

Lazy modules

Keiko Nakata*

Institute of Cybernetics, Tallinn University of Technology

Abstract

We investigate evaluation strategies for ML-style modules supporting recursion. More precisely we propose and examine five evaluation strategies: a call-by-value strategy and four call-by-need strategies with different degrees of laziness. We formalize the strategies by translating a source syntax for modules into target languages, which are very much inspired by Felleisen and Hieb's call-by-value lambda calculus with state and Ariola and Felleisen's call-by-need cyclic lambda calculus. Different strategies are expressed by tweaking the translation as well as the operational semantics of the target languages. We look at the strategies through a series of examples and state inclusion between the strategies.

1 Introduction

Modern programming languages have mechanisms for promoting modular programming. For instance objects encapsulate methods which operate on object states. F# (Syme & Margetson, 2008) has a module system on top of its object system for better support of large program structuring. The ML module system is well-known for its strong support of modular programming (Milner *et al.*, 1997a; Leroy, 2000). OCaml (Leroy *et al.*, 2008) and Moby (Fisher & Reppy, 1999) incorporate both the ML module system and object systems in a single language. In the case of Scala, its object system is expressive enough that it serves as both an object system and a ML-like module system (Odersky *et al.*, 2003).

Common to all the above mentioned languages is support of nestable structure, namely the ability to decompose a program into hierarchy. Nesting is a powerful way to organize program code and namespaces that should be simple and intuitive for any programmer. An abbreviation mechanism, or the ability to make aliases for modules, is also common. For instance it is useful to give a module a meaningful name, like `Hashtable`. At the same time, it is often convenient to locally refer to the module with an abbreviation, like `H`, when the referred module is clear from the context.

Support for features beyond nesting and abbreviations varies among languages and some features are closely tied to the evaluation strategies of the underlying languages. Below we compare two opposites.

Object systems support recursive class definitions by nature. In the case of Java (Gosling *et al.*, 2005), classes are initialized lazily in accordance with its dynamic class loading

* The early development of this work is done when the author was a post-doc at CNAM/INRIA .

mechanism. In object systems safety of object initialization is typically checked at runtime by signaling exceptions; at creation time fields of an object are initialized to contain default values, which are supposed to be initialized to meaningful values afterward, for instance through constructor calls. While the problem of object initialization is notorious, promising efforts have been made to eliminate unsafe initialization patterns at compile time (Fähndrich & Xia, 2007; Qi & Myers, 2009). An optimizing compiler may remove redundant null-checks (Scales *et al.*, 2000). It is noteworthy that in object systems there is a clear distinction between methods and fields. Moreover a method of an object may be invoked during its initialization and a field of an object may be safely read immediately after the field is initialized but before the object is fully initialized.

The ML module system comprises structures and functors. A structure encloses type and value definitions, which we call core fields of the structure, and possibly sub-structures, which we call module fields of the structure¹. Functors are functions on structures. They facilitate code reuse in a modular way. ML modules are initialized eagerly, or in call-by-value: all the core fields of a structure, including those of the sub-structures, are evaluated at once following the textual definition order in the source program. The simple evaluation order of call-by-value is appreciated in the presence of arbitrary side effects, which the ML core language permits. Since traditionally ML does not support recursively defined structures or functors and since recursive value definitions are syntactically restricted, module initialization is always successful. This enhances not only safety but also efficiency. Since references to fields of structures are necessarily proper values, for instance, no null-check is required. ML enjoys this property at the cost of difficulties in extending the module system with recursion or dynamic loading of modules.

Being ML programmers, we expect a module system to support all the familiar features of ML modules as well as flexible recursion between modules as afforded by object systems. Much work has been devoted to design type systems for ML-style modules supporting recursion (Crary *et al.*, 1999; Russo, 2001; Dreyer, 2005; Nakata & Garrigue, 2006). As for initialization, previous work proposed type systems which statically guarantee safe initialization of recursively defined modules in a call-by-value setting (Boudol, 2004; Hirschowitz & Leroy, 2005; Dreyer, 2004). Both OCaml and Moscow ML (Romanenko *et al.*, 2004) support recursive module extensions, where initialization safety is checked at runtime by signaling exceptions in case of “unsupported” recursive initialization patterns. Interestingly OCaml and Moscow ML implement recursive modules in different ways; as a result, they permit different initialization patterns.

In this paper we study different evaluation strategies for ML-style modules supporting recursion. Concretely we propose and examine five evaluation strategies: a call-by-value strategy and four call-by-need strategies with different degrees of laziness. We look at the strategies in depth through a series of examples. In particular we deploy examples of taking fix-points of functors as a central guideline for judging flexibility of a strategy in handling recursive initialization patterns. The ability to take fix-points of functors is a crucial motivation for introducing recursive modules; it affords separate and extensible

¹ To be precise a structure may contain functors and functors may be higher-order.

<i>Module expressions</i>	$E ::= \{(X) f\} \mid p \mid \Lambda X.E \mid E_1(E_2)$
<i>Definitions</i>	$f ::= \varepsilon \mid M = E; f \mid c = e; f$
<i>Module paths</i>	$p ::= X \mid M \mid p.n$
<i>Core expressions</i>	$e ::= c \mid p.n \mid \dots$

Fig. 1. Syntax for *Osan*

development of recursively defined modules (Crary *et al.*, 1999; Russo, 2001; Nakata & Garrigue, 2006).

We formalize the strategies by translating a source syntax for modules into target languages. The translation is very much inspired by an intermediate phase of the OCaml compiler, which translates a typed abstract syntax for modules into a lower-level lambda-like syntax. The target languages are very much inspired by Felleisen and Hieb’s call-by-value lambda calculus with state (Felleisen & Hieb, 1992) and Ariola and Felleisen’s call-by-need cyclic lambda calculus (Ariola & Felleisen, 1997). We express different strategies by tweaking the translation as well as the operational semantics of the target languages and state inclusion between the strategies.

The contributions of the paper are summarized as follows:

1. We examine five evaluation strategies for ML-style modules supporting recursion by gradually introducing laziness, and formalize the strategies by translating a source syntax for modules into target languages equipped with call-by-value/call-by-need operational semantics.
2. We state inclusion between the strategies.
3. We investigate expressiveness of the strategies through a series of examples. To design a language, it is important to examine potential consequences of adopting a particular design choice. Hence, while this investigation does not yield any technical results, we believe it is a contribution of the paper.

The remainder of the paper is organized as follows. In Section 2 we define the source syntax for modules. Then we examine the five evaluation strategies in Sections 3, 4, 5, 6 and 7 in turn. In Section 8 we consider an extension of the target languages with state. We discuss related work in Section 9 to conclude in Section 10.

2 The source syntax *Osan*

In Figure 1 we define the source syntax, named *Osan*. We use *Osan* to introduce evaluation strategies we examine in the paper. Metavariables M , c , X and n range over module names, core names, module variables and positive integers, respectively. The notation ε denotes an empty sequence. A module expression is either a *recursive structure*, *module path*, *functor* or *functor application*. A recursive structure, simply called structure hereafter, $\{(X) f\}$ is a sequence of definitions with a declaration of a “self variable” X , which is bound to the structure $\{(X) f\}$. An expression in f can refer to any name defined in f through X . A definition either binds a module name to a module expression or a core name to a core expression. In addition to recursive references through self variables, *Osan* also supports ML’s traditional backward references; namely, a definition’s scope extends over the subsequent fields to the end of the structure. Thus a preceding field

may be referred to simply by its name or by going through a self variable. Moreover, these two ways of referring to preceding fields may have different semantics depending on the evaluation strategy, as we will see in the next section. For simplicity we assume the definition sequence of a structure does not bind the same name twice.

A module path is a reference to a module. It is either a module variable X , module name M or qualified identifier $p.n$, which accesses the n -th field of the structure that p refers to. In this paper we only consider well-typed programs where the type system ensures that the n -th field is a module expression, not a core expression. A functor $\Lambda X.E$ is a function on modules, where X is the parameter and E is the body. $E_1(E_2)$ denotes functor application.

Our formalization of evaluation strategies for modules is largely independent of the core language. We only assume expressions of the core language include core names c and *value paths* $p.n$, which accesses the n -th field of the structure that p refers to. Again we assume the type system ensures that the n -th field is a core expression.

In examples we use a programmer-friendly syntax for paths like $X.M$, instead of $X.n$, for module paths and $X.c$, instead of $X.n$, for value paths. That is, we use sequences of names, instead of indexes, punctuated by “dots” for paths. Indeed in the surface syntax the programmer would write these name-based paths, but the compiler has an elaboration phase, prior to runtime, to translate name-based paths into index-based paths. For the translation we only need to know the definition order in which core and module names are bound in a structure to map names to indexes. Such translation is standard in practical implementations of ML modules, and usual type checkers for ML modules can collect enough type information for this purpose. The translation is applicable in the presence of sealing with less informative signatures, using an implementation technique known as module thinning.

As alluded to in Introduction, we do not address typing issues in this paper. As far as type soundness is concerned, to the best of our knowledge, most type systems for ML modules, including at least our previous work on a recursive module extension (Nakata & Garrigue, 2006), are independent of whether modules are evaluated eagerly or lazily.

3 Call-by-value modules

We define evaluation strategies for *Osan* by translating the source syntax into target languages, which are very much inspired by Felleisen and Hieb’s call-by-value lambda calculus with state (Felleisen & Hieb, 1992) and Ariola and Felleisen’s call-by-need cyclic lambda calculus (Ariola & Felleisen, 1997). Several strategies are expressed by tweaking the translation as well as the operational semantics of the target languages.

3.1 The target language λ_{value} for call-by-value modules

We start by considering a call-by-value evaluation strategy. This subsection introduces the target language λ_{value} . The next subsection presents translation from *Osan* to λ_{value} , thus formalizes the call-by-value strategy for *Osan*.

Figure 2 defines the syntax of λ_{value} . The dotted notation denotes possibly empty sequences, e.g., a, \dots denotes an empty sequence or a_1, \dots, a_n where $n \geq 1$; \dots, a_n, \dots denotes a non-empty sequence whose n -th element is a_n . Expressions are variables, abstraction,

<i>Expressions</i>	$a ::= x \mid \lambda x. a \mid a_1 a_2 \mid \langle x \rangle \mid \{x, \dots\} \mid a!n \mid$ $\mid \text{let rec } d \text{ in } a$
<i>Values</i>	$v ::= \lambda x. a \mid \langle x \rangle \mid \{x, \dots\}$
<i>Definitions</i>	$d ::= x = a \text{ and } \dots$
<i>Value definitions</i>	$D ::= x = v \text{ and } \dots$
<i>Lift contexts</i>	$L ::= [] \mid a \mid []!n$
<i>Nested lift contexts</i>	$N ::= [] \mid L[N]$
<i>By-value evaluation contexts</i>	$C ::= D \vdash N \mid D \text{ and } x = N \text{ and } d \vdash a$

Fig. 2. Syntax for λ_{value}

<i>Contraction rules</i>	
$\beta_{\text{need}} :$	$(\lambda x. a) a' \xrightarrow{\text{value}} \text{let rec } x = a' \text{ in } a$
$\text{lift} :$	$L[\text{let rec } d \text{ in } a] \xrightarrow{\text{value}} \text{let rec } d \text{ in } L[a]$
<i>Reduction rules</i>	
$\text{cxt} :$	$C[a] \xrightarrow{\text{value}} C[a'] \text{ if } a \xrightarrow{\text{value}} a'$
$\text{deref}_{\text{value}} :$	$C[x] \xrightarrow{\text{value}} C[v] \text{ if } x = v \in C$
$\text{arr}_{\text{value}} :$	$C[\langle x \rangle!n] \xrightarrow{\text{value}} C[x_n] \text{ if } x = \{\dots, x_n, \dots\} \in C$
$\text{alloc}_{\text{value}} :$	$D \vdash \text{let rec } d \text{ in } a \xrightarrow{\text{value}} D \text{ and } d \vdash a$
$\text{alloc-env}_{\text{value}} :$	$D \text{ and } x = (\text{let rec } d \text{ in } a) \text{ and } d \vdash a'$ $\xrightarrow{\text{value}} D \text{ and } d \text{ and } x = a \text{ and } d \vdash a'$
<i>Access in evaluation contexts</i>	
$\text{acc}_{\text{value}} :$	$x = a \in D \vdash N \text{ if } x = a \in D$
$\text{acc}_{\text{back}} :$	$x = a \in D \text{ and } x = N \text{ and } d \vdash a' \text{ if } x = a \in D$

Fig. 3. Reduction rules for λ_{value}

application, *pointers* $\langle x \rangle$, *arrays* $\{x, \dots\}$, *accessors* $a!n$, and *letrec* $\text{let rec } d \text{ in } a$, where a is the body and d is bindings. The order of bindings is significant. That is, re-ordering of bindings in a *letrec* is not allowed. A value is either an abstraction, a pointer or an array. Expressions of λ_{value} feature three primitives for array manipulation. $\langle x \rangle$ represents a pointer to an expression referenced by x ; $\langle \cdot \rangle$ is reminiscent of the address-of operator $\&$, as provided in C. $\{x, \dots\}$ represents an array whose elements are referenced by x 's. $a!n$ represents access to the n -th element of the array pointed to by a .

We give an operational semantics for λ_{value} by defining a deterministic partial function on *configurations*; the formulation is also inspired by Hirschowitz *et al.*'s λ_o language (Hirschowitz *et al.*, 2003). A configuration $d \vdash a$ is a pair of bindings d and an expression a . We respectively call the bindings and the expression the *heap* and the *body* of the configuration. The order of bindings in a heap is significant. A *result* is a configuration $D \vdash v$ consisting of value definitions and a value.

The function is defined in terms of one-step reduction relations given in Figure 3. Rule β_{need} reduces an application to a *letrec*. Since bindings are evaluated eagerly, functions are call-by-value in the presence of β_{need} . An administrative rule lift lifts a *letrec* out of a lift context. Rule cxt performs contraction in the hole of an evaluation context. Rule $\text{deref}_{\text{value}}$ dereferences a variable from the heap. Rule $\text{arr}_{\text{value}}$ accesses the n -th element of the array referenced by x . Bindings in a heap are available only from the evaluated part

$\xrightarrow{\text{value}}$	$\vdash \text{let rec } x_0 = \langle x_2 \rangle \text{ and } x_1 = (\lambda x.x) (\lambda x'.x') \text{ and } x_2 = \{x_1\} \text{ in } x_0!1$	$\text{alloc}_{\text{value}}$
$\xrightarrow{\text{value}}$	$x_0 = \langle x_2 \rangle \text{ and } x_1 = (\lambda x.x) (\lambda x'.x') \text{ and } x_2 = \{x_1\} \vdash x_0!1$	β_{need}
$\xrightarrow{\text{value}}$	$x_0 = \langle x_2 \rangle \text{ and } x = \lambda x'.x' \text{ and } x_1 = x \text{ and } x_2 = \{x_1\} \vdash x_0!1$	$\text{assoc-env}_{\text{value}}$
$\xrightarrow{\text{value}}$	$x_0 = \langle x_2 \rangle \text{ and } x = \lambda x'.x' \text{ and } x_1 = \lambda x''.x'' \text{ and } x_2 = \{x_1\} \vdash x_0!1$	$\text{deref}_{\text{value}}$
$\xrightarrow{\text{value}}$	$x_0 = \langle x_2 \rangle \text{ and } x = \lambda x'.x' \text{ and } x_1 = \lambda x''.x'' \text{ and } x_2 = \{x_1\} \vdash \langle x_2 \rangle!1$	$\text{deref}_{\text{value}}$
$\xrightarrow{\text{value}}$	$x_0 = \langle x_2 \rangle \text{ and } x = \lambda x'.x' \text{ and } x_1 = \lambda x''.x'' \text{ and } x_2 = \{x_1\} \vdash x_1$	$\text{arr}_{\text{value}}$
$\xrightarrow{\text{value}}$	$x_0 = \langle x_2 \rangle \text{ and } x = \lambda x'.x' \text{ and } x_1 = \lambda x''.x'' \text{ and } x_2 = \{x_1\} \vdash \lambda x'''.x'''$	$\text{deref}_{\text{value}}$

Fig. 4. The reduction sequence for
 $\vdash \text{let rec } x_0 = \langle x_2 \rangle \text{ and } x_1 = (\lambda x.x) (\lambda x'.x') \text{ and } x_2 = \{x_1\} \text{ in } x_0!1$

(rules $\text{acc}_{\text{value}}$ and acc_{back}). The notation $x = a \in d$ denotes that the binding $x = a$ appears in d . Administrative rules $\text{assoc}_{\text{value}}$ and $\text{assoc-env}_{\text{value}}$ allocate bindings in the heap. We write $\xrightarrow{\text{value}}$ to denote the reflexive and transitive closure of $\xrightarrow{\text{value}}$, defining the reduction function.

