in the worst case the whole live heap would have to be scanned by *subst*, just like a full garbage collection.

It should furthermore be observed that the concurrent execution of *threads*, that each may evaluate recursive bindings according to our scheme, offers no new complications, even though they might be sharing the same heap. This is because any data sharing between threads in a purely functional language must be handled by the primitives of a *monadic extension* to the language, and the evaluation of our recursive bindings involves just pure expressions. We might even use separate instances of the "unique-illegal-address-pointer" for each thread, simply because any two threads will only be able to exchange cyclic data once their bindings have been fully evaluated.

6. Performance

Giving quantitative performance measurements of our technique using real programs as examples is hard, because our proposal is fundamentally about *extending* the set of programs that can be compiled under call-by-value. In short, common benchmarks that emphasize the performance of value recursion techniques are simply not present. Moreover, even if both benchmarks and alternative implementations were readily available, isolating the impact of other code generation aspects would still be a daunting task, simply because the whole point of supporting unrestricted recursion is to allow cyclic memory allocation to be intertwined with general computations.

What we can do at this point, however, is to provide a *qualitative* assessment of the performance of our implementation technique. We can note the following:

- The use of illegal pointers and computer hardware to trap ill-founded recursion allows data inspection to be compiled without any additional cost whatsoever compared to implementations that offer no value recursion support. In particular, the C code for a field selection $x.l$ can be x->l, a case expression **case** x **of** ... can be compiled as switch (x->tag) ..., and the translation of a function call $x\ arg$ can be (*x)(arg). This holds irrespective of whether the compiled code may be invoked during the initialization of a recursive binding or not.

- A *subst* call traverses the given roots in much the same way as the scan phase (or mark phase) of a garbage collector does. The cost of those two run-time operations should thus be closely comparable. Furthermore, assuming the *skipsubst* optimizations described in the previous section, *subst* will only be called once for every cycle in the dependency graph between let-bound variables. Specifically, no cycles implies no additional cost because of *subst*.

- The cost for creating cyclic data – i.e., the cost for *subst* calls – must be added to the cost for accessing the cyclic data later on. Because of our zero cost accesses, our implementation technique can always be shown to outperform techniques based on indirections etc, simply by requiring the data to be accessed a sufficiently large number of times. And in our experience, cyclic data structures are created to be used extensively.

- The cost for a *subst* call should also be put against the number of times a cyclic data structure is expected to be garbage-collected. That is, each time a cyclic structure survives a garbage collection (of the whole heap), the relative cost of one additional scan of that structure (i.e., the cost of an initial *subst* call) will drop towards zero.

The above assessment is based on two successful implementations of our ideas, of which one is a model compiler to a small FAM-like abstract machine (Cardelli 1984), and the other one being our full-scale compiler for the Timber language (Nordlander et al. 2008). Timber is a strict purely functional language that allows programs to be structured as graphs of concurrent reactive objects that encapsulate monadic state, and our original motivation for the work in this paper was actually to make it possible to define these constant structures in a declarative manner, rather than having to resort to imperative methods where mutation is not desired.

However, space does not permit us to include any elaboration on the Timber language in this paper. In the next section we will instead study the expressiveness of our proposal in the terms of an application area more well-known to Haskell programmers: parsing combinators.

7. A combinator parser example

Combinator parsers are widely popular in Haskell, but not so much in strict functional languages.

In particular, parsing combinators that are designed using *applicative functors* (McBride and Paterson 2008) allows us to write Haskell parsers in a very concise style. Can this style be carried over to a call-by-value language extended according to our proposal? In this section we will see that the answer is yes.

In the following example, we will use the type $P\ a$ to denote parsers of a; i.e., parsers that return semantic values of type a upon a successful parse. The key combinators are

- *accept* :: $[Char] \rightarrow P\ Char$: accepts one character from a given set,

- *return* :: $a \rightarrow P\ a$: succeeds without consuming any input,

- ($\$\$$) :: $P\ (a \rightarrow b) \rightarrow P\ a \rightarrow P\ b$: sequentially composes two parsers, and

- ($\$+$) :: $P\ a \rightarrow P\ a \rightarrow P\ a$: expresses an alternative of two parsers.

