Directly Reflective Meta-Programming *

Aaron Stump
Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
stump@cse.wustl.edu

Abstract

Existing meta-programming languages operate on encodings of programs as data. This paper presents a new meta-programming language, based on an untyped lambda calculus, in which structurally reflective programming is supported directly, without any encoding. The language features call-by-value and call-by-name lambda abstractions, as well as novel reflective features enabling the intensional manipulation of arbitrary program terms. The language is scope safe, in the sense that variables can neither be captured nor escape their scopes. The expressiveness of the language is demonstrated by showing how to implement quotation and evaluation operations, as proposed by Wand. The language's utility for meta-programming is further demonstrated through additional representative examples. A prototype implementation is described and evaluated.

Keywords: lambda calculus, reflection, meta-programming, Church encoding, Mogensen-Scott encoding, Wand-style fexprs, alpha equivalence, language implementation.

1 Introduction

This paper presents a new meta-programming language called Archon, in which programs can operate directly on other programs as data. Existing meta-programming languages suffer from defects such as encoding programs at too low a level of abstraction, or in a way that bloats the encoded program terms. Other issues include the possibility for variables either to be captured or escape their static scopes. Archon rectifies these defects by trading the complexity of the reflective encoding for additional programming constructs. We adopt

*This work is supported by funding from the U.S. National Science Foundation under award CCF-0448275. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

the identity function as our encoding, and add new reflective constructs to traverse (unencoded) program terms. These results are achieved in a *scope-safe* way: variables may neither be captured by substitution nor escape their scopes. This work is the crucial first part of the ARCHON project, which aims to unite reflective programming and reflective theorem proving.

In order to situate the present work with respect to the existing literature, we first survey work on several kinds of reflection in Computer Science (Section 2). We then further motivate the work by briefly discussing the role of meta-programming in the ARCHON project (Section 3). In Sections 4 through 6, we consider in detail related work by Wand [50]. We will see how Wand's language achieves desirable properties which other reflective programming languages lack, but at the same time fails to be suitable for general meta-programming. This requires a brief review of the Mogensen-Scott encoding used in Wand's language (Section 5). The Archon language itself is presented starting in Section 7. An outline for the sections of the paper presenting ARCHON may be found in Section 3.3.

2 On Reflection

Abstractly speaking, reflection is concerned with enabling entities at one linguistic level (called the object level) to interact with entities at a higher level (the meta-level). This is typically done via some kind of encoding of the meta-level entities in the object language. This abstract description is specialized by stating what the languages are, what kind of entities are involved, and what sort of interaction is allowed. A crucial point is how the meta-level entities are encoded in the object language. The rest of this section categorizes related work on reflection in Computer Science according to these criteria.

Note that in related work on logical frameworks, object languages (such as logics) are seen as being embedded in meta-languages (logical frameworks) [33]. This determination of what is object-level and what is meta-level differs from that of our abstract description of reflection. The difference in viewpoint arises from a difference in emphasis. Logical frameworks are typically concerned with representing other deductive systems. In reflection, the emphasis is on the ability of an object-level language to encode entities from its own meta-level.

2.1 Reflective Programming Languages

In programming languages, the object-level language is a programming language, and the entities there are programs. Following Ferber, we may then informally distinguish structural reflection and computational reflection [9]. In structural reflection, the meta-level entities are determined in some way by program source code, and have a more static character. In computational reflection, the meta-level entities are constituents of the running program's execution environment, and thus are more dynamic. This paper is concerned with meta-programming, which is a form of structural reflection. We first briefly survey

other forms of reflective programming.

Much early interest in reflective programming languages centered around language extension via computational reflection [18, 41]. Control operators such as SCHEME's `call/cc` are also computationally reflective, since they expose an aspect of the program's execution environment, namely its continuation, to the program itself. Object-oriented programming languages like SMALLTALK and JAVA include computationally reflective features, for example the ability, in JAVA, of a running program to replace its class loader [1, 13].