Herein, we work with α -equivalence classes of terms and assume free variables and binding occurrences of variables in a representative use pairwise distinct names. In particular the reduction rules are applied by taking appropriate α -equivalent terms, preserving binding structure.

This evaluation strategy for *letrec* is motivated by PLT Scheme's implementation of *letrec* (Flatt *et al.*, 2009) and Scheme's semantics for *letrec** as standardized in (Sperber *et al.*, 2009) in that bindings in a *letrec* are evaluated from left to right and that bindings become available immediately after they are evaluated.

Reductions may terminate for the following reasons:

- A configuration reduces to a result, which is successful termination.
- Unsound initialization is encountered. This applies to configurations of the form D and $x = L[x']$ and $d \vdash a$ where either $x = x'$ or x' is bound in d , which attempt to dereference a variable that has not been evaluated.
- Ill-typed operation such as $\langle x \rangle a$ or $(\lambda x.a)!n$ is encountered.

The third possibility could be simply eliminated by a traditional type system. There have been proposals to eliminate the second possibility by more advanced type systems (Boudol, 2004; Dreyer, 2004). Many implementations of recursive records, such as OCaml's and Moscow ML's recursive modules, and F#'s and Java's classes and objects, eliminate the third possibility statically and cope with the second possibility at runtime by for instance signaling exceptions. The subject of the paper is to explore the design space of different evaluation strategies, but not to address unsound initialization.

Definition 3.1

The λ_{value} language is the set of terms equipped with the partial function $\xrightarrow{\text{value}}$.

As an example, Figure 4 presents the reduction sequence for $\vdash \text{let rec } x_0 = \langle x_2 \rangle \text{ and } x_1 = (\lambda x.x) (\lambda x'.x') \text{ and } x_2 = \{x_1\} \text{ in } x_0!1$.

$$\begin{aligned}
str : & \quad Tr_V(\{(X) f\})_\rho = \\
& \quad \text{let rec } x = \langle x' \rangle \text{ and } x' = TrFld_V(f; \varepsilon)_{\rho[X \mapsto x]} \text{ in } \langle x' \rangle \quad (x, x' \text{ fresh}) \\
mfld : & \quad TrFld_V(M = E; f; x, \dots)_\rho = \\
& \quad \text{let rec } x' = Tr_V(E)_\rho \text{ in } TrFld_V(f; x, \dots, x')_{\rho[M \mapsto x']} \quad (x' \text{ fresh}) \\
cfld : & \quad TrFld_V(c = e; f; x, \dots)_\rho = \\
& \quad \text{let rec } x = TrC_V(e)_\rho \text{ in } TrFld_V(f; x, \dots, x')_{\rho[c \mapsto x']} \quad (x' \text{ fresh}) \\
strbody : & \quad TrFld_V(\varepsilon; x, \dots)_\rho = \{x, \dots\} \\
cpath : & \quad TrC_V(p.n)_\rho = Tr_V(p)_\rho ! n \\
mpath : & \quad Tr_V(p.n)_\rho = Tr_V(p)_\rho ! n \\
mvar : & \quad Tr_V(X)_\rho = \rho(X) \\
funct : & \quad Tr_V(\Lambda X.E)_\rho = \lambda x. Tr_V(E)_{\rho[X \mapsto x]} \quad (x \text{ fresh}) \\
app : & \quad Tr_V(E_1(E_2))_\rho = Tr_V(E_1)_\rho Tr_V(E_2)_\rho \\
mname : & \quad Tr_V(M)_\rho = \rho(M) \\
cname : & \quad Tr_V(c)_\rho = \rho(c)
\end{aligned}$$
Fig. 5. Translation from *Osan* to λ_{value}

3.2 Translation

In Figure 5 we define translation from *Osan* to λ_{value} . The translation is very much inspired by an intermediate phase of the OCaml compiler (Leroy *et al.*, 2008), which translates a typed abstract syntax for modules into a lower-level lambda-like syntax. A metavariable ρ ranges over finite mappings from module and core names and module variables of *Osan* to variables of λ_{value} ; ρ is an empty mapping when translation begins. We assume given a translation function $TrC_V(e)_\rho$ for core expressions other than value paths and core names. The notation $\rho[X \mapsto x]$ denotes mapping extension of ρ with x at X . The notations $\rho[M \mapsto x]$ and $\rho[c \mapsto x]$ are similar.

Rule *str* translates a structure into a pointer $\langle x' \rangle$ to the variable x' , which is bound to the array corresponding to the structure's body. The self variable is bound to the pointer, too. That is, ρ is extended to map X to x , where x is bound to $\langle x' \rangle$. The introduction of indirection to reference an array is natural and reflects OCaml's implementation. It also makes it smooth to implement various evaluation strategies, as we will do in the subsequent sections.

Translation of the fields of a structure (*mfld*, *cfld* and *strbody*) elaborately takes account of backward references, which are treated differently from forward references. In other words, paths which go through self variables have different semantics from paths which do not. A path which refers to a preceding field but which goes through a self variable is deemed to be a forward reference. This implies we syntactically determine whether a path is a forward reference or not. For instance let us consider the following two *Osan* structures:

$$\{(X) \ c_1 = \text{fun } x \rightarrow x; \ c_2 = c_1 \ 3; \}$$

and

$$\{(X) \ c_1 = \text{fun } x \rightarrow x; \ c_2 = X.c_1 \ 3; \}$$

The former structure does not contain forward references, but the latter does, i.e., $X.c_1$. As a result evaluation of the former succeeds, while the latter fails in call-by-value. One might not think the different behavior between the above two structures intuitive. Yet the behavior

is faithful to that of both OCaml's and Moscow ML's recursive modules. Therefore we think it worth examining.

Then the fields of a structure is translated into a cascade of *letrec*'s so that the scope of a variable bound to a field extends to the subsequent fields. The second argument x, \dots of $TrFld_V(f; x, \dots)_\rho$ accumulates the fields translated so far, and is the result when f is empty (*strbody*). The dot notation of paths is translated into an accessor (*cpath* and *mpath*). Translation of variables, abstraction, application, module and core names is straightforward.

3.3 Discussion

The rest of this section examines the call-by-value strategy through examples. We assume a simple call-by-value functional core language in the examples. The core language contains, among others, a primitive *print e* which prints the result of evaluating e to visualize the evaluation order. We may omit declaring self variables when they are not used.

Most importantly the call-by-value strategy results in a simple evaluation order, which is important when a programming language allows arbitrary side effects. Borrowing from Syme's nomenclature (Syme, 2005), it allows delayed dependencies on forward references, but not immediate dependencies. In other words, a field in a structure can safely contain a forward reference as long as evaluation of the referring field need not evaluate the referred field. Typical cases are where the forward reference is under abstraction. For instance evaluation of the following *Osan* structure succeeds:

$$\{ M = \{(X) \\ c_1 = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else } X.c_2 (i - 1); \\ c_2 = \text{fun } i \rightarrow \text{if } i = 0 \text{ then false else } c_1 (i - 1); \}; \\ c = M.c_1 2; \}$$

But evaluation of the following structure fails:

$$\{(X) \ c_1 = X.c_2 + 3; \ c_2 = 2; \}$$

The latter structure is translated into:

$$\text{let rec } x = \langle x' \rangle \text{ and } x' = (\text{let rec } c_1 = x!2 + 3 \text{ in let rec } c_2 = 2 \text{ in } \{c_1, c_2\}) \text{ in } \langle x' \rangle$$

which reduces to:

$$x = \langle x' \rangle \text{ and } c_1 = \langle x' \rangle!2 + 3 \text{ and } x' = (\text{let rec } c_2 = 2 \text{ in } x' = \{c_1, c_2\}) \vdash \langle x' \rangle$$

and unsound initialization is encountered.

The former structure is translated into:

$$\text{let rec } x_1 = \langle x'_1 \rangle \\ \text{and } x'_1 = \\ \text{let rec } m = \\ \text{let rec } x = \langle x' \rangle \text{ and } x' = (\text{let rec } c_1 = \text{body}_{c_1} \text{ in let rec } c_2 = \text{body}_{c_2} \text{ in } \{c_1, c_2\}) \text{ in} \\ \langle x' \rangle \text{ in} \\ \text{let rec } c = (m!1) 2 \text{ in} \\ \{m, c\} \text{ in} \\ \langle x'_1 \rangle$$

where body_{c_1} and body_{c_2} respectively abbreviate $\text{fun } i \rightarrow \text{if } i = 0 \text{ then true else } (x!2) (i - 1)$ and $\text{fun } i \rightarrow \text{if } i = 0 \text{ then false else } c_1 (i - 1)$. The above expression successfully reduces

to a result:

$$\begin{aligned} &x_1 = \langle x'_1 \rangle \text{ and } x = \langle x' \rangle \text{ and } c_1 = \text{body}_{c_1} \text{ and } c_2 = \text{body}_{c_2} \\ &\text{and } x' = \{c_1, c_2\} \text{ and } m = \langle x' \rangle \text{ and } c = \text{true} \text{ and } x'_1 = \{m, c\} \\ &\vdash \langle x'_1 \rangle \end{aligned}$$

It should be noticed that the last core field c cannot be included in M :

$$\begin{aligned} \{ &M = \{(X) \\ &c_1 = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else } X.c_2(x-1); \\ &c_2 = \text{fun } x \rightarrow \text{if } i = 0 \text{ then false else } c_1(x-1); \\ &c = c_1\ 2; \}; \} \end{aligned}$$

Then evaluation of c forces evaluation of $X.c_2$. Since the structure bound to X is currently under evaluation, however, none of its fields can be accessed yet.

The call-by-value strategy entails some limitations on the possible recursion between modules. Notably it does not permit taking fix-points of functors unless arguments are expanded into module-level immediate values by applying eta-expansions. This limitation, as well as the solution with eta-expansions, have been discussed by Russo (Russo, 2001) on a typical example of recursive modules. As a simplified example from his paper, call-by-value fails to evaluate the following *Osan* structure:

$$\begin{aligned} \{ &F = \Lambda Y. \{ g = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } Y.g(i-1); \}; \\ &M = \{(X) \ M' = F(X.M'); \}; \} \end{aligned}$$

We need to expand the argument:

$$\begin{aligned} \{ &F = \Lambda Y. \{ g = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } Y.g(i-1); \}; \\ &M = \{(X) \ M' = F(\{ g = \text{fun } i \rightarrow X.M'.g\ i; \}); \}; \} \end{aligned}$$

Such expansions are neither natural nor always possible in particular in the presence of type abstraction, e.g., F 's signature may hide the concrete type of g , so that $\text{fun } i \rightarrow X.M'.g\ i$ does not type check.

The ability to take fix-points of functors is a crucial motivation for introducing recursive modules; it allows separate and extensible development of recursively defined modules (Crary *et al.*, 1999; Russo, 2001; Nakata & Garrigue, 2006). Thus the call-by-value strategy enjoys the simple evaluation order at the cost of flexibility in handling recursion between modules.

If functors are lazy, then we can take fix-points without resorting to expansions. In the subsequent sections we will examine such call-by-need evaluation strategies.

4 Call-by-need modules

In this section we examine a call-by-need strategy. Below are two of its key features, which contrast with the call-by-value strategy of the previous section.

1. A functor argument is evaluated lazily when the argument is used.
2. Evaluation of a module is delayed until the module is accessed for the first time. In particular, accessing a field of a structure does not trigger evaluation of a submodule it contains, unless evaluation of a core field of the structure requires the submodule by accessing it.

Apart from lazy submodules, evaluation of a structure proceeds in a similar way to the call-by-value strategy, which means when a structure is accessed for the first time, all its

<i>Expressions</i>	a	$::=$	$x \mid \lambda x.a \mid a_1 a_2 \mid (a, \dots) \mid a.n \mid \langle x \rangle \mid \{r, \dots\} \mid a!n$ $\mid \text{let rec } d \text{ in } a$
<i>References</i>	r	$::=$	$x \mid \lambda_{..}x$
<i>Dereferences</i>	$\#x$	$::=$	$x \mid \langle x \rangle!n$
<i>Values</i>	v	$::=$	$\lambda x.a \mid (v, \dots) \mid \langle x \rangle \mid \{r, \dots\}$
<i>Definitions</i>	d	$::=$	$x = a \text{ and } \dots$
<i>Lift contexts</i>	L	$::=$	$\square \mid a \mid (\dots, v, \square, a, \dots) \mid \square.n \mid \square!n$
<i>Nested lift contexts</i>	N	$::=$	$\square \mid L[N]$
<i>Lazy evaluation contexts</i>	K	$::=$	$d \vdash N \mid x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\#x]$
<i>Dependencies</i>	$d[x, x']$	$::=$	$x = N[\#x'] \mid d[x, x''] \text{ and } x'' = N[\#x']$

Fig. 6. Syntax for λ_{need}

core fields, not including those of submodules, are evaluated in the top-to-bottom order at once.

The first feature is typical to call-by-need. We have introduced lazy submodules to give a simpler semantics to the following *Osan* structure:

$$\{ F = \Lambda X. \{ c = \text{print } \text{"bye"}; \}; M = F(\{ c = \text{print } \text{"hello"}; \}); \}$$

If we introduce lazy functors but evaluate submodules eagerly, then what should evaluation of the above structure print? Probably it would only print “bye”. The submodule M is evaluated eagerly, thus the instantiation of F is evaluated eagerly and “bye” is printed. Since F evaluates arguments lazily and its body does not use the arguments, however, “hello” would not be printed. Now consider the following variation:

$$\{ F = \Lambda X. \{ c = \text{print } \text{"bye"}; \}; N = \{ c = \text{print } \text{"hello"}; \}; N' = F(N); \}$$

Evaluating submodules eagerly, the above structure would print “hello bye”. The different behavior between the above two structures does not look intuitive.

If we make both functors and submodules lazy, then the above two structures do not print anything; it’s up to the programmer’s control. That is, the programmer has to explicitly force evaluation by accessing a field of a module. For instance evaluation of the following structure:

$$\{ F = \Lambda X. \{ c = \text{print } \text{"bye"}; \}; M = F(\{ c = \text{print } \text{"hello"}; \}); c = M.c; \}$$

prints “bye”. Evaluation of the following structure:

$$\{ F = \Lambda X. \{ c_1 = X.c; c_2 = \text{print } \text{"bye"}; \}; M = F(\{ c = \text{print } \text{"hello"}; \}); c = M.c_2; \}$$

prints “hello bye”.