Using these combinators, and the derived combinator *parens* for building parenthesizing parsers, we can express the parser alluded to in the introduction section as follows, complete with semantic actions for building an abstract syntax tree:

$$\textbf{data}\ Exp = EOp\ Var\ Op\ Exp$$
$$|\ \ EVar$$

$$\textbf{data}\ Var = Var\ Char$$

$$\textbf{data}\ Op = Op\ Char$$

$$
\begin{aligned}
pExp\ =\ &return\ EOp\ \$\$\ pVar\ \$\$\ pOp\ \$\$\ pExp\\
\$+\ &return\ EVar\ \$\$\ pVar\\
\$+\ &\qquad\qquad\quad\ parens\ pExp
\end{aligned}
$$

$$
\begin{aligned}
pVar\ &=\ return\ Var\ \ \$\$\ accept\ [\ 'A'\ ..\ 'z']\\
pOp\ &=\ return\ Op\ \ \ \$\$\ accept\ [\ '+',\ '-',\ '*',\ '/']
\end{aligned}
$$

Note that *pExp* is a recursive definition that is not on value form; hence the definition would be invalid in both OCaml and SML. One could try to work around this limitation by explicitly parameterizing every parser over the input token stream, thus turning every parser into a function value. However, this solution would break the abstraction barrier provided by P. Moreover, there are useful parser implementations that are not functions over the input token stream.

One prominent example of such parsers is Swierstra's and Duponcheel's self-optimizing parsing combinators (Swierstra and Duponcheel 1996). The structure of these parsers typically includes a static part that can be computed before applying the parser on any input, just by analyzing the grammar. The representation of a self-optimizing parser must therefore be a data structure; in general a *recursive* data structure not unlike the NFAs that were discussed in Section 2.

Here is an implementation of the abstract parser type $P\ a$ that supports a simple form of self-optimization.

$$\textbf{type}\ P\ a = (Maybe\ a,\ [(Char, String \rightarrow (String, Maybe\ a))])$$

The first component of such a parser is of the form *Just a* if the parser can accept the empty string, and the second component is a mapping from the first characters of the non-empty strings the parser can accept to functions for parsing the rest of the input. These functions return unconsumed input and a semantic value upon success. Both the empty-string case and the mapping for the non-empty case can be computed statically, before applying the parser to any input.

It should now be clear how we can implement *return* and *accept*:

$$
\begin{aligned}
return\ a\ &=\ (Just\ a,\ [])\\
accept\ cs\ &=\ (Nothing,\ [(c, \lambda s.(s, Just\ c))\ |\ c \leftarrow cs])
\end{aligned}
$$

The combinators for sequential and parallel composition are a little more involved, so we just outline the sequential case, which is the interesting one from the point of recursive values. The reason can be seen in the definition of *pExp* above: here we have a variable defined in terms of itself (as *pExp* also appears as the second argument to $\$\$$). This means that when we implement $\$\$$ we have to be a little careful about what we do with our second argument. Here is the suggested implementation:

$$
\begin{aligned}
fp\ \$\$\ ap = (empty, nonempty)\ &\textbf{where}\\
empty = \textbf{case}\ &fst\ fp\ \textbf{of}\\
&Nothing \rightarrow Nothing\\
&Just\ f \rightarrow \textbf{case}\ fst\ ap\ \textbf{of}\\
&\qquad\qquad Nothing \rightarrow Nothing\\
&\qquad\qquad Just\ a \rightarrow Just\ (f\ a)\\
nonempty = combineSeq\ &fp\ ap
\end{aligned}
$$

The termination subtleties are reflected in the *Just* branch in the definition of *empty*: we must only inspect argument *ap* if the first parser can accept the empty string! The *nonempty* case is handled by the auxiliary *combineSeq*, in which similar considerations apply.