Note that lazy submodules mean not only that structures are evaluated lazily, but also that abbreviations are evaluated lazily. That is, the following program prints nothing:

$$\{ M = \{ c = \text{print } \text{"hi"}; \}; N = M; \}$$

Arguably lazy functors with lazy submodules result in a more intuitive semantics than lazy functors with eager submodules.

4.1 The target language λ_{need} for call-by-need modules

In Figure 6 we define the syntax of the target language λ_{need} for the call-by-need strategy. It includes eager tuples as well as array primitives. We will use eager tuples to implement initializers for arrays, as explained below. In λ_{need} no ordering among bindings in a letrec or

Lazy modules

11

Contraction rules

$$\begin{array}{lcl}
\beta_{need} : & (\lambda x.a) a' & \xrightarrow[\text{need}]{} \text{let rec } x = a' \text{ in } a \\
prj : & (\dots, v_n, \dots).n & \xrightarrow[\text{need}]{} v_n \\
lift : & L[\text{let rec } d \text{ in } a] & \xrightarrow[\text{need}]{} \text{let rec } d \text{ in } L[a]
\end{array}$$

Reduction rules

$$\begin{array}{lcl}
cxt : & K[a] & \xrightarrow[\text{need}]{} K[a'] \text{ if } a \xrightarrow[\text{need}]{} a' \\
deref : & K[x] & \xrightarrow[\text{need}]{} K[v] \text{ if } x = v \in K \\
arr_{need} : & K[\langle x \rangle !n] & \xrightarrow[\text{need}]{} K[(r, \dots).n] \text{ if } x = \{r, \dots\} \in K \\
alloc : & d \vdash \text{let rec } d' \text{ in } a & \xrightarrow[\text{need}]{} d \text{ and } d' \vdash a \\
alloc-env : & x' = (\text{let rec } d \text{ in } a) \text{ and } d^*[x, x'] \text{ and } d' \vdash N[\sharp x] & \\
& \xrightarrow[\text{need}]{} d \text{ and } x' = a \text{ and } d^*[x, x'] \text{ and } d' \vdash N[\sharp x] &
\end{array}$$

Access in evaluation contexts

$$\begin{array}{lcl}
acc : & x = a \in d \vdash N & \text{if } x = a \in d \\
acc-env : & x = a \in x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x] & \text{if } x = a \in d
\end{array}$$

Fig. 7. Reduction rules for λ_{need}

$$\begin{array}{lcl}
& \vdash \text{let rec } x = (\lambda y.\lambda y'.y) x \text{ in } x (\lambda x'.x') & \\
\begin{array}{l} \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \\ \xrightarrow[\text{need}]{} \end{array} & x = (\lambda y.\lambda y'.y) x \vdash x (\lambda x'.x') & \text{by } alloc \\
& x = (\text{let rec } y = x \text{ in } \lambda y'.y) \vdash x (\lambda x'.x') & \text{by } \beta_{need} \\
& y = x \text{ and } x = \lambda y'.y \vdash x (\lambda x'.x') & \text{by } alloc-env \\
& y = x \text{ and } x = \lambda y'.y \vdash (\lambda y_1.y) (\lambda x'.x') & \text{by } deref \\
& y = x \text{ and } x = \lambda y'.y \vdash \text{let rec } y_1 = \lambda x'.x' \text{ in } y & \text{by } \beta_{need} \\
& y = x \text{ and } x = \lambda y'.y \text{ and } y_1 = \lambda x'.x' \vdash y & \text{by } alloc \\
& y = \lambda y_2.y \text{ and } x = \lambda y'.y \text{ and } y_1 = \lambda x'.x' \vdash y & \text{by } deref \\
& y = \lambda y_2.y \text{ and } x = \lambda y'.y \text{ and } y_1 = \lambda x'.x' \vdash \lambda y_3.y & \text{by } deref
\end{array}$$

Fig. 8. The reduction sequence of $\vdash \text{let rec } x = (\lambda y.\lambda y'.y) x \text{ in } x (\lambda x'.x')$

a heap is assumed. That is, we identify $\text{let rec } d \text{ in } a$ as well as $d \vdash a$ up to re-ordering of the bindings in d . The notation $\lambda_{\dots} x$ denotes $\lambda x'.x$ where x' does not bind x . Here an element of an array may be a *delayed variable* $\lambda_{\dots} x$ or a variable x . *Answers* are configurations $d \vdash v$ consisting of bindings and a value; answers may contain non-value expressions in the heaps. Dependencies $d[x, x']$ stands for a non-empty set of bindings of the form “ $x = N_1[\sharp x_1]$ and $x_1 = N_2[\sharp x_2]$ and \dots and $x_{n-1} = N_n[\sharp x']$ ”, where $\sharp x$ denotes either x or $\langle x \rangle !n$. We write $d^*[x, x']$ to denote either $d[x, x']$, or an empty sequence with the side-condition $x = x'$. λ_{need} evaluates bindings in a heap lazily: a binding is evaluated when the bound variable is used. Production $x' = N$ and $d^*[x, x']$ and $d \vdash N'[\sharp x]$ of evaluation contexts forces evaluation of a binding on request.

Figure 7 presents the reduction rules. Now a binding in a heap is available if it is not under evaluation (rules *acc* and *acc-env*). Rule *alloc-env* allocates bindings in the heap only when needed. In respect of *Osan*, the call-by-need strategy evaluates all the core field of a structure when the structure is accessed for the first time. This effect is implemented by rule *arr_{need}*. Accessing the n -th field of an array referenced by x , or $\langle x \rangle !n$, reduces to $(r, \dots).n$,

$$\begin{aligned}
str &: Tr_N(\{(X) f\})_\rho = \\
&\quad \text{let rec } x = \langle x' \rangle \text{ and } x' = TrFld_N(f; \varepsilon)_\rho[x \mapsto x] \text{ in } \langle x' \rangle \quad (x, x' \text{ fresh}) \\
mfld &: TrFld_N(M = E; f; r, \dots)_\rho = \\
&\quad \text{let rec } x = Tr_N(E)_\rho \text{ in } TrFld_N(f; r, \dots, \lambda_{\dots} x)_\rho[M \mapsto x] \quad (x \text{ fresh}) \\
cfld &: TrFld_N(c = e; f; r, \dots)_\rho = \\
&\quad \text{let rec } x = TrC_N(e)_\rho \text{ in } TrFld_N(f; r, \dots, x)_\rho[c \mapsto x] \quad (x \text{ fresh}) \\
strbody &: TrFld_N(\varepsilon; r, \dots)_\rho = \{r, \dots\} \\
vpath &: TrC_N(p.n)_\rho = Tr_N(p)_\rho ! n \\
mpath &: Tr_N(p.n)_\rho = (Tr_N(p)_\rho ! n) I \\
mvar &: Tr_N(X)_\rho = \rho(X) \\
funct &: Tr_N(\Lambda X.E)_\rho = \lambda x. Tr_N(E)_\rho[x \mapsto x] \quad (x \text{ fresh}) \\
app &: Tr_N(E_1(E_2))_\rho = Tr_N(E_1)_\rho Tr_N(E_2)_\rho \\
mname &: Tr_N(M)_\rho = \rho(M) \\
cname &: Tr_N(c)_\rho = \rho(c)
\end{aligned}$$
Fig. 9. Translation from *Osan* to λ_{need} , λ_{modest} and λ_{full}

therefore forcing evaluation of all the elements of the array. This way of implementing an initializer in λ_{need} is inspired by an alternative implementation, written in OCaml syntax, we have in mind:

```

let get (a : ('a Lazy.t) array) n =
  for i = 0 to Array.length a - 1 do Lazy.force a.(i) done;
  Lazy.force a.(n)

```

As defined in the next subsection, submodules are translated into delayed variables $\lambda_{\dots} x$, so that the initializer (r, \dots) does not force evaluation of submodules. We write $\xrightarrow[\text{need}]{} \rightarrow$ to denote the reflexive and transitive closure of $\xrightarrow{\text{need}}$.

As an example, Figure 8 presents the reduction sequence of $\vdash \text{let rec } x = (\lambda y. \lambda y'. y) x \text{ in } x (\lambda x'. x')$ in λ_{need} . Note that λ_{value} does not reduce the expression to a result.

Definition 4.1

The λ_{need} language is the set of terms equipped with the partial function $\xrightarrow[\text{need}]{} \rightarrow$.

The reduction semantics of λ_{need} differs from the standard reduction semantics of Ariola and Felleisen's call-by-need cyclic lambda calculus (Ariola & Felleisen, 1997) in the following two points. Firstly λ_{need} allocates the bindings of a letrec in the heap before evaluating the body of the letrec. Secondly λ_{need} discards nested structure of heap-allocated letrec's by rule *alloc*, which is absent from Ariola and Felleisen's. We have departed from Ariola and Felleisen's formulation to accommodate an extension with state in the style of Felleisen and Hieb (Felleisen & Hieb, 1992). Additionally, our formulation is substantially easier than Ariola and Felleisen's to prove correct with respect to a corresponding natural semantics (Nakata & Hasegawa, 2009).

We underline the absence of back-patching from λ_{value} and λ_{need} , thus confirming the previous observations by Felleisen and Hieb (Felleisen & Hieb, 1992) and Ariola and Felleisen (Ariola & Felleisen, 1997) that call-by-value and call-by-need operational semantics for arbitrary recursive bindings may have functional specifications.

4.2 Translation

In Figure 9 we define translation from *Osan* to λ_{need} . The notation I abbreviates $\lambda x.x$. Translation for λ_{need} differs from that for λ_{value} in that module fields are translated into delayed variables (rule *mfld*), and accordingly translation of a module path (rule *mpath*) introduces one application to force the evaluation of the delayed variable. Otherwise the translation is unchanged.

As an example let us consider the following *Osan* program:

$$\{ M = \{ c_1 = \text{print } \text{"good"}; c_2 = \text{print } \text{"bye"}; \}; c_1 = \text{print } \text{"hello"}; c_2 = M.c_1; \}$$

Hereafter we assume a convention that a program is a structure and evaluation of a program returns the result of evaluating the last field of the structure. Then the above program is translated into:

```
let rec x = ⟨x'⟩
and x' =
  let rec m =
    let rec x1 = ⟨x'1⟩
    and x'1 = (let rec c'1 = print "good" in let rec c'2 = print "bye" in {c'1, c'2}) in
    ⟨x'1⟩ in
  let rec c1 = print "hello" in
  let rec c2 = m!1 in
  {λ..m, c1, c2}} in
x!3
```

where the top-level structure is translated specially, i.e., the body is $x!3$, in accordance with the convention. The reduction first allocates the toplevel letrec in the heap and reduces the body into $\langle x' \rangle!3$. Then with administrative steps it reduces into:

```
x = ⟨x'⟩
and m =
  let rec x1 = ⟨x'1⟩
  and x'1 = (let rec c'1 = print "good" in let rec c'2 = print "bye" in {c'1, c'2}) in
  ⟨x'1⟩
and c1 = print "hello" and c2 = m!1 and x' = {λ..m, c1, c2}}
⊢ ⟨x'⟩!3
```

Now rule *arr_{need}* is applicable:

```
x = ⟨x'⟩
and m =
  let rec x1 = ⟨x'1⟩
  and x'1 = (let rec c'1 = print "good" in let rec c'2 = print "bye" in {c'1, c'2}) in
  ⟨x'1⟩
and c1 = print "hello" and c2 = m!1 and x' = {λ..m, c1, c2}}
⊢ (λ..m, c1, c2}).3
```

Then the next reduction focuses on $(\lambda_{..m, c_1, c_2})$, and "hello" is first printed by forcing evaluation of c_1 , then c_2 is forced to evaluate. After an administrative step *alloc-env* fol-

lowed by *deref* we obtain:

$$\begin{aligned} x &= \langle x' \rangle \text{ and } x_1 = \langle x'_1 \rangle \\ \text{and } x'_1 &= (\text{let rec } c'_1 = \text{print "good" in let rec } c'_2 = \text{print "bye" in } \{c'_1, c'_2\}) \\ \text{and } m &= \langle x'_1 \rangle \text{ and } c_1 = () \text{ and } c_2 = \langle x'_1 \rangle!1 \text{ and } x' = \{\lambda _ . m, c_1, c_2\} \\ &\vdash (\lambda _ . m, (), c_2).3 \end{aligned}$$

After two administrative steps *alloc-env* followed by *arr_{need}* we obtain:

$$\begin{aligned} x &= \langle x' \rangle \text{ and } x_1 = \langle x'_1 \rangle \text{ and } c'_1 = \text{print "good" and } c'_2 = \text{print "bye" and } x'_1 = \{c'_1, c'_2\} \\ \text{and } m &= \langle x'_1 \rangle \text{ and } c_1 = () \text{ and } c_2 = (c'_1, c'_2).1 \text{ and } x' = \{\lambda _ . m, c_1, c_2\} \text{ in} \\ &(\lambda _ . m, (), c_2).3 \end{aligned}$$

Then the next reduction focuses on the rhs of c_2 and “good bye” is printed by evaluating c'_1 then c'_2 in order. The reductions terminate with an answer:

$$\begin{aligned} x &= \langle x' \rangle \text{ and } x_1 = \langle x'_1 \rangle \text{ and } c'_1 = () \text{ and } c'_2 = () \text{ and } x'_1 = \{c'_1, c'_2\} \\ \text{and } m &= \langle x'_1 \rangle \text{ and } c_1 = () \text{ and } c_2 = () \text{ and } x' = \{\lambda _ . m, c_1, c_2\} \\ &\vdash () \end{aligned}$$

Proposition 4.1 states the call-by-need strategy is more successful than the call-by-value strategy. This is what one would expect. We prove the proposition by going through natural semantics. We define a natural semantics for λ_{value} (resp. λ_{need}), in the style of Launchbury (Launchbury, 1993) and Sestoft (Sestoft, 1997), which evaluates an expression to a value if and only if the corresponding reduction semantics reduces the same expression to a result (resp. answer). We prove the two versions of the translation $Tr_V(\cdot)$ and $Tr_N(\cdot)$ produce semantically equivalent expressions with respect to the natural semantics for λ_{value} . Then we prove if the natural semantics for λ_{value} evaluates an expression to a value, then so does the natural semantics for λ_{need} . Thus we obtain an equivalent result to the proposition in terms of the natural semantics, and the proposition is ascribed to the correspondence between the reduction and the natural semantics. The proof is found in Appendix.

Proposition 4.1

$$\text{If } \vdash Tr_V(E)_\emptyset \xrightarrow[\text{value}]{} D \vdash v \text{ then } \vdash Tr_N(E)_\emptyset \xrightarrow[\text{need}]{} d \vdash v'.$$

4.3 Discussion

The call-by-need strategy permits taking fix-points of functors without expanding arguments. Hence it successfully evaluates the following program from the previous section:

$$\begin{aligned} \{ F &= \Lambda Y. \{ g = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } Y.g (i - 1); \}; \\ M &= \{ (X) M' = F(X.M'); \}; \\ c &= M.M'.g 2; \} \end{aligned}$$

where the last field c is added to force evaluation of module $M.M'$. One might be tempted to take the fix-point directly:

$$\begin{aligned} \{ (X) \\ F &= \Lambda Y. \{ g = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } Y.g (i - 1); \}; \\ M &= F(X.M); \\ c &= M.g 2; \} \end{aligned}$$

This attempt fails, however, since evaluation of core field c forces evaluation of $X.M$ while

the structure bound to X is still under evaluation. Although this is not an essential limitation, an extra layer of nesting to take fix-points is a possibly cumbersome consequence of supporting forward references through self variables; we discuss this issue later in Section 4.3.1, by considering the module `rec` construct as provided in OCaml.

The programmer may exploit lazy submodules to tame the restriction on immediate dependencies on forward references. For instance the following *Osan* program successfully evaluates in the call-by-need strategy:

$$\{ M = \{(X) M_1 = \{ c = (\text{fun } i \rightarrow i + 1) X.M_2.c; \}; M_2 = \{ c = 0; \}; \}; \\ c = M.M_1.c; \}$$

whereas the following program fails to evaluate:

$$\{ M = \{(X) c = (\text{fun } i \rightarrow i + 1) X.M_2.c; M_2 = \{ c = 0; \}; \}; \\ c = M.c; \}$$

The top-to-bottom evaluation order in a structure requires preceding core fields of the same and enclosing structures to be evaluated first. The enclosing structure may contain other submodules; lazy submodules imply that core fields of those submodules need not be evaluated first. According to these criteria, the latter program fails since the field $M.c$ is a preceding field of the enclosing structure in respect of field $M_2.c$.

One potential source of limitations of the call-by-need strategy is that a field of a structure becomes accessible only after all the core fields of the structure have been evaluated, even though the accessed field has been evaluated. As a result the call-by-need strategy does not permit the following program:

$$\{ F = \\ \quad \lambda Y. \{ g = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } Y.g (i - 1); c = g 2; \}; \\ M = \{(X) M' = F(X.M'); \}; \\ c = M.M'.c; \}$$

The problem here is that evaluating the body of functor F accesses the argument, since evaluation of its core field c calls g , which in turn calls $Y.g$. Hence, by taking a fix-point of F evaluation of $M.M'$ calls $M.M'.g$ via the functor parameter, which happens to be bound to $M.M'$ itself. Since $M.M'$ is still under evaluation, none of its fields is accessible.

The next section proposes a strategy which permits the above program, by making fields of a structure accessible immediately after they are evaluated.

4.3.1 The module `rec` construct

The idea of extending structures with self variables has been used in Moscow ML (Russo, 2001). Unlike Moscow ML, OCaml supports recursive modules via the module `rec` construct, which is a module-level counterpart to `let rec` (Leroy, 2003); modules simultaneously defined with a module `rec` can refer to each other recursively. Our experience in programming with recursive modules is mainly in OCaml and we found module `rec` handy as a programmer. For instance we can take fix-points of functors directly without going

$$\begin{array}{ll}
\text{Expressions} & a ::= x \mid \lambda x.a \mid a_1 a_2 \mid (a, \dots) \mid a.n \mid \langle x \rangle \mid \{r, \dots\} \mid \{\{r, \dots\}\} \mid a!n \\
& \quad \mid \text{let rec } d \text{ in } a \\
\text{Values} & v ::= \lambda x.a \mid (v, \dots) \mid \langle x \rangle \mid \{r, \dots\} \mid \{\{r, \dots\}\}
\end{array}$$
Fig. 10. Syntax for λ_{lazy}

$$\begin{array}{ll}
\text{init} : & x = a \text{ and } d \vdash N[\langle x \rangle!n] \xrightarrow{\text{lazy}} x = a' \text{ and } d \vdash N[(r, \dots).n] \\
& \text{where } a = \{\{r, \dots\}\} \text{ and } a' = \{r, \dots\} \\
\text{init-env} : & x'' = a \text{ and } x' = N[\langle x'' \rangle!n] \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x] \\
& \xrightarrow{\text{lazy}} x'' = a' \text{ and } x' = N[(r, \dots).n] \text{ and } d[x, x'] \text{ and } d \vdash N'[\sharp x] \\
& \text{where } a = \{\{r, \dots\}\} \text{ and } a' = \{r, \dots\} \\
\text{arr}_{\text{lazy}} : & K[\langle x \rangle!n] \xrightarrow{\text{lazy}} K[(r_1, \dots, r_n).n] \text{ if } x = \{r_1, \dots, r_n, r_{n+1} \dots\} \in K
\end{array}$$
Fig. 11. Reduction rules for λ_{lazy}

through self variables²:

module rec $M = F(M)$

Whether to introduce self variables or module rec is a matter of design choices at the surface syntax level. There is no technical difficulty in introducing module rec in the syntax of *Osan*. A straightforward adaptation of rule *mfd* in the translation is sufficient. In this paper we preferred self variables in favor of simplicity and generality, e.g., module rec does not permit mixing module and core definitions.

5 Lazy-field modules

In this section we propose a *lazy-field* strategy, which differs from the call-by-need strategy of the previous section in that fields of a structure become accessible immediately after they are evaluated, while all the core fields of the structure might not have been evaluated yet.

5.1 The target language λ_{lazy} for lazy-field modules

A subtle point of the lazy-field strategy is that we need to distinguish the first access to a structure from the second and later accesses, so that we evaluate all the core fields of a structure when the structure is accessed for the first time. We implement this effect by adopting a rather simple machinery from Java's lazy class initialization, which prevents recursive initialization of the same class (Gosling *et al.*, 2005): we mark arrays to indicate if they have been accessed or not. Therefore we extend the syntax of λ_{need} with *frozen arrays* $\{\{r, \dots\}\}$, as given in Figure 10. Frozen arrays represent arrays which have not been accessed, whereas (ordinary) arrays represent arrays which have been accessed.

As to the reduction semantics we replace rule *arr_{need}* of Figure 7 with the three rules given in Figure 11. Rules *init* and *init-env* handle the case where an array is accessed for the first time. In this case all the elements of the array are forced to evaluate and the

² Interestingly OCaml permits taking fix-points without expanding arguments, while functors are call-by-value. This point will be discussed in Related work.

$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = x!1$ and $x' = \{\{x_1, x_2\}\}$ in $x!1$		
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = x!1$ and $x' = \{\{x_1, x_2\}\} \vdash x!1$		by <i>alloc</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = x!1$ and $x' = \{\{x_1, x_2\}\} \vdash \langle x' \rangle!1$		by <i>deref</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = x!1$ and $x' = \{x_1, x_2\} \vdash (x_1, x_2).1$		by <i>init</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = x!1$ and $x' = \{x_1, x_2\} \vdash (1, x_2).1$		by <i>deref</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = \langle x' \rangle!1$ and $x' = \{x_1, x_2\} \vdash (1, x_2).1$		by <i>deref</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = (x_1).1$ and $x' = \{x_1, x_2\} \vdash (1, x_2).1$		by <i>arr_{need}</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = (1).1$ and $x' = \{x_1, x_2\} \vdash (1, x_2).1$		by <i>deref</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = 1$ and $x' = \{x_1, x_2\} \vdash (1, x_2).1$		by <i>prj</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = 1$ and $x' = \{x_1, x_2\} \vdash (1, 1).1$		by <i>deref</i>
$\xrightarrow{\text{lazy}}$	$x = \langle x' \rangle$ and $x_1 = 1$ and $x_2 = 1$ and $x' = \{x_1, x_2\} \vdash 1$		by <i>prj</i>

Fig. 12. The reduction sequence of $\vdash \text{let rec } x = \langle x' \rangle \text{ and } x_1 = 1 \text{ and } x_2 = x!1 \text{ and } x' = \{\{x_1, x_2\}\} \text{ in } x!1$

$$\text{strbody} : \text{TrFld}_L(\varepsilon; r, \dots)_\rho = \{\{r, \dots\}\}$$

Fig. 13. Translation from *Osan* to λ_{lazy}

array is unmarked, namely the frozen array is updated by the corresponding ordinary array. Rule *arr_{lazy}* handles the case where an array is accessed for the second time or later. In this case only the preceding elements to the accessed element are forced to be evaluated. If a preceding element is still under evaluation, which means evaluation of an element of the array attempts to access a subsequent element of the same array, then unsound initialization is encountered. We write $\xrightarrow{\text{lazy}}$ to denote the reflexive and transitive closure of $\xrightarrow{\text{lazy}}$.

As an example, Figure 12 presents the reduction sequence of $\vdash \text{let rec } x = \langle x' \rangle \text{ and } x_1 = 1 \text{ and } x_2 = x!1 \text{ and } x' = \{\{x_1, x_2\}\} \text{ in } x!1$. A similar term obtained by replacing $\{\{x_1, x_2\}\}$ with $\{x_1, x_2\}$ does not reduce to an answer in λ_{need} .

Definition 5.1

The λ_{lazy} language is the set of terms equipped with the partial function $\xrightarrow{\text{lazy}}$.

5.2 Translation

Translation $\text{Tr}_L(\cdot)$. from *Osan* to λ_{lazy} is obtained by translating structures into frozen arrays, rather than into ordinary arrays. Therefore we replace rule *strbody* of Figure 9 by the rule given in Figure 13. The other rules are unchanged.

We demonstrate the translation and the reductions of the resulting expression of the following program:

$$\{(X) \ M = X.M'; \ M' = \{ c_1 = 3; \ c_2 = X.M.c_1 + 3; \}; \ c = M.c_2; \}$$

The core field $M'.c_2$ refers to the preceding field $M'.c_1$ via the forward referencing abbreviation $X.M'$. Evaluation of the above program should succeed according to our informal explanation of the lazy-field strategy: fields of a structure are accessible immediately after they are evaluated. Indeed, $M'.c_1$ has been evaluated when evaluating $M'.c_2$. The above

program is translated into:

```

let rec x = ⟨x'⟩
and x' =
  let rec m = (x!2) I in
  let rec m' =
    let rec x1 = ⟨x'1⟩ and x'1 = (let rec c1 = 3 in c2 = ((x!1) I)!1 + 3 in {{c1, c2}}) in ⟨x'1⟩ in
    let rec c = m!2 in
    {{λ-.m, λ-.m', c}} in
  x!3

```

After some work we obtain:

```

x = ⟨x'⟩ and m = ((x')!2) I
and m' =
  let rec x1 = ⟨x'1⟩ and x'1 = (let rec c1 = 3 in c2 = ((x!1) I)!1 + 3 in {{c1, c2}}) in ⟨x'1⟩
and c = m!2 and x' = {λ-.m, λ-.m', c}
⊢ (λ-.m, λ-.m', c).3

```

In λ_{need} the above term would encounter unsound initialization since $m = ((x')!2) I$ reduces to $m = ((\lambda_{-}.m, \lambda_{-}.m', c).2) I$ where c is under evaluation. In λ_{lazy} the above initialization pattern is permitted since $m = ((x')!2) I$ reduces to $m = ((\lambda_{-}.m, \lambda_{-}.m').2) I$, which further reduces to $m = \text{let rec } _ = I \text{ in } m'$. Hereafter we omit introducing anonymous bindings $\text{let rec } _ = I \text{ in}$ in examples for conciseness.

Continuing the reductions we obtain:

```

x = ⟨x'⟩ and m = ⟨x'1⟩ and x1 = ⟨x'1⟩ and c1 = 3 and c2 = ((x!1) I)!1 + 3
and x'1 = {c1, c2} and m' = ⟨x'1⟩ and c = (3, c2).2 and x' = {λ-.m, λ-.m', c}
⊢ (λ-.m, λ-.m', c).3

```

Here notice that both m and m' , which respectively correspond to modules M and M' , are bound to pointer $\langle x'_1 \rangle$ to the array representing M' 's structure body. The next reduction focuses on the rhs of c_2 , which reduces as follows:

$$((x!1) I)!1 + 3 \rightarrow ((\langle x' \rangle!1) I)!1 + 3 \rightarrow ((\lambda_{-}.m).1) I)!1 + 3 \rightarrow ((\lambda_{-}.m) I)!1 + 3$$

$$\rightarrow m!1 + 3 \rightarrow \langle x'_1 \rangle!1 + 3 \rightarrow (c_1).1 + 3 \rightarrow (3).1 + 3 \rightarrow 3 + 3 \rightarrow 6$$

Finally the reductions terminate with an answer:

```

x = ⟨x'⟩ and m = ⟨x'1⟩ and x1 = ⟨x'1⟩ and c1 = 3 and c2 = 6
and x'1 = {c1, c2} and m' = ⟨x'1⟩ and c = 6 and x' = {λ-.m, λ-.m', c}
⊢ 6

```

Proposition 5.1 states the lazy-field strategy is more successful than the call-by-need strategy. Again it is proved by going through natural semantics. The proof is found in Appendix.

Proposition 5.1

If $\vdash Tr_N(E)_\emptyset \xrightarrow{\text{need}} d \vdash v$ then $\vdash Tr_L(E)_\emptyset \xrightarrow{\text{lazy}} d' \vdash v'$.

Given that our proofs always go through natural semantics, one may wonder why we do not present the natural semantics upfront and define the operational semantics of the target languages by the natural semantics. On the one hand, we prefer to use reduction semantics to reason about behaviour of different evaluation strategies. For instance we are able to compare them with respect to when one strategy encounters unsound initialization, which is permitted by another strategy, by locally rewriting terms step-by-step. Our choice

of the reduction semantics shall be also supported by PLT Redex (Felleisen *et al.*, 2009), a tool with which one can interactively experiment with reduction semantics. On the other hand, we found proofs are easier with natural semantics. Therefore we choose to enjoy advantages of both the styles of formalization of operational semantics.

5.3 Discussion

In the lazy-field strategy whether a path is a forward reference or not is determined based on the actual semantics, irrespective of whether it goes through a self variable. This is in contrast to the call-by-value and call-by-need strategies. For instance the lazy-field strategy successfully evaluates the following program, which neither the call-by-value nor the call-by-need strategy can handle:

$$\{(X) \ c_1 = \text{fun } x \rightarrow x; \ c_2 = X.c_1 \ 3; \}$$

The lazy-field strategy permits the last example in the previous section, which the call-by-need strategy does not permit. Moreover we can take fix-points directly without an extra nesting, as follows:

$$\{(X) \ F = \Lambda Y. \{ g = \text{fun } i \rightarrow \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } Y.g \ (i - 1); \ c = g \ 2; \}; \\ M = F(X.M); \\ c = M.c; \}$$

One potential disadvantage of the lazy-field strategy is its asymmetry. Namely, whether evaluating mutually recursive modules succeeds may rely on which module is forced to evaluate first. For instance in the following program:

$$\{(X) \ M = \{ c_1 = 1; \ c_2 = X.N.c_2 \}; \ N = \{ c_1 = M.c_1; \ c_2 = 2; \}; \}$$

if M is forced first then the evaluation is successful, but if N is forced first then the evaluation fails due to unsound initialization. This asymmetry has been already predictable from rules *init* and *init-env*, which perform semantic non-preserving update. The asymmetry is not necessarily unacceptable: in Java if the programmer observes implicit default values such as null-pointers may depend on the order of class initialization. Recall that classes are initialized lazily in Java.

6 Modest-fields modules

The lazy-field strategy does not allow evaluation to alternate between modules, yet follows stack-like discipline in the sense that when evaluation of a module triggers evaluation of another module, then the triggered evaluation must be completed before the triggering evaluation is resumed. As a result, the lazy-field strategy does not permit the following program:

$$\{(X) \ M_1 = \{ c_1 = X.M_2.c_1; \ c_2 = c_1 + 2; \ c_3 = X.M_2.c_3 + c_2; \}; \\ M_2 = \{ c_1 = 3; \ c_2 = M_1.c_2; \ c_3 = c_1 + c_2; \}; \\ c = M_1.c_3; \}$$

To be successful, evaluation of the fields of M_1 and M_2 needs to be interleaved. However, the above example could be successfully evaluated by adopting an alternative evaluation

$$arr_{modest} : K[(x).n] \xrightarrow[\text{modest}]{} K[(r_1, \dots, r_n).n] \text{ if } x = \{r_1, \dots, r_n, r_{n+1}, \dots\} \in K$$

Fig. 14. Reduction rule for λ_{modest}

strategy which evaluates core fields of a structure as much as necessary. This alternative is originally suggested to us by Xavier Leroy, and we name it *modest-field* strategy. This section formalizes the modest-field strategy.