As an illustration to the issues that must be mastered even when writing combinator parsers in Haskell, consider the following example:

$$p = return\ (1:)\ \$\$\ p$$

What this definition actually says is "in order to parse a p, parse a p, and then prepend 1 to the result". That is, p is a parser that will always diverge, but we accept this behavior even in Haskell because a top-down parser implementing a left-recursive grammar is not expected to work anyway.

However, the observant Haskell programmer might have noticed by now that the local definition of *empty* above could have been written more concisely as

$$empty = fst\ fp\ \ \$\$_{Maybe}\ \ fst\ ap$$

where

$$\$\$_{Maybe}\ ::\ Maybe\ (a \rightarrow b) \rightarrow Maybe\ a \rightarrow Maybe\ b$$

is the applicative functor for the *Maybe* type. Unfortunately, this shortened version of *empty* now always forces inspection of the second argument to $\$\$$, since function arguments are evaluated eagerly in a strict language. That is, our definition of *empty* has become too strict to be used in $\$\$$, which in turn has become too strict to be used in even right-recursive parsers!

The fundamental programming difficulty encountered here cannot be ignored by any user of a call-by-value language. Call-by-value evaluation allows abstraction over values only, not general computations, so the occasional need to circumvent this limitation by means of less than ideal encodings should come as no surprise. Avoiding abstraction by inlining a function definition at its call site is of course a very natural solution, as it effectively delays the computation of problematic argument expressions in a call-by-name like manner (this is in fact exactly how we constructed our first definition of *empty* above).

However, from the point of view of a trained call-by-value programmer, the problematic computations to look out for when applying abstractions are those that might fail to terminate, or may cause exceptions like division-by-zero – i.e., expressions with undesirable but still well-known computational effects. What our shift into the realm of unrestricted call-by-value recursion is causing is actually a new form of computations that might be at odds with abstraction: plain inspection of data denoted by innocent-looking

variables. It is therefore reasonable to ask to what extent programmers in such an extended language should be expected to take the possibility of ill-founded recursion into account. For example, should the implementor of a non-recursive function really have to foresee all possible uses for it inside future recursive bindings?

A Haskell programmer would probably answer this question negatively, but at the same time implicitly argue that lazy evaluation is the logical step forward. For an ML programer, to whom the predicatibity and efficiency of call-by-value evaluation is a fundamental requirement, the answer must become less clear-cut. Ideally, a type system or some other static analysis method should be available to catch all uses of ill-founded recursion at compile-time (and as should be evident from the discussion on left-recursive grammars above, such a tool would be of benefit in a Haskell context as well). But even so, it seems like the particular combination of unrestricted recursion and call-by-value evaluation has its own ramifications, and a better understanding of both the sources of ill-foundedness as well as ways to avoid it would be of great practical importance.

In the next section we will present one investigation of this topic.

8. Delayed selection

In the basic calculus extended with records and algebraic datatypes, it is only when a variable appears as the right-hand side of a binding, or as the head of an application, field selection, or case analysis expression, that a VAR step is absolutely necessary in order to avoid a stuck evaluation state. Consequently, an expression that is stuck because of an attempt to dereference a yet undefined variable must be a let-expression encapsulating a term of one of these forms.

If we ignore the first alternative for a moment, we can succinctly capture the form of a stuck expression as

$$\mathbf{let}\,\Gamma, x = \mathcal{E}^*[\mathcal{S}[y]], b\,\mathbf{in}\,e$$

where $\mathcal{E}^*$ means the composition of zero or more evaluation contexts, $y \in \{x\} \cup bv(b)$, and $\mathcal{S}$ is a *selection context* defined as

$$\mathcal{S} \quad ::= \quad [\,]\,e \; \mid \; \mathbf{case}\,[\,]\,\mathbf{of}\,\{k_i\,\overline{x}_i \to e_i\} \; \mid \; [\,].l$$

A concrete example of a stuck expression can be found in Section 4, example (13):

$$\mathbf{let}\,x = \{l_h = y.l_h, l_t = y\}, y = \{l_h = 2, l_t = x\}\,\mathbf{in}\,e$$

Here x can not be evaluated further because the sub-expression $y.l_h$ needs to reduce to a value and y is a forward reference. Note especially that $y.l_h$ matches the third alternative in the definition of $\mathcal{S}$ above, and that the $\mathcal{E}^*$ instance in this case is just the simple evaluation context $\{l_h = [\,], l_t = y\}$.