6.1 The target language λ_{modest} and translation for modest-field modules

The syntax of the target language λ_{modest} and the translation from *Osan* to λ_{modest} are same as those of λ_{need} , respectively given in Figures 6 and 9. As to the reduction semantics we replace rule arr_{need} in Figure 7 by rule arr_{modest} given in Figure 14. The new rule is in fact same as rule arr_{lazy} of λ_{lazy} . In the light of *Osan*, the modest-field strategy retains an invariant that when a structure is accessed, those core fields in the same and enclosing structures that exist before the accessed field are evaluated in the top-to-bottom order. Rule arr_{modest} enforces this invariant. The core fields that follow the accessed field are not evaluated, as suggested by the absence of *init* and *init-env*, in contrast to λ_{lazy} . We write $\xrightarrow[\text{modest}]{}^*$ to denote the reflexive and transitive closure of $\xrightarrow[\text{modest}]{}.$

Definition 6.1

The λ_{modest} language is the set of terms equipped with the partial function $\xrightarrow[\text{modest}]{}.$

We demonstrate the translation and the reductions of the resulting expression of the first example in this section. The example is translated into:

```

let rec x = ⟨x′⟩
and x′ =
  let rec m1 =
    let rec x1 = ⟨x′1⟩
    and x′1 =
      let rec c1 = ((x!2) I)!1 in let rec c2 = c1 + 2 in let rec c3 = ((x!2) I)!3 + c2 in {c1, c2, c3} in
      ⟨x′1⟩ in
    let rec m2 =
      let rec x2 = ⟨x′2⟩
      and x′2 = (let rec c′1 = 3 in let rec c′2 = m1!2 in let rec c′3 = c′1 + c′2 in {c′1, c′2, c′3}) in
      ⟨x′2⟩ in
    let rec c = m1!3 in
    {λ..m1, λ..m2, c} in
  x!3

```

After some work we obtain:

```

x = ⟨x′⟩ and x1 = ⟨x′1⟩ and c1 = ((x!2) I)!1 and c2 = c1 + 2 and c3 = ((x!2) I)!3 + c2
and x′1 = {c1, c2, c3} and m1 = ⟨x′1⟩
and m2 =
  let rec x2 = ⟨x′2⟩
  and x′2 = (let rec c′1 = 3 in let rec c′2 = m1!2 in let rec c′3 = c′1 + c′2 in {c′1, c′2, c′3}) in
  ⟨x′2⟩ in
  and c = (c1, c2, c3).3 and x′ = {λ..m1, λ..m2, c}
  ⊢ (λ..m1, λ..m2, c).3

```

Then the next reduction focuses on the rhs of c_1 , which reduces as follows:

```

((x!2) I)!1 → ((⟨x′⟩!2) I)!1 → (((λ..m1, λ..m2).2) I)!1 → ((λ..m2) I)!1 → m2!1
→ ⟨x′2⟩!1 → (c′1).1 → (3).1 → 3

```

Then we will have :

```

let rec x = ⟨x′⟩ and x1 = ⟨x′1⟩ and c1 = 3 and c2 = 5 and c3 = ((x!2) I)!3 + c2
and x′1 = {c1, c2, c3} and m1 = ⟨x′1⟩ and x2 = ⟨x′2⟩ and c′1 = 3 and c′2 = m1!2 and c′3 = c′1 + c′2
and x′2 = {c′1, c′2, c′3} and m2 = ⟨x′2⟩ and c = (3, 5, c3).3 and x′ = {λ..m1, λ..m2, c} in
(λ..m1, λ..m2, c).3

```

Then the next reduction focuses on the rhs of c_3 , which reduces to $(c'_1, c'_2, c'_3).3 + c_2$. Therefore c'_2 is forced and it reduces to 5; then c'_3 is forced and it reduces to 8. Thus the rhs of c_3 finally reduces to 13. The reductions terminates with an answer: :

```

x = ⟨x′⟩ and x1 = ⟨x′1⟩ and c1 = 3 and c2 = 5 and c3 = 13
and x′1 = {c1, c2, c3} and m1 = ⟨x′1⟩ and x2 = ⟨x′2⟩ and c′1 = 3 and c′2 = 5 and c′3 = 8
and x′2 = {c′1, c′2, c′3} and m2 = ⟨x′2⟩ and c = 13 and x′ = {λ..m1, λ..m2, c}
⊢ 13

```

Proposition 6.1 states the modest-field strategy is more successful than the lazy-field strategy. It is proved similarly as Proposition 5.1.

Proposition 6.1

If $\vdash Tr_L(E)_\emptyset \xrightarrow{\text{lazy}} d \vdash v$ then $\vdash Tr_N(E)_\emptyset \xrightarrow{\text{modest}} d' \vdash v'$.

$$arr_{full} \quad K[\langle x \rangle!n] \quad \xrightarrow[\text{full}]{} \quad K[r_n] \quad \text{if } x = \{\dots, r_n, \dots\} \in K$$

Fig. 15. Reduction rule for λ_{full}

6.2 Discussion

The modest-field strategy introduces more laziness, thus it permits more recursive initialization patterns. It may yield a different evaluation order from the call-by-need or lazy-field strategy, as the following example illustrates:

$$\begin{aligned} \{ M = \{ & (X) \\ & c_1 = \text{print } 1; \\ & M_1 = \{ c_1 = \text{print } 2; c_2 = X.M_2.c; c_3 = \text{print } 3; \}; \\ & c_2 = \text{print } 4; \\ & M_2 = \{ c = \text{print } 5; \}; \\ & c_3 = \text{print } 6; \}; \\ & c = M.M_1.c_3; \} \end{aligned}$$

Whereas the above program prints “1 4 6 2 5 3” in the call-by-need and lazy-field strategies, it prints “1 2 4 5 3” in the modest-field strategy.

7 Fully-lazy modules

For the completeness of our exploration we define a *fully-lazy* evaluation strategy, where a field of a structure is evaluated only when accessed.

The syntax of the target language λ_{full} as well as the translation from *Osan* to λ_{full} are same as those of λ_{need} . As to the reduction semantics we replace rule arr_{need} in Figure 7 with rule arr_{full} given in Figure 15. The notation $\xrightarrow[\text{full}]{} .$ denotes the reflexive and transitive closure of $\xrightarrow{\text{full}}$.

Definition 7.1

The λ_{full} language is the set of terms equipped with the partial function $\xrightarrow[\text{full}]{} .$

Proposition 7.1

If $\vdash Tr_N(E)_0 \xrightarrow[\text{modest}]{} d \vdash v$ then $\vdash Tr_N(E)_0 \xrightarrow[\text{full}]{} d' \vdash v'$.

λ_{full} differs from Haskell (Jones, 2003) in that it does not permit direct cycles such as `let rec x = x in x`, which simply diverges in Haskell. Our lazy letrec is rather close to value recursion (Erk ok, 2002; Erk ok *et al.*, 2002); in particular Moggi and Sabry (Moggi & Sabry, 2004) treat unsound initialization as a monadic error. A difference of *mfix* and *MFix* from lazy letrec is that in the former computations are sequenced by monadic combinators whereas in the latter bindings are evaluated on-demand. In our exploration, sequencing is imposed by variations of initializers, which evaluate elements of an array left-to-right, or fields of a structure top-to-bottom in the light of *Osan*.

8 Extensions with state

In this section we extend λ_{value} and λ_{need} with the set! operation   la Scheme (Sperber *et al.*, 2009) in the style of Felleisen and Hieb (Felleisen & Hieb, 1992); λ_{lazy} , λ_{modest} and λ_{full}

$$\begin{array}{lcl}
\text{Expressions} & a ::= & x \mid \lambda x.a \mid a_1 a_2 \mid (a, \dots) \mid a.n \mid \langle x \rangle \mid \{r, \dots\} \mid a!n \\
& & \mid \text{let rec } d \text{ in } a \mid \text{set! } x a \\
\text{Lift contexts} & L ::= & [] a \mid (\dots, v, [], a, \dots) \mid [] .n \mid [] !n \mid \text{set! } x []
\end{array}$$
Fig. 16. Syntax for λ_{need} with state
$$\begin{array}{lcl}
\text{set}_{value} : & D \text{ and } x = v' \text{ and } D' \vdash N[\text{set! } x v] \xrightarrow[\text{value}]{} D \text{ and } x = v \text{ and } D' \vdash N[v] \\
\text{set-env}_{value} : & D \text{ and } x' = v' \text{ and } D' \text{ and } x = N[\text{set! } x v] \text{ and } d \vdash a \\
& \xrightarrow[\text{value}]{} D \text{ and } x' = v \text{ and } D' \text{ and } x = N[v] \text{ and } d \vdash a
\end{array}$$
Fig. 17. Reduction rules for set! for λ_{value}

can be extended as well in the same way as λ_{need} . The ML core language supports reference cells with primitives for creating, reading from and writing to reference cells. λ_{value} and λ_{need} could be extended with these primitives. However, adding a single primitive set! is sufficient to demonstrate how the target languages can accommodate state.

The extension is in fact straightforward; in other words, we have formulated the operational semantics of the target languages anticipating the extension. We extend expressions of λ_{value} and λ_{need} with set! $x a$ and lift contexts with set! $x []$; e.g., Figure 16 gives the extended syntax for λ_{need} . set! eagerly evaluates the expression to be written. This choice reflects our desire to study lazy modules in the setting of a call-by-value core language.

Figures 17 and 18 respectively present two new rules to be added to the reduction rules of λ_{value} and λ_{need} ; they are self-explanatory. λ_{value} allows assignments to (already-evaluated) preceding bindings, but not to (yet-to-be-evaluated) subsequent bindings; this behaviour is inspired by Scheme's *letrec**. Given that we are concerned with a call-by-value core language, it could be a moot point that λ_{need} allows a variable to be updated without forcing evaluation of the expression initially bound to the variable. That is, an alternative rule for set! could be

$$x = v' \text{ and } d \vdash N[\text{set! } x v] \xrightarrow[\text{need}]{} x = v \text{ and } d \vdash N[v]$$

by extending productions of dereferences with set! $x v$.

An immediate consequence of adding state is that it breaks inclusion between the strategies.

9 Related work

Technical development of the paper is very much inspired by Felleisen and Hieb's call-by-value lambda calculus with state (Felleisen & Hieb, 1992) and Ariola and Felleisen's call-by-need cyclic lambda calculus (Ariola & Felleisen, 1997) for the target languages, the OCaml compiler (Leroy *et al.*, 2008) for the translation, and our previous work (Nakata & Hasegawa, 2009), which proved equivalence of a reduction semantics for call-by-need in the style of Ariola and Felleisen to a corresponding natural semantic in the style of Launchbury (Launchbury, 1993) and Sestoft (Sestoft, 1997). In (Felleisen & Friedman, 1986), Felleisen and Friedman considered several different semantics for imports and exports, in the course of demonstrating a way to incorporate modules into Scheme 84 (Friedman *et al.*, 1985); our work shares the same spirit with them by exploring the design space of different evaluation strategies.

$$\begin{array}{l}
\text{set} : \quad x = a \text{ and } d \vdash N[\text{set! } x \ v] \xrightarrow{\text{need}} x = v \text{ and } d \vdash N[v] \\
\text{set-env} : \quad x'' = a \text{ and } x' = N[\text{set! } x'' \ v] \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\#x] \\
\quad \quad \quad \xrightarrow{\text{need}} x'' = v \text{ and } x' = N[v] \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\#x]
\end{array}$$

Fig. 18. Reduction rules for set! in λ_{need}

OCaml's recursive modules The experimental implementation of recursive modules in OCaml was our starting point of the study (Leroy, 2003). In the family of the ML module system, OCaml's modules are evaluated eagerly. Initialization of recursive modules signals exceptions at runtime in case of unsupported forms of recursive initialization patterns. The compiler places a restriction on possible signatures of recursive modules, where the restriction allows the compiler to implement recursive modules efficiently: there must be at least one “safe” module, of which all the core fields have function types, for a dependency cycle between modules simultaneously defined with a module `rec`. Then evaluation of a module `rec` definition proceeds by building initial values for the safe modules involved, whose core fields are initially set to exception-raising default values; the default values for the safe modules are updated afterward by values thus computed. The evaluation order of the modules in a module `rec` may differ from the textual definition order, since the compiler infers a more successful evaluation order according to the dependency cycles it computes. Notably OCaml permits taking fix-points of functors without expansions of arguments. For instance, the following program successfully evaluates in OCaml:

```

module F = functor(X : sig val f : int → int end) →
  struct let f x = if x = 0 then 1 else x * X.f (x - 1) end
module rec Fix : sig val f : int → int end = F(Fix)

```

At the same time, unsound initialization might not be signaled at the initialization time and the programmer may trigger an exception on attempting to access a field of a recursively defined module after the evaluation phase of the module has been completed. For instance in the following program the first line successfully evaluates and the second line raises an exception:

```

module rec M : sig val f : int → int = M // Unsoundness is not signaled.
let _ = M.f 3 // Exception: Undefined_recursive_module

```

OCaml's strategy also entails the potential of incomplete handling of value aliases, i.e., core definitions of the form `let c = p.c'`, in the sense that a referring field might not be updated but continue to hold a default value, although the referred field has been initialized to a proper value.

Moscow ML's recursive modules Moscow ML implements recursive modules using back-patching (Russo, 2001; Romanenko *et al.*, 2004). A self variable is initialized to a reference to a cell labeled with, say NONE. Then the corresponding recursive structure is initialized, and the cell is updated with the computed value as well as the label being updated with SOME. Dereferencing a self variable entails runtime check of the label, and raises an exception when the label is NONE. In comparison with OCaml, Moscow ML does not

permit taking a fixpoint directly. Therefore the following program fails to evaluate:

```

functor F(X : sig val f : int → int end) = struct
  fun f x = if x = 0 then 1 else x * X.f (x - 1) end
structure FIX = rec(X : sig structure Fix : sig val f : int → int end end)
  struct structure Fix = F(X.Fix) end // Uncaught exception: Bind

```

One has to eta-expand the argument:

```

structure FIX = rec(X : sig structure Fix : sig val f : int → int end end)
  struct structure Fix = F(struct fun f x = X.Fix.f x end) end

```

But Moscow ML properly signals unsound initialization for the following program:

```

structure M = rec(X : sig structure M' : sig val f : int → int end end)
  struct structure M' = X.M' end // Uncaught exception: Bind

```

Besides, unlike OCaml, Moscow ML permits defining “recursive functors” such as:

```

structure Funct =
  rec (X : sig functor F : functor(Y : sig end) → sig val next : int → int end end)
  struct
    functor F(Y : sig end) = struct
      structure Next = X.F(Y)
      fun next n = if n = 0 then 0 else Next.next (n - 1)
    end
  end

```

As expected, evaluation of any instantiation of *Funct.F* diverges; it would not diverge in the call-by-need strategy of Section 4, where the sub-module *Next* is evaluated lazily.

Syme’s initialization graphs Syme proposed a relaxed form of recursive bindings for a ML-like call-by-value language, named initialization graphs, to help build mutually referential graphs using lazy evaluation, where initialization soundness is checked at runtime (Syme, 2005). Each node of an initialization graph is a delayed computation and at the creation time all the nodes are forced eagerly and sequentially. His work motivated us to study lazy evaluation in the context of recursive modules. But our technical development is different. We consider the module language focusing on evaluation strategies for records (i.e., modules), whereas Syme considers the core language and no special attention is paid to records.