Now, an interesting aspect about terms of the above form is that they can always be rewritten so that the immediate reason behind the stuck condition is removed! To see this, notice that the following example actually works, even though it is equal to the previous one except for the introduction of an intermediate binding.

$$\Gamma \vdash \mathbf{let}\,x = \{l_h = z, l_t = y\}, y = \{l_h = 2, l_t = x\}, z = y.l_h\,\mathbf{in}\,e \;\to^*$$
$$\mathbf{let}\,x = \{l_h = z, l_t = y\}, y = \{l_h = 2, l_t = x\}, z = 2\,\mathbf{in}\,e$$

This expression is not stuck anymore because the previous selection expression $y.l_h$ has been replaced with the variable z, which in turn has turned the right-hand side of x into a value. And since y is a value as well, the evaluator can go on and reduce the old selection expression $y.l_h$ – that is, the right-hand of z – to 2.

What we see here is the beginning of a systematic approach to resolving problematic recursive dependencies: when selection from a forward reference is required, put the selection computation in a binding that is even more forward, and move on with a reference
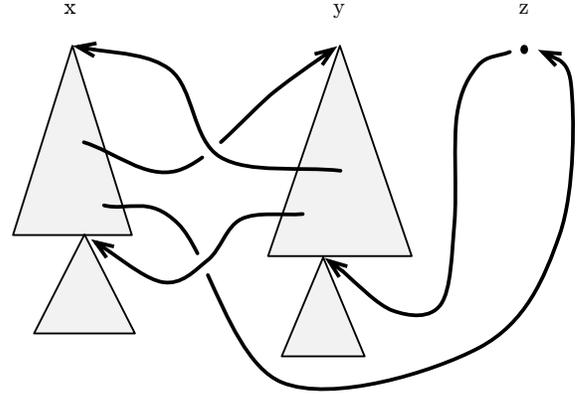
to that binding instead. Figure 7 illustrates the general form of the resulting cyclic data structure. Note especially how this graph relates to the one in Figure 5.

We can go further and formalize the idea above as a hypothetical rewrite rule, and we will see that it can be used to transform programs that otherwise would be stuck.

$$\frac{y \in (\{x\} \cup bv(b)) \backslash \, bv(\mathcal{E}^*) \qquad z\ \text{fresh}}{\begin{array}{l}\mathbf{let}\,\Gamma, x = \mathcal{E}^*[\mathcal{S}[y]], b\,\mathbf{in}\,e \,\rightsquigarrow \\ \mathbf{let}\,\Gamma, x = \mathcal{E}^*[z], b, z = \mathcal{S}[y]\,\mathbf{in}\,e\end{array}} \; \text{Delay1}$$

The remaining form of a stuck expression – a forward reference appearing directly as the right-hand side of a let-binding – can also be addressed by means of a hypthetical rewrite rule. We first define the context of a right-hand side in focus as follows:

$$\mathcal{R} \quad ::= \quad [\,] \; \mid \; \mathcal{E}^*[\mathbf{let}\,\Gamma, x = [\,], b\,\mathbf{in}\,e]$$

The suggested rewrite step can then be formulated as a simple reordering of bindings.

$$\frac{y \in (\{x\} \cup bv(b)) \backslash \, bv(\mathcal{R})}{\mathbf{let}\,\Gamma, x = \mathcal{R}[y], b\,\mathbf{in}\,e \rightsquigarrow \mathbf{let}\,\Gamma, b, x = \mathcal{R}[y]\,\mathbf{in}\,e} \; \text{Delay2}$$

We call these rewrite rules DELAY1 and DELAY2, because what they achieve is actually a delay of the problematic dereferencing of a yet undefined variable. In fact, DELAY1 and DELAY2 capture *all* forms of expressions in our calculus that are stuck due to ill-founded definitions. They do not, however, offer a means to eliminate ill-founded recursion altogether, as rule DELAY2 might actually introduce one problematic forward reference for each one it removes. In particular, the case where y is identical to x is matched by rule DELAY2, but obviously not improved by it.

Moreover, it should be noted that these rewrite rules only apply to terms that are stuck as they are, not to terms that might evaluate into a stuck state. To some extent this could be addressed by just generalizing the rules a bit, for example by replacing the Γ component in both rules with some unevaluated binding b'. In general, though, the effects of evaluation cannot be captured by simple syntactic means, primarily due to the drastic effect evaluation steps BETA and VAR might have on the structure of terms. For example, recall the parser combinator from Section 7:

$$\mathbf{let}\,pExp = \ldots\ \$\$\ pExp\,\mathbf{in}\,\ldots$$

This term is obviously not of a form matched by any of the delay rules above, although we know that if we define $\$\$$ using the applicative functor $\$\$_{Maybe}$ internally, evaluation of the recursive binding of *pExp* will get stuck.

Figure 7. Delayed selection in mutually recursive data structures

However, *after a few evaluation steps*, the term will indeed be of the form expected by DELAY1:

$$\textbf{let } pExp = \textbf{let } empty \quad = \dots \ \$\$_{Maybe}$$
$$(\textbf{case } pExp \textbf{ of } (x, \_) \to x)$$
$$nonempty = \dots$$
$$\textbf{in } (empty, nonempty)$$
$$\textbf{in } \dots$$

Note especially that evaluation is now stuck because of the attempted case analysis of *pExp*, and that this sub-expression matches the second form of $\mathcal{S}$ contexts previously defined. Had only the *pExp* combinator looked like this originally, we could have rewritten it using rule DELAY1 and obtained the following unproblematic term:

$$\textbf{let } pExp = \textbf{let } empty \quad = \dots \ \$\$_{Maybe} \ z$$
$$nonempty = \dots$$
$$\textbf{in } (empty, nonempty)$$
$$z \qquad = \textbf{case } pExp \textbf{ of } (x, \_) \to x$$
$$\textbf{in } \dots$$

So what this example really illustrates is that in order to be as applicable as possible, the hypothetical rewrite rules we have defined should ideally be promoted to fully-fledged evaluation rules. Such a move would of course alter the semantics of our calculus in a non-trivial way, leading to a less predictable evaluation order, but also to greater flexibility in the use of abstractions inside recursive bindings. Moreover, the ability to execute the DELAY rules dynamically would render our calculus insensitive to the order in which bindings appear inside let-expressions, in a way quite reminiscent of lazy evaluation.

Still, extending our calculus with the DELAY rules would not mean reinventing the notion of lazy evaluation on the whole. All function arguments would still be evaluated eventually, thus keeping infinite computations and functions like *enumFrom* discussed in Section 2 as incompatible with the extended calculus as with any call-by-value regime. And variables would still denote just values, not arbitrary computations, thereby preserving the predictability of expression evaluation in all contexts except for the initialization of recursive bindings.

Even so, implementing the DELAY rules efficiently in a compiled setting seems to pose some very specific challenges not covered by the techniques we have developed in this paper. We therefore believe that a calculus with dynamic delay rules constitutes a unique and interesting point of its own in the design space between call-by-value and lazy evaluation. Although we do have an implementation completed even for the extended calculus, we are not in a position to report on its properties within the scope of the current work. Instead we hope to be able to do so within the near future.