PLT Scheme’s units PLT Scheme implements units using back-patching (Flatt & Felleisen, 1998; Owens & Flatt, 2006; Owens, 2007; Flatt *et al.*, 2009). Dreyer and Rossberg’s evidence translation for MixML also closely follows this approach and uses a back-patching semantics (Dreyer & Rossberg, 2008). A unit is initialized when it is invoked. Each constituent unit of the invoked unit is initialized eagerly; the order of the initialization is determined by the order of declarations of the constituent units in the link clause. Each member of a constituent unit is initialized eagerly top-to-bottom along the definition order. While units are initialized eagerly, the separation of linking and initialization allows for

more flexible recursive initialization patterns than the call-by-value modules including fix-points patterns. Initialization of units uses a default value, *undefined*, to initialize recursive bindings, allowing arbitrary expressions to be at the right-hand side of the bindings. In PLT Scheme *undefined* is a permissible value, and unsound initialization might not be signaled when a unit is invoked. For instance in the following code unit $w@$ is initialized without errors, while the exported value y is bound to *undefined*.

```
(define-signature  $\hat{u}(x)$ )
(define-signature  $\hat{v}(y)$ )
(define-unit  $u@$  (import  $\hat{u}$ ) (export  $\hat{v}$ ) (define  $y x$ ))
(define-unit  $v@$  (import) (export  $\hat{u}$ ) (define  $x 3$ ))
(define-compound-unit/infer  $w@$  (import) (export  $\hat{v}$ ) (link  $u@ v@$ ))
(define-values/invoke-unit/infer  $w@$ )
```

Although this behaviour is consistent with PLT Scheme's *letrec*, we found it unsatisfactory in that it potentially introduces a similar problem as the problems with null pointers in object-oriented languages.

PLT Scheme provides an interesting construct, *init-depend* declarations, which constrains the allowed orders of linking. A order violation is signaled at runtime when a *compound-unit* expression is evaluated. For instance the incorrect linking order in the above code could be signaled using *init-depend*; but *init-depend* does not eliminate potentials of unreported unsound initialization. Knit (Reid *et al.*, 2000) further develops this idea by requiring the programmer to explicitly provide information about initialization and finalization functions and constraints on the initialization order. Knit uses this information to schedule the initialization and finalization of units.

We are currently working on (re-)implementing PLT Scheme's units using *delay* and *force* operators to experiment with a constraint system on the initialization for units (Nakata, 2009).

Java's classes and objects In Java, classes are loaded and initialized lazily, whereas objects are initialized when they are created (Gosling *et al.*, 2005). The fields of a class or an object are first initialized implicitly by the JVM to default values according to their types, then are supposed to be updated explicitly by the programmer to meaningful values through for instance field initializers or a constructor call. Explicit reinitialization is not mandatory, and any field is accessible during the explicit reinitialization, therefore initial default values can be observed.

F#'s classes and objects In F#, implementation files, modules and classes are initialized lazily (Syme & Margetson, 2008). In particular dynamic-link libraries (DLLs) are not initialized when they are loaded. Objects are initialized when they are created. F#'s object initialization semantics is different from that of Java. Fields of an object are required to be initialized explicitly. The self variable of an object cannot be dereferenced before initialization is complete. Such an attempt is signaled at runtime: if the self variable is used anywhere in the initialization code, then the compiler inserts a runtime check so that an exception is signaled if the self variable is dereferenced during initialization. According

to the language specification, these design choices are motivated to reduce the presence of null pointers.

Equational theories for lambda calculi with state Felleisen and Hieb (Felleisen & Hieb, 1992) studied equational theories for the call-by-value λ_v -calculus of Plotkin (Plotkin, 1975) extended with control and state. Using ideas developed in (Felleisen & Hieb, 1992), Ariola and Sabry (Ariola & Sabry, 1998) introduced an imperative call-by-need lambda calculus, a variation of the call-by-need cyclic lambda calculus of Ariola and Felleisen extended with destructive operations on reference cells, to formalize an implementation of monadic state and prove its correctness. Moggi and Sabry (Moggi & Sabry, 2004) studied equational theories for value recursion (Erkök, 2002; Erkök & Launchbury, 2000) in monadic metalanguages; Moggi and Sabry’s calculus treats unsound initialization as a monadic error, rather than divergence. These works use different machineries to thread effectful computations consistently, in accordance with the different underlying semantics. Felleisen and Hieb’s and Moggi and Sabry’s calculi are confluent. Ariola and Sabry’s calculus has unique infinite normal forms. Our technical development is also inspired by those works.

The treatment of variables and locations is noteworthy. Variables are not values in (Ariola & Felleisen, 1997). (Felleisen & Hieb, 1992) distinguishes assignable variables and binding variables, where the latter are values but the former are not; the distinction is motivated to faithfully extend Plotkin’s λ_v -calculus, where variables are values. (Ariola & Sabry, 1998) introduces different constructs for variables and locations, where the latter are values but the former are not; locations are explicitly created, read and written by the primitives *new*, *read* and *write* respectively. We do not consider variables as values and introduced an “address-of” construct $\langle \cdot \rangle$ which turns a variable into a value.

Mixin modules In (Ancona *et al.*, 2003), Ancona *et al.* proposed a purely functional mixin calculus (Ancona & Zucca, 2002) equipped with recursive monadic bindings (Erkök & Launchbury, 2000) in the style of Moggi and Fagorzi’s monadic metalanguages (Moggi & Fagorzi, 2003). In a series of papers (Ancona *et al.*, 2004; Ancona *et al.*, 2004; Fagorzi & Zucca, 2007) they studied lazy module operators, or dynamic reconfiguration of modules, which can be performed after selecting and executing a member of a module. While dynamic reconfiguration would be an interesting feature, it is not the subject of our paper.

Further discussions Standard ML (Milner *et al.*, 1997b) requires the right-hand side of recursive bindings to be syntactic abstractions. OCaml (Leroy *et al.*, 2008) relaxes this restriction and allows limited forms of constructor applications. In (Hirschowitz *et al.*, 2003), Hirschowitz *et al.* studied an efficient compilation scheme of a wider range of recursive definitions than is allowed in OCaml. Haskell (Jones, 2003) avoids unsound initialization problems altogether by lazy evaluation.

There have been proposals to statically guarantee initialization soundness for ML-style call-by-value modules with recursion (Boudol, 2004; Dreyer, 2004), call-by-value mixin modules (Hirschowitz & Leroy, 2005), and objects (Fähndrich & Xia, 2007; Qi & Myers, 2009). These works seek program invariants e.g., when initialization is completed, with respect to a particular semantics, whereas we studied invariants between different semantics.

It has been known that equational theories for letrec poses a subtle issue on confluence (Ariola & Klop, 1994; Ariola & Blom, 2002), which is not the subject of the paper. Our target languages implement the restriction suggested by Ariola and Felleisen (Ariola & Felleisen, 1997): substitution is performed only under evaluation contexts. Moreover reductions terminate when they encounter a cyclic dereference, which we deem as unsound initialization.

10 Conclusion

We have studied five evaluation strategies for ML-style modules supporting recursion by gradually introducing laziness. The objective of the paper is not to convince the reader that one strategy is better than another, but to shed light on the design space of different evaluation strategies and to present a tool and proof techniques to explore them. We have interactively experimented with the reduction semantics using PLT Redex, which was indeed both interesting and instructive for us³.

One compelling direction for future work would be to study interaction between lazy modules and exception handling mechanisms. That is, an exception may be raised during evaluation of a module, which is caught outside the module:

$$\{ M = \{ c_1 = \text{fun } i \rightarrow i + 1; c_2 = \text{raise } \text{Stop}; c_3 = c_1\ 3; \}; \\ c = \text{try } M.c_2 \text{ with } \text{Stop} \rightarrow M.c_3; \}$$

The question is when to evaluate core field $M.c_3$ if M is evaluated lazily accessed by c in the call-by-need or the lazy-field strategy. We may evaluate $M.c_3$ when accessed in the exception handling branch of c . Or, we may sternly cause a runtime error whenever module evaluation is aborted halfway as above. There seems to be interesting design space here too.

Acknowledgment

I owe this paper to many people. I am grateful to an anonymous reviewer of previous drafts on the earlier work, who referred me to Ariola and Felleisen's work and suggested I should look at different strategies. I am grateful to Matthias Felleisen, the corresponding editor, who referred me to Felleisen and Hieb's work and let me know that PLT Scheme has a primitive for the initializer. Their comments were most influential on the development of this work. I am grateful to Xavier Leroy for valuable advice and many pleasant discussions, during the earlier development, when I was a post-doc in the Gualium team at INRIA Rocquencourt. I thank Andrew Tolmach and Jacques Garrigue for discussions. I thank Don Syme for encouragement on working on this subject, and discussions on the initialization semantics of F#'s object system. I thank reviewers of previous drafts on the earlier work for their constructive comments.

³ Due to technical reasons we were unable to implement lazy letrec in the same way as is formalized in the paper. Our implementation in PLT Redex uses back-patching; it is straightforward to prove this "implementation" in PLT Redex correct with respect to the "specification" in the paper.

References

- Ancona, D., & Zucca, E. (2002). A calculus of module systems. *Journal of Functional Programming*, **12**(2), 91–132.
- Ancona, D., Fagorzi, S., Moggi, E., & Zucca, E. (2003). Mixin modules and computational effects. *Pages 224–238 of: Baeten, J. C. M., Lenstra, J. K., Parrow, J., & Woeginger, G. J. (eds), Proc. International Colloquium on Automata, Languages and Programming (ICALP)*. Lecture Notes in Computer Science, vol. 2719. Eindhoven, The Netherlands: Springer.
- Ancona, D., Fagorzi, S., & Zucca, E. (2004). A calculus with lazy module operators. *Pages 423–436 of: Lévy, J., Mayr, E. W., & Mitchell, J. C. (eds), Proc. IFIP International Conference on Theoretical Computer Science (IFIP TCS)*. Toulouse, France: Kluwer.
- Ariola, Z., & Felleisen, M. (1997). The call-by-need lambda calculus. *Journal of Functional Programming*, **7**(3), 265–301.
- Ariola, Z. M., & Blom, S. (2002). Skew confluence and the lambda calculus with letrec. *Ann. pure appl. logic*, **117**(1-3), 95–168.
- Ariola, Z. M., & Klop, J. W. (1994). Cyclic lambda graph rewriting. *Pages 416–425 of: Proc. Symposium on Logic in Computer Science (LICS)*.
- Ariola, Z. M., & Sabry, A. (1998). Correctness of monadic state: An imperative call-by-need calculus. *Pages 62–74 of: Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. San Diego, CA, USA.: ACM.
- Boudol, G. (2004). The recursive record semantics of objects revisited. *Journal of Functional Programming*, **14**, 263–315.
- Crary, K., Harper, R., & Puri, S. (1999). What is a recursive module? *Pages 50–63 of: Proc. of Programming Language Design and Implementation (PLDI)*.
- Dreyer, D. (2004). A type system for well-founded recursion. *Pages 293–305 of: Jones, N. D., & Leroy, X. (eds), Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*. Venice, Italy: ACM.
- Dreyer, D. (2005). Recursive Type Generativity. *Pages 41–53 of: Proc. of ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM.
- Dreyer, D., & Rossberg, A. (2008). Mixin’ up the ml module system. *Pages 307–320 of: Hook, J., & Thiemann, P. (eds), Proc. ACM SIGPLAN international conference on Functional programming (ICFP)*. Victoria, BC, Canada: ACM.
- Erkok, L. (2002). *Value recursion in monadic computations*. Ph.D. thesis, Oregon Graduate Institute School of Science Engineering, OHSU.
- Erkok, L., & Launchbury, J. (2000). Recursive monadic bindings. *Pages 174–185 of: Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Montreal, Canada: ACM.
- Erkok, L., Launchbury, J., & Moran, A. (2002). Semantics of value recursion for monadic input/output. *Theoretical informatics and applications*, **36**(2), 155–180.
- Fagorzi, S., & Zucca, E. (2007). A calculus of open modules: call-by-need strategy and confluence. *Mathematical structures in computer science*, **17**(4), 675–751.
- Fahndrich, M., & Xia, S. (2007). Establishing object invariants with delayed types. *Pages 337–350 of: Gabriel, R. P., Bacon, D. F., Lopes, C. V., & Jr., G. L. Steele (eds), Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. Montreal, Quebec, Canada: ACM.
- Felleisen, M., & Friedman, D. P. (1986). A closer look at export and import statements. *Comput. Lang.*, **11**(1), 29–37.
- Felleisen, M., & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, **103**(2), 235–271.

- Felleisen, M., Findler, R. B., & Flatt, M. (2009). *Semantics engineering with plt redex*. The MIT Press.
- Fisher, K., & Reppy, J. H. (1999). The design of a class mechanism for moby. *Pages 37–49 of: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Atlanta, Georgia, USA: ACM.
- Flatt, M., & Felleisen, M. (1998). Units: Cool modules for hot languages. *Pages 236 – 248 of: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. Montreal, Canada: ACM press.
- Flatt, Matthew, *et al.* . 2009 (July). *Reference: PLT scheme*. Reference Manual PLT-TR2009-reference-v4.2.1. PLT Scheme Inc. Available at: <http://plt-scheme.org/techreports/> (Accessed 18 September 2009).
- Friedman, D., Haynes, C., Kohlbecker, E., & Wand, M. (1985). *Scheme 84 interim reference manual*. Tech. rept. 153. Indiana University.
- Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *The javaTM language specification*. 3 edn. ADDISON-WESLEY.
- Hirschowitz, T., & Leroy, X. (2005). Mixin modules in a call-by-value setting. *ACM Trans. Program. Lang. Syst.*, **27**(5), 857–881.
- Hirschowitz, T., Leroy, X., & Wells, J. B. (2003). Compilation of extended recursion in call-by-value functional languages. *Pages 160–171 of: Proc. International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. Uppsala, Sweden: ACM.
- Jones, S. P. (ed). (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.
- Launchbury, J. (1993). A natural semantics for lazy evaluation. *Proc. of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL)*.
- Leroy, X. (2000). A modular module system. *Journal of Functional Programming*, **10**(3), 269–303.
- Leroy, X. (2003). *A proposal for recursive modules in Objective Caml*. Available at http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf.
- Leroy, X., Doligez, D., Garrigue, J., Rémy, D., & Vouillon, J. 2008 (November). *The Objective Caml system, release 3.11*. Software and documentation available on the Web, <http://caml.inria.fr/> (Accessed 18 September 2009).
- Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997a). *The Definition of Standard ML - Revised*. The MIT Press.
- Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997b). *The definition of standard ml, revised edition*. The MIT Press.
- Moggi, E., & Fagorzi, S. (2003). A monadic multi-stage metalanguage. *Pages 358–374 of: Gordon, A. D. (ed), Proc. International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. Lecture Notes in Computer Science, vol. 2620. Warsaw, Poland: Springer.
- Moggi, E., & Sabry, A. (2004). An abstract monadic semantics for value recursion. *Theoretical Informatics and Applications*, **38**(4).
- Nakata, K. (2009). *Lazy mixin modules and disciplined effects*. [arXiv:0908.3650v1](https://arxiv.org/abs/0908.3650v1) [cs.PL].
- Nakata, K., & Garrigue, J. (2006). Recursive Modules for Programming. *Proc. of ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM.
- Nakata, K., & Hasegawa, M. (2009). *Small-step and big-step semantics for call-by-need*. To appear in *Journal of Functional Programming*. The draft available at <http://cs.ioc.ee/~keiko/petitcbn.pdf>.
- Odersky, M., Cremet, V., Röckl, C., & Zenger, M. (2003). A nominal theory of objects with dependent types. *Pages 201–224 of: Proc. of 17th European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2743. Springer.