9. Related work

The earliest example of call-by-value recursion with relaxed right-hand sides is Scheme (Kelsey et al. 1998). Scheme's letrec construct is implemented using reference cells initialized to a void value, that get implicitly dereferenced wherever they occur in a context that is not a lambda abstraction. This means that cyclic data bindings like $x = k \, (\lambda y.x)$ are accepted, whereas a definition like $x = k \, x$ leads to a run-time error because x must be dereferenced during evaluation of the right-hand side.

Boudol and Zimmer describe an abstract machine for a call-by-value calculus with letrec, that also uses reference cells for tying the recursive knot, but where the dereferencing operation is postponed until the value of a variable is actually needed (Boudol and Zimmer 2002). Their calculus thus allows unrestricted use of function calls in the right-hand sides of recursive definitions, but the price is an extra level of indirection for variable accesses.

Our implementation method achieves the same effect without the indirections, by using illegal addresses instead of empty reference cells. The data traversals we use for tying the knot are obviously more costly than simply updating reference cells, but it is a once-only cost compared to the recurring burden of having to access variables through indirections.

The work closest in spirit to ours is the compilation method for extended recursion developed by Hirschowitz et al. and partially made available in the Objective Caml compiler (OCaml). The basis of their work is also a call-by-value calculus where evaluated recursive let-bindings take on the role of a dynamic heap, where subsequent bindings in a recursive group can see the result of previous ones, and where variables count as values (although not distinctively *weak* ones) (Hirschowitz et al. 2003). Like us they are also able to avoid indirections. Our semantics has a considerably simpler formulation and comes with a referential transparency property, although they provide a formal correctness proof of their translation into an intermediate language with explicit pointers. However, their implementation method for recursive bindings is significantly different from ours. In principle, Hirschowitz et al. resolve forward pointers by allocating the topmost node of the corresponding right-hand sides *before* any bindings are evaluated. When the real value of such a binding eventually becomes available, its topmost node is copied over to the pre-allocated block, which then takes over the role of the "real" value of the binding. Because run-time accesses to an uninitialized dummy block cannot be distinguished from valid ones, compile-time restrictions must be imposed on the right-hand sides of bindings with forward references to them; in (Hirschowitz et al. 2003) this is assumed to be taken care of by a static analysis similar to the one reported in (Hirschowitz and Leroy 2002). The pre-allocation method also requires the sizes of objects to be statically known, which can be a severe restriction if one intends to define dynamic arrays, records coerced by subtyping, or function closures in recursive bindings. The system we propose shares none of these limitations. On the other hand, copying of the topmost node only is likely to be faster in general than the substitution traversals we use, although it is certainly also possible to imagine cases where the opposite is true.

Syme presents an approach to initializing relaxed recursive bindings in ML (Syme 2006), that essentially amounts to *delaying* the right-hand sides and *forcing* the left-hand variables wherever they appear, relying on lambda abstractions to break what would otherwise be ill-founded cyclic dependies. Thus, Syme's system does not directly support recursive data structures as in $x = k \, x$, nor does it allow abstraction over yet unevaluated names as in $x = f \, x$ (in both these cases, Syme's technique will result in eager evaluation of the right-hand sides before x is bound). What it does provide, however, is order independence between bindings in a recursive group, which Syme puts to good use in his exposition of programming patterns that become enabled just because of the ability to do computations in the right-hand sides. We do not have independence of binding order in our basic calculus, although our tentative extension with the DELAY rules appears to achieve a similar effect. A thorough comparison between this extension and the delaying and forcing primitives of ML is indeed an important topic for future work.

Several researchers have proposed type-based approaches for capturing ill-founded recursion at compile-time. Boudol and Zimmer's calculus comes with a type system that decorates the function type with a boolean flag indicating whether the function body is strict in its argument (Boudol and Zimmer 2002). Hirschowitz and Leroy provide a slightly generalized typing logic, where the free variables of a term are associated with integers indicating the number of lambda abstractions preventing the value of the variable from being demanded (Hirschowitz and Leroy 2002). Dreyer goes

even further and annotates the function type with a set of names that must be fully evaluated if a call to a function of that type is going to succeed (Dreyer 2004). In work more specifically aimed at imperative programming in object-oriented languages, Fähndrich and Xia introduce a subtyping system where the earliest time a variable can be dereferenced is captured in its type (Fahndrich and Xia 2007). These systems are all able to guarantee the absence of ill-founded recursion at run-time, using varying degrees of conservative approximation. Our work does not yet offer any static means for outruling ill-founded recursion, but it could quite possibly be extended with the type systems proposed in any of the cited works.

Our operational semantics for call-by-value recursion is influenced by the work by Ariola and Blom on various lambda calculi with a letrec construct (Ariola and Blom 2002), especially in the use of variables as values and the treatment of value bindings floating outwards in rule MERGE. The system defined by Ariola and Blom is much more complex than our semantics, primarily because their interest is to study general equational theories of cyclic lambda terms, and partly because they are essentially working in a generic framework which they instantiate to different kinds of lambda calculi (of which the call-by-value calculus is just one example). Our confluence result is probably related to their *confluence up to information content*, but we have not yet been able to work out any details.

The technique we use to prohibit dereferencing of variables that are in the process of being defined is a loan from Launchbury's seminal paper on natural semantics for lazy evaluation (Launchbury 1993). That paper also provided the idea to treat lambda- and let-bound variables in fundamentally different ways in the reduction axioms. We believe that a closer comparison between our respective systems is likely to reveal many useful insights, especially regarding the system obtained when we introduced the DELAY rules in Section 8.

10.   Conclusion and further work

In this paper we have introduced an operational semantics for a call-by-value language that allows recursive bindings to (1) define non-functional data and (2) freely use functional computations in their right-hand sides. We have shown that the semantics is referentially transparent and that it has an efficient implementation, where recursively bound variables in the process of being defined are represented as illegal addresses. The key to efficient data access lies in our reliance on the computer hardware to detect eager dereferencing of such data. Our implementation technique implies taking the cost of recursive knot-tying at definition time instead, by making a substitution traversal of the created memory structures.

We have shown that Haskell-style programming which crucially depends on unrestricted recursion can be straightforwardly expressed under call-by-value, in particular an implementation of a non-trivial combinator library for parsing. We have also identified the need for program rewrites that delays overly eager selections in some special cases, and the generality of our proposed solution has pointed us towards a tentative extension to our semantics that would achieve the effect of these rewrites automatically at run-time.

Avenues for further work include exploiting the implications of the DELAY extension in more depth, especially its implementation consequences and its relation to full-blown lazy evaluation. We would also like to study the denotational aspects of call-by-value recursion in more depth, with the hope of finding a succinct model of the infinite, yet highly regular forms of data our language allows us to compute.

References

Z. M. Ariola and S. Blom. Skew conuence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117((13)):95178, 2002.

Gérard Boudol and Pascal Zimmer. Recursion in the call-by-value λ-calculus. In *Fixed Points in Computer Science, FICS 2002*, Copenhagen, Denmark, July 2002.

Luca Cardelli. Compiling a functional language. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 208–217, New York, NY, USA, 1984. ACM.

D. Dreyer. A type system for well-founded recursion. In *Conference Record of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice, Italy, 2004.

Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, New York, NY, USA, 1997. ACM.

Manuel Fahndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 337–350, New York, NY, USA, 2007. ACM.

T. Hirschowitz, X. Leroy, and J. Wells. Compilation of extended recursion in call-by-value functional languages. In *Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.

Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002.

Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised[5] report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33 (9):26–76, 1998.

John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Johan Nordlander, Magnus Carlsson, Andy Gill, Per Lindgren, and Björn von Sydow. The Timber home page, 2008. URL `http://timber-lang.org`.

OCaml. Objective Caml. `http://caml.inria.fr/ocaml/`.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, pages 184–207, 1996.

Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electr. Notes Theor. Comput. Sci.*, 148(2):3–25, 2006.