- Owens, S. (2007). *Compile-time information in software components*. Ph.D. thesis, University of Utah.
- Owens, S., & Flatt, M. (2006). From structures and functors to modules and units. *Pages 87–98 of: Reppy, J. H., & Lawall, J. L. (eds), Proc. of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Portland, Oregon, USA: ACM.
- Plotkin, G. D. (1975). Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, **1**(2), 125–159.
- Qi, X., & Myers, A. C. (2009). Masked types for sound object initialization. *Pages 53–65 of: Shao, Z., & Pierce, B. C. (eds), Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*.
- Reid, A., Flatt, M., Stoller, L., Lepreau, J., & Eide, E. (2000). Knit: Component Composition for Systems Software. *Pages 347–360 of: Proc. of USENIX Symposium on Operating System Design and Implementation (OSDI)*. San Diego, California, USA: USENIX Association.
- Romanenko, S., Russo, C., Kokholm, N., & Sestoft, P. 2004 (January). *Moscow ML, version 2.01*. Software and documentation available on the Web, <http://www.dina.dk/~sestoft/mosml.html> (Accessed 18 September 2009).
- Russo, C. (2001). Recursive Structures for Standard ML. *Pages 50–61 of: Proc. of ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM.
- Scales, D. J., Randall, K. H., Ghemawat, S., & Dean, J. (2000). *The swift java compiler: Design and implementation*. Tech. rept. WRL-2000-2. Compaq & DEC.
- Sestoft, P. (1997). Deriving a lazy abstract machine. *Journal of Functional Programming*, **7**(3), 231–264.
- Sperber, M., Dybvig, R. K., Flatt, M., Straaten, A. V., Findler, R., & Matthews, J. (2009). Revised⁶ report on the algorithmic language scheme. *Journal of Functional Programming*, **19**(S1), 1–301.
- Syme, D. (2005). Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge. *Pages 3–25 of: Proc. of the ACM-SIGPLAN Workshop on ML*.
- Syme, D., & Margetson, J. 2008 (September). *The F# Programming Language, version 1.9.6.2*. Software and documentation available on the Web, <http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/> (Accessed 18 September 2009).

A Proof

We prove inclusion of the call-by-value strategy into the call-by-need strategy, and the call-by-need strategy into the lazy-field strategy in the following steps:

1. We present natural semantics for λ_{value} , λ_{need} and λ_{lazy} and prove they evaluate an expression to a value if and only if the corresponding reduction semantics reduce the same expression to a result in the case of λ_{value} , or an answer in the case of λ_{need} and λ_{lazy} .
2. We prove inclusion between those natural semantics.
3. The inclusion between the strategies follows from the inclusion between the reduction semantics, which is ascribed to the inclusion between the corresponding natural semantics.

A.1 Call-by-value \subset Call-by-need

In Figure A 1 we define the natural semantics for λ_{value} . A metavariable X ranges over sets of variables. Metavariables Ψ and Φ range over finite mappings from variables to

$$\begin{array}{c}
\textit{Value} \\
\frac{}{\vdash_v \langle \Psi \rangle v \downarrow_X \langle \Psi \rangle v} \\
\textit{Application} \\
\frac{\frac{\vdash_v \langle \Psi \rangle a_1 \downarrow_X \langle \Psi' \rangle \lambda x.a \quad \vdash_v \langle \Psi' \rangle a_2 \downarrow_X \langle \Psi'' \rangle v'}{\vdash_v \langle \Psi'' \rangle [x' \mapsto v']} \quad \vdash_v \langle \Psi'' \rangle [x' \mapsto v'] a[x'/x] \downarrow_X \langle \Phi \rangle v \quad x' \text{ fresh}}{\vdash_v \langle \Psi \rangle a_1 a_2 \downarrow_X \langle \Phi \rangle v} \\
\textit{Variable} \\
\frac{\vdash_v \langle \Psi|_{\bar{x}} \rangle \Psi(x) \downarrow_{X \cup \{x\}} \langle \Phi \rangle v \quad x \in \text{dom}(\Psi)}{\vdash_v \langle \Psi \rangle x \downarrow_X \langle \Phi[x \mapsto v] \rangle v} \\
\textit{Leterc} \\
\frac{\frac{\vdash_v \langle \Psi \rangle a'_1 \downarrow_{X \cup \{x'_1, \dots, x'_n\}} \langle \Psi_1 \rangle v_1 \quad \vdash_v \langle \Psi_1[x'_1 \mapsto v_1] \rangle a'_2 \downarrow_{X \cup \{x'_2, \dots, x'_n\}} \langle \Psi_2 \rangle v_2 \quad \dots}{\vdash_v \langle \Psi_n[x'_n \mapsto v_n] \rangle a' \downarrow_X \langle \Phi \rangle v \quad x'_1, \dots, x'_n \text{ fresh}}}{\vdash_v \langle \Psi \rangle \text{let rec } x_1 = a_1 \text{ and } \dots \text{ } x_n = a_n \text{ in } a \downarrow_X \langle \Phi \rangle v} \\
\textit{Array} \\
\frac{\vdash_v \langle \Psi \rangle a \downarrow_X \langle \Psi' \rangle \langle x \rangle \quad \vdash_v \langle \Psi' \rangle x \downarrow_X \langle \Psi'' \rangle \{ \dots, x_n, \dots \} \quad \vdash_v \langle \Psi'' \rangle x_n \downarrow_X \langle \Phi \rangle v}{\vdash_v \langle \Psi \rangle a!n \downarrow_X \langle \Phi \rangle v}
\end{array}$$

Fig. A 1. Natural semantics for λ_{value}

$$\begin{array}{ll}
FV(x) & = \{x\} \\
FV(\lambda x.a) & = FV(a) \setminus \{x\} \\
FV(a_1 a_2) & = FV(a_1) \cup FV(a_2) \\
FV(\langle x \rangle) & = \{x\} \\
FV(\{x, \dots\}) & = \{x, \dots\} \\
FV(a!n) & = FV(a) \\
FV(\text{let rec } x_1 = a_1 \text{ and } \dots \text{ in } a) & = \{FV(a_1) \cup \dots \cup FV(a)\} \setminus \{x_1, \dots\}
\end{array}$$

Fig. A 2. Free variables

expressions. The notation $\text{dom}(\Psi)$ denotes the domain of Ψ . The notation $\Psi|_{\bar{x}}$ denotes the restriction of Ψ to $\text{dom}(\Psi) \setminus \{x\}$. The notation $a[x'/x]$ denotes substitution of x' for the free occurrences of x in a . The notion of free variables is standard and is defined in Figure A 2. In *Leterc* rule, a'_i 's and a' denote expressions obtained from a_i 's and a by substituting x'_i 's for x_i 's respectively. We may abbreviate $\langle \Psi \rangle M$ where Ψ is an empty mapping, i.e. the domain of Ψ is empty, to $\langle \rangle M$. Following Sestoft (Sestoft, 1997) the semantics keeps track of variables which are temporarily deleted from the heap in *Variable* rule to make freshness conditions locally checkable. When heaps only map variables to values, *Variable* rule has the same effect as the following alternative formulation:

$$\frac{x \in \text{dom}(\Psi)}{\vdash_v \langle \Psi \rangle x \downarrow_X \langle \Psi \rangle \Psi(x)}$$

The present formulation however facilitates later proofs when heaps may contain reducible expressions. The natural semantics does not assume implicit α -renaming, but works with (raw) expressions.

The notation $\lceil d \rceil$ denotes a heap Ψ such that $\text{dom}(\Psi) = \text{dom}(d)$ and for all x in $\text{dom}(\Psi)$, $\Psi(x) = a$ iff d contains $x = a$, where $\text{dom}(x = a \text{ and } \dots) = \{x, \dots\}$. A heap Ψ is *evaluated*

if for all x in $\text{dom}(\Psi)$, $\Psi(x)$ is a value. We write $d \vdash a \xrightarrow[\text{value}]{}^n d' \vdash a'$ to denote $d \vdash a$ reduces to $d' \vdash a'$ in n -steps in λ_{value} .

Lemma A.1

D and $x = a$ and $d \vdash a' \xrightarrow[\text{value}]{}^n D'$ and $x = v$ and $d \vdash a'$ iff $D \vdash a \xrightarrow[\text{value}]{}^n D' \vdash v$

Proof

By simultaneous induction on the length of the reductions of D and $x = a$ and $d \vdash a'$ and $D \vdash a$. \square

Lemma A.2

If $D \vdash L[a] \xrightarrow[\text{value}]{}^n D' \vdash L[v]$ then $D \vdash a \xrightarrow[\text{value}]{}^{n'} D' \vdash v$ with $n' \leq n$.

Proof

By induction on the length of the reductions of $D \vdash L[a]$. \square

Lemma A.3

If $D \vdash a \xrightarrow[\text{value}]{} D' \vdash v$ then $D \vdash L[a] \xrightarrow[\text{value}]{} D' \vdash L[v]$

Proof

By induction on the length of the reductions of $D \vdash a$. \square

Proposition A.1

The following two conditions hold:

1. if $D \vdash a \xrightarrow[\text{value}]{} D' \vdash v$, then for any X such that $\text{dom}(D)$ and X are disjoint, there exists a configuration $D'' \vdash v'$ such that $D' \vdash v$ and $D'' \vdash v'$ belong to the same α -equivalence class and $\vdash_v \langle [D] \rangle a \downarrow_X \langle [D''] \rangle v'$;
2. if $\text{dom}(\Psi)$ and X are disjoint and Ψ is evaluated and $\vdash_v \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$, then for any D such that $[D] = \Psi$, $D \vdash a \xrightarrow[\text{value}]{} D' \vdash v$ where $[D'] = \Phi$.

Proof

1. By induction on the length of the reductions of $D \vdash a$ with case analysis on a . Use Lemmas A.1 and A.2.
2. By induction on the derivation of $\vdash_v \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$ with case analysis on the last rule used. Use Lemmas A.1 and A.3.

\square

Corollary A.1

The following two conditions hold:

1. if $\vdash a \xrightarrow[\text{value}]{} D \vdash v$, then there exists a configuration $D' \vdash v'$ such that $D \vdash v$ and $D' \vdash v'$ belong to the same α -equivalence class and $\vdash_v \langle \rangle a \downarrow_\emptyset \langle [D'] \rangle v'$;
2. if $\vdash_v \langle \rangle a \downarrow_\emptyset \langle \Psi \rangle v$ then $\vdash a \xrightarrow[\text{value}]{} D \vdash v$ where $[D] = \Psi$.

To relate the call-by-value translation $Tr_V(\cdot)$ and the call-by-need translation $Tr_N(\cdot)$, we syntactically distinguish module and core fields and their accessors by labelling them. More precisely we label a module field by m , and a core field by c ; $a!n$ is an accessor for a core field, and $a!!n$ is for a module field. Accordingly we revise the call-by-value

translation by replacing rules *mfld*, *cfld* and *mpath* by the following rules to reflect the labelling.

$$\begin{aligned}
cfld : \quad & TrFld_V(c = e; f; l : x, \dots)_\rho = \\
& \text{let rec } x = TrC_V(e)_\rho \text{ in } TrFld_V(f; l : x, \dots, c : x')_{\rho[c \mapsto x']} \quad (x' \text{ fresh}) \\
mfld : \quad & TrFld_V(M = E; f; l : x, \dots)_\rho = \\
& \text{let rec } x = Tr_V(E)_\rho \text{ in } TrFld_V(f; l : x, \dots, m : x')_{\rho[M \mapsto x']} \quad (x' \text{ fresh}) \\
mpath : \quad & Tr_V(p.n)_\rho = Tr_V(p)_\rho !! n
\end{aligned}$$

where l denotes either c or m . The reduction semantics for labelled terms is defined by identifying a labelled term with its original term; the labelling is merely for a bookkeeping purpose.

An expression a is a *by-need variant* of a' , written $a <: a'$, if a is obtained from a' by substituting λ_x for $m : x$, x for $c : x$, and $(a!n) I$ for $a!!n$ respectively.

A configuration $\langle \Psi \rangle a$ is a *by-need variant* of $\langle \Psi' \rangle a'$, written $\langle \Psi \rangle a <: \langle \Psi' \rangle a'$, if the following conditions holds:

1. $a <: a'$
2. $dom(\Psi') \subseteq dom(\Psi)$
3. for any x in $dom(\Psi')$, $\Psi(x) <: \Psi'(x)$
4. for any x in $dom(\Psi) \setminus dom(\Psi')$, x does not appear in a or $\Psi(x')$ for any x' in $dom(\Psi)$.

The natural semantics for λ_{value} can be obviously extended to handle arrays containing delayed variables:

$$\frac{\text{Array} \quad \vdash_v \langle \Psi \rangle a \downarrow_X \langle \Psi' \rangle \langle x \rangle \quad \vdash_v \langle \Psi' \rangle x \downarrow_X \langle \Psi'' \rangle \{ \dots, r_n, \dots \} \quad \vdash_v \langle \Psi'' \rangle r_n \downarrow_X \langle \Phi \rangle v}{\vdash_v \langle \Psi \rangle a!n \downarrow_X \langle \Phi \rangle v}$$

Lemma A.4

If $dom(\Psi')$ and X are disjoint and $\langle \Psi' \rangle a' <: \langle \Psi \rangle a$ and $\vdash_v \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$, then $\vdash_v \langle \Psi' \rangle a' \downarrow_X \langle \Phi' \rangle v'$ and $\langle \Phi' \rangle v' <: \langle \Phi \rangle v$.

Proof

By induction on the derivation of $\vdash_v \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$. \square

In Figure A 3 we define the natural semantics for λ_{need} . In *Letrec* rule a'_i 's and a' denote expressions obtained from a_i 's and a by substituting x'_i 's for x_i 's respectively. We write $d \vdash a \xrightarrow[\text{need}]{}^n d' \vdash a'$ to denote $d \vdash a$ reduces to $d' \vdash a'$ in n -steps in λ_{need} .

Lemma A.5

$x' = a$ and $d^*[x, x']$ and $d \vdash N[x] \xrightarrow[\text{need}]{}^n x' = v$ and $d^*[x, x']$ and $d' \vdash N[x]$ iff $d \vdash a \xrightarrow[\text{need}]{}^n d' \vdash v$

Proof

By simultaneous induction on the length of the reductions of $x' = a$ and $d^*[x, x']$ and $d \vdash N[x]$ and $d \vdash a$ with case analysis on the possible reductions of a . We only consider the main cases; the other cases are either obvious or similar to the last case below.

- The case in which $a \xrightarrow[\text{need}]{} a'$ is easy.

Lazy modules

35

$$\begin{array}{c}
\text{Value} \\
\frac{}{\vdash_n \langle \Psi \rangle v \downarrow_X \langle \Psi \rangle v} \\
\text{Application} \\
\frac{\vdash_n \langle \Psi \rangle a_1 \downarrow_X \langle \Phi \rangle \lambda x.a \quad \vdash_n \langle \Phi[x' \mapsto a_2] \rangle a[x'/x] \downarrow_X \langle \Psi' \rangle v \quad x' \text{ fresh}}{\vdash_n \langle \Psi \rangle a_1 a_2 \downarrow_X \langle \Psi' \rangle v} \\
\text{Variable} \\
\frac{\vdash_n \langle \Psi[\bar{x}] \rangle \Psi(x) \downarrow_{X \cup \{x\}} \langle \Phi \rangle v}{\vdash_n \langle \Psi \rangle x \downarrow_X \langle \Phi[x \mapsto v] \rangle v} \\
\text{Letrec} \\
\frac{\vdash_n \langle \Psi[x'_i \mapsto a'_i]_{i \in \{1, \dots, n\}} \rangle a' \downarrow_X \langle \Phi \rangle v \quad x'_1, \dots, x'_n \text{ fresh}}{\vdash_n \langle \Psi \rangle \text{let rec } x_1 = a_1 \text{ and } \dots \text{ } x_n = a_n \text{ in } a \downarrow_X \langle \Phi \rangle v} \\
\text{Tuple} \\
\frac{\vdash_n \langle \Psi \rangle a_1 \downarrow_X \langle \Psi_1 \rangle v_1 \quad \dots \quad \vdash_n \langle \Psi_{n-1} \rangle a_n \downarrow_X \langle \Psi_n \rangle v_n}{\vdash_n \langle \Psi \rangle (a_1, \dots, a_n) \downarrow_X \langle \Psi_n \rangle (v_1, \dots, v_n)} \\
\text{Projection} \\
\frac{\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle (\dots, v_n, \dots)}{\vdash_n \langle \Psi \rangle a.n \downarrow_X \langle \Phi \rangle v_n} \\
\text{Array} \\
\frac{\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Psi_0 \rangle \langle x \rangle \quad \vdash_n \langle \Psi_0 \rangle x \downarrow_X \langle \Psi_1 \rangle \{r, \dots\} \quad \vdash_n \langle \Psi_1 \rangle (r, \dots).n \downarrow_X \langle \Phi \rangle v}{\vdash_n \langle \Psi \rangle a!n \downarrow_X \langle \Phi \rangle v}
\end{array}$$

Fig. A.3. Natural semantics for λ_{need}

- The case in which a is let rec d' in a' . We have

$$x' = (\text{let rec } d' \text{ in } a') \text{ and } d^*[x, x'] \text{ and } d \vdash N[x] \xrightarrow[\text{need}]{} d' \text{ and } x' = a' \text{ and } d^*[x, x'] \text{ and } d \vdash N[x]$$

and

$$d \vdash \text{let rec } d' \text{ in } a' \xrightarrow[\text{need}]{} d \text{ and } d' \vdash a'$$

Thus this case follows from the induction hypothesis.

- The case in which a is $N'[x'']$ and d is $x'' = a'$ and d_0 . We have

$$\begin{array}{l}
\text{let rec } x'' = a' \text{ and } x' = N'[x''] \text{ and } d^*[x, x'] \text{ and } d_0 \text{ in } N[x] \\
\begin{array}{l}
\xrightarrow[\text{need}]{}^{n_0} \\
\text{let rec } x'' = v' \text{ and } x' = N'[x''] \text{ and } d^*[x, x'] \text{ and } d_1 \text{ in } N[x] \\
\text{let rec } x'' = v' \text{ and } x' = N'[v'] \text{ and } d^*[x, x'] \text{ and } d_1 \text{ in } N[x] \\
\xrightarrow[\text{need}]{}^{n_1} \\
\text{let rec } x' = v \text{ and } d^*[x, x'] \text{ and } d' \text{ in } N[x]
\end{array}
\end{array}$$

By ind. hyp. we have let rec d_0 in $a' \xrightarrow[\text{need}]{}^{n_0}$ let rec d_1 in v' . Thus we have

$$\begin{array}{l}
\text{let rec } x'' = a' \text{ and } d_0 \text{ in } N'[x''] \\
\begin{array}{l}
\xrightarrow[\text{need}]{}^{n_0} \\
\text{let rec } x'' = v' \text{ and } d_1 \text{ in } N'[x''] \quad \text{by ind. hyp.} \\
\text{let rec } x'' = v' \text{ and } d_1 \text{ in } N'[v'] \\
\xrightarrow[\text{need}]{}^{n_1} \\
\text{let rec } d' \text{ in } v \quad \text{by ind. hyp.} \quad \square
\end{array}
\end{array}$$

*Lemma A.6*If $d \vdash L[a] \xrightarrow[\text{need}]{}^n d' \vdash L[v]$, then $d \vdash a \xrightarrow[\text{need}]{}^{n'} d' \vdash v$ with $n' \leq n$.

Proof

By induction on the length of the reductions of $d \vdash L[a]$. \square

Lemma A.7

If $d \vdash a \xrightarrow[\text{need}]{} d' \vdash v$, then $d \vdash L[a] \xrightarrow[\text{need}]{} d' \vdash L[v]$

Proof

By induction on the length of the reductions of $d \vdash a$. \square

Proposition A.2

The following two conditions hold:

1. if $d \vdash a \xrightarrow[\text{need}]{} d' \vdash v$, then for any X such that $\text{dom}(d)$ and X are disjoint, there exists a configuration $d'' \vdash v'$ such that $d' \vdash v$ and $d'' \vdash v'$ belong to the same α -equivalence class and $\vdash_n \langle [d] \rangle a \downarrow_X \langle [d''] \rangle v'$;
2. if $\text{dom}(\Psi)$ and X are disjoint and $\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$, then for any d such that $[d] = \Psi$, $d \vdash a \xrightarrow[\text{need}]{} d' \vdash v$ where $[d'] = \Phi$.

Proof

1. Without loss of generality we assume $d' \vdash v$ and $d'' \vdash v'$ are syntactically identical. We prove by induction on the length of the reductions of $d \vdash a$ with case analysis on a . Below we prove two interesting cases.

- The case in which a is $a_0 a_1$. We have

$$\begin{array}{c} d \vdash a_0 a_1 \xrightarrow[\text{need}]{}^n d_0 \vdash (\lambda x.a') a_1 \xrightarrow[\text{need}]{} d_0 \vdash \text{let rec } x = a_1 \text{ in } a' \\ \xrightarrow[\text{need}]{} d_0 \text{ and } x = a_1 \vdash a' \xrightarrow[\text{need}]{} d' \vdash v \end{array}$$

By Lemma A.6, $d \vdash a_0 \xrightarrow[\text{need}]{}^{n'} d_0 \vdash \lambda x.a'$ with $n' \leq n$. By ind. hyp., $\vdash_n \langle [d] \rangle a_0 \downarrow_X \langle [d_0] \rangle \lambda x.a'$ and $\vdash_n \langle [d_0 \text{ and } x = a_1] \rangle a' \downarrow_X \langle [d'] \rangle v$. Thus we deduce $\vdash_n \langle [d] \rangle a_0 a_1 \downarrow_X \langle [d'] \rangle v$.

- The case in which a is x and d is $x = a'$ and d_0 . We have

$$x = a' \text{ and } d_0 \vdash x \xrightarrow[\text{need}]{}^n x = v \text{ and } d_1 \vdash x \xrightarrow[\text{need}]{} x = v \text{ and } d_1 \vdash v$$

By Lemma A.5, $d_0 \vdash a' \xrightarrow[\text{need}]{}^n d_1 \vdash v$. By ind. hyp., $\vdash_n \langle [d_0] \rangle a' \downarrow_{X \cup \{x\}} \langle [d_1] \rangle v$. Thus we deduce $\vdash_n \langle [d] \rangle x \downarrow_X \langle [x = v \text{ and } d_1] \rangle v$.

2. By induction on the derivation of $\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$ with case analysis on the last rule used. Below we prove two interesting cases.

- The case in which a is $a_0 a_1$. Suppose we deduce $\vdash_n \langle \Psi \rangle a_0 a_1 \downarrow_X \langle \Phi \rangle v$ from $\vdash_n \langle \Psi \rangle a_0 \downarrow_X \langle \Psi' \rangle \lambda x.a_2$ and $\vdash_n \langle \Psi' [x' \mapsto a_1] \rangle a_2[x'/x] \downarrow_X \langle \Phi \rangle v$. By ind. hyp. $d \vdash a_0 \xrightarrow[\text{need}]{} d' \vdash \lambda x.a_2$ with $[d'] = \Psi'$. Thus we have

$$\begin{array}{c} d \vdash a_0 a_1 \\ \xrightarrow[\text{need}]{} d' \vdash (\lambda x.a_2) a_1 \quad \text{by Lemma A.3} \\ \xrightarrow[\text{need}]{} d' \vdash \text{let rec } x' = a_1 \text{ in } a_2[x'/x] \\ \xrightarrow[\text{need}]{} d' \text{ and } x' = a_1 \vdash a_2[x'/x] \\ \xrightarrow[\text{need}]{} d'' \vdash v \quad \text{by ind. hyp.} \end{array}$$

$$\begin{array}{c}
\textit{Init} \\
\frac{\vdash_1 \langle \Psi \rangle a \downarrow_X \langle \Psi' \rangle \langle x \rangle \quad \vdash_1 \langle \Psi' \rangle x \downarrow_X \langle \Psi'' \rangle \{\{r, \dots\}\} \quad \vdash_1 \langle \Psi'' [x \mapsto \{r, \dots\}] \rangle (r, \dots).n \downarrow_X \langle \Phi \rangle v}{\vdash_1 \langle \Psi \rangle a!n \downarrow_X \langle \Phi \rangle v} \\
\textit{Array} \\
\frac{\vdash_1 \langle \Psi \rangle a \downarrow_X \langle \Psi' \rangle \langle x \rangle \quad \vdash_1 \langle \Psi' \rangle x \downarrow_X \langle \Psi'' \rangle \{r_1, \dots, r_n, r_{n+1}, \dots\} \quad \vdash_1 \langle \Psi'' \rangle (r_1, \dots, r_n).n \downarrow_X \langle \Phi \rangle v}{\vdash_1 \langle \Psi \rangle a!n \downarrow_X \langle \Phi \rangle v}
\end{array}$$

Fig. A.5. Natural semantics for λ_{lazy} *Proposition A.3*

If $\vdash Tr_V(E)_\emptyset \xrightarrow[\text{value}]{} D \vdash v$ then $\vdash Tr_N(E)_\emptyset \xrightarrow[\text{need}]{} d \vdash v'$.

Proof

By Corollary A.1, $\vdash_v \langle \rangle Tr_V(E)_\emptyset \downarrow_\emptyset \langle \Psi \rangle v$. By Lemma A.4, $\vdash_v \langle \rangle Tr_N(E)_\emptyset \downarrow_\emptyset \langle \Psi' \rangle v'$. By Lemma A.8, $\vdash_n \langle \rangle Tr_N(E)_\emptyset \downarrow \langle \Psi'' \rangle v'$. By Corollary A.2, $\vdash Tr_N(E)_\emptyset \xrightarrow[\text{need}]{} d \vdash v'$. \square

A.2 Call-by-need \subset Lazy-field

We define the natural semantics for λ_{lazy} by replacing *Array* rule in Figure A.3 with the two new rules given in Figure A.5. The following proposition is proved similarly to Proposition A.2.

Proposition A.4

The following two conditions hold:

1. if $\vdash a \xrightarrow[\text{lazy}]{} d \vdash v$, then there exists a configuration $d' \vdash v'$ such that $d \vdash v$ and $d' \vdash v'$ belong to the same α -equivalence class and $\vdash_1 \langle \rangle a \downarrow_\emptyset \langle [d'] \rangle v'$;
2. if $\vdash_1 \langle \rangle a \downarrow_\emptyset \langle \Psi \rangle v$, then $\vdash a \xrightarrow[\text{lazy}]{} d \vdash v$ where $[d] = \Psi$.

Next we want to relate the natural semantics for λ_{need} to that for λ_{lazy} . Inspired by a proof technique present in (Launchbury, 1993), we define an alternative natural semantics for λ_{need} by replacing *Variable* rule with

$$\frac{\vdash_n \langle \Psi |_{\bar{x}} \rangle \Psi(x) \downarrow_{X \cup \{x\}} \langle \Phi \rangle v \quad x \in \text{dom}(\Psi)}{\vdash_n \langle \Psi \rangle x \downarrow_X \langle \Phi[x \mapsto \Psi(x)] \rangle v}$$

and *Array* rule with

$$\frac{\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Psi_0 \rangle \langle x \rangle \quad \vdash_n \langle \Psi_0 \rangle x \downarrow_X \langle \Psi_1 \rangle \{r_1, \dots, r_n, r_{n+1}, \dots\} \quad \vdash_n \langle \Psi_1 \rangle (r_1, \dots, r_n).n \downarrow_X \langle \Phi \rangle v \quad \vdash_n \langle \Phi \rangle (r_{n+1}, \dots) \downarrow_X \langle \Phi' \rangle v'}{\vdash_n \langle \Psi \rangle a!n \downarrow_X \langle \Phi \rangle v}$$

Note that in the new *Array* rule bindings newly introduced while evaluating (r_{n+1}, \dots) are not in the scope of v or Φ ; this is why we can safely discard Φ' . The effect of the new *Variable* rule is to remove updating from the semantics. The effect of the new *Array* rule is to discard irrelevant bindings, which is made possible by the new *Variable* rule. Apart from these changes the two versions of the natural semantics are equivalent: the same heap/expression pairs reduce successfully. This is shown by induction.

Similarly, we define an alternative natural semantics for λ_{lazy} by replacing *Variable* rule with

$$\frac{\vdash_1 \langle \Psi|_{\bar{x}} \rangle \Psi(x) \downarrow_{X \cup \{x\}} \langle \Phi \rangle v \quad x \in \text{dom}(\Psi)}{\vdash_1 \langle \Psi \rangle x \downarrow_X \langle \Phi[x \mapsto \Psi(x)] \rangle v}$$

and *Init* rule with

$$\frac{\vdash_1 \langle \Psi \rangle a \downarrow_X \langle \Psi' \rangle \langle x \rangle \quad \vdash_1 \langle \Psi' \rangle x \downarrow_X \langle \Psi' \rangle \{\{r_1, \dots, r_n, r_{n+1}, \dots\}\} \\ \vdash_1 \langle \Psi'[x \mapsto \{r_1, \dots, r_n, r_{n+1}, \dots\}] \rangle (r_1, \dots, r_n).n \downarrow_X \langle \Phi \rangle v \quad \vdash_1 \langle \Phi \rangle (r_{n+1}, \dots) \downarrow_X \langle \Phi' \rangle v'}{\vdash_1 \langle \Psi \rangle a!n \downarrow_X \langle \Phi'|_{\text{dom}(\Phi)} \rangle v}$$

The notation $\Psi|_X$ denotes the restriction of Ψ to X . Again routine induction proves the same heap/expression pairs reduce successfully in the two versions of the natural semantics.

We write $\Psi \vdash a \prec \Psi' \vdash a'$ when $\Psi \vdash a$ is obtained from $\Psi' \vdash a'$ by replacing any occurrence of $\{\{r, \dots\}\}$ with $\{r, \dots\}$. Now in terms of the alternative versions of the natural semantics it is easy to prove λ_{lazy} is more successful than λ_{need} .

Lemma A.9

If $\text{dom}(\Psi)$ and X are disjoint and $\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$, then for any $\Psi' \vdash a'$ such that $\Psi \vdash a \prec \Psi' \vdash a'$, $\vdash_1 \langle \Psi' \rangle a \downarrow_X \langle \Phi' \rangle v'$ and $\Phi \vdash v \prec \Phi' \vdash v'$.

Proof

By induction on the derivation of $\vdash_n \langle \Psi \rangle a \downarrow_X \langle \Phi \rangle v$. \square

We are ready to prove our second result.

Proposition A.5

If $\text{Tr}_N(E)_\emptyset \xrightarrow{\text{need}} d \vdash v$ then $\text{Tr}_L(E)_\emptyset \xrightarrow{\text{lazy}} d' \vdash v'$.

Proof

By Corollary A.2, $\vdash_n \langle \text{Tr}_N(E)_\emptyset \rangle \downarrow_\emptyset \langle \Psi \rangle v$. By Lemma A.9, $\vdash_1 \langle \text{Tr}_L(E)_\emptyset \rangle \downarrow_\emptyset \langle \Psi' \rangle v'$. By Proposition A.4, $\vdash \text{Tr}_L(E)_\emptyset \xrightarrow{\text{lazy}} d' \vdash v'$. \square