Object-oriented programming languages typically also provide certain structurally reflective features, mediated by abstracted representations of classes. For example, JAVA's reflection mechanism makes essentially just the names and types of object and class components available to running object-level programs. Interaction is restricted mostly to inspection, although certain simple updating operations like setting the values of object fields are allowed. Object-oriented languages like SMALLTALK allow much more liberal forms of interaction. Runtime type information in C++ may be considered another example [43]. Similar functionality is provided in functional languages by intensional polymorphism, where code can dispatch at runtime on a term's type [6].

2.1.1 Meta-Programming

Meta-programming languages are based on a shallow encoding of program texts. In general, the kind of interaction allowed in meta-programming is to perform arbitrary computation of output values, including program texts, from input program texts. Note that this excludes meta-programs from modifying running code. Self-modifying code can arguably be viewed as both structurally and computationally reflective, since the running code can be viewed as forming part of the program's execution environment.

With meta-programming, object-level programs manipulate encoded program texts as data. Languages in the LISP family, such as modern dialects COMMON LISP, EMACS LISP, and SCHEME, exemplify this kind of reflective programming [42, 27, 21, 28]. In these languages, it is possible to view any program as a piece of data, and any piece of data as a program (albeit possibly one which cannot execute without error, as in trying to call something, such as a numeric literal, which is not a function). Quotation (`quote`) encodes program expressions as data, specifically nested lists of symbols. As lists, program expressions such as β-redexes, which would otherwise evaluate, are frozen. They may be inspected and decomposed using LISP's list constructs. For example, one may obtain the applicand `(lambda (x) x)` from the quoted application `'((lambda (x) x) 3)` by applying `car`. Similarly, evaluation (`eval`) transforms data into programs. Frozen terms are, so to speak, thawed, and reduction of their redexes takes place again. Note that since LISP languages typically include functions for obtaining the name of a symbol as a string (e.g., `symbol->string` in SCHEME), detailed lexical information about variables is available to programs in these languages. So the reflective encoding provided by these languages is at a rather low level of abstraction.

Among the research problems related to meta-programming that have been considered in the literature are static typing of meta-programs, and how meta-programs deal with the scoping of variables. It is no accident that the LISP languages, which best exemplify meta-programming, are not statically typed. The soft type system of Wright and Cartwright for SCHEME covers the features of SCHEME version R4RS, which does not include the meta-programming feature **eval** [52]. Static typing of general meta-programs appears difficult, and the existing literature mostly considers static typing for a weaker form of meta-programming called staged computation, used also for macro systems [22, 3, 32, 10, 44]. In staged computation, programs may generate programs as data, but not inspect or decompose them. Some work on static typing of more general meta-programs has been done, but a full account is currently lacking [31]. Extensible languages support the addition of new syntax to a host language, as well as accompanying semantic analyses for more informative error reporting [11].

2.1.2 Variables in Meta-Programming

A meta-programming language is *scope safe* (or *hygienic*) iff variables may not be captured or escape their scopes during computation. Dynamic variables in EMACS LISP and COMMON LISP are a good example of a violation of scope safety [30, 24]. SCHEME R5RS's macro language is designed to be scope safe [21]. Other constructs in SCHEME R5RS, however, enable violation of scope safety, even though the language does not have dynamic variables. For a violation of scope safety in spirit, though not technically, we have that `(caddr '(lambda (x) x))` evaluates to `x`. According to the R5RS language definition, `'(lambda (x) x)` is a literal expression, and hence the occurrences of `x` in it are not variables at all, but just (unscoped) literal data. So in this example, a variable has been created (namely, the resulting unquoted `x`), but not by means of removing it from its scope. Using R5RS's macro system, however, this example may be modified to give a true violation of scope safety. The following macro extracts the variable `x` from its scope, by transforming the binding `lambda` expression into a piece of literal data, and then extracting and evaluating the quoted variable. [1]

```
(define-syntax exv
(syntax-rules () ((exv arg) (car (car (cdr (car (cdr ''arg))))))))

(exv (lambda (x) x))
```

Representation of variables is another issue considered in the literature on meta-programming. Higher-order abstract syntax (HOAS), where variables in

[1] Please note that in the published version of this paper in HOSC, I incorrectly stated that this sort of example could be done using quasiquotation. The example I gave there does not work in R5RS Scheme, because lambda abstractions evaluate to procedures, from which one cannot extract components, even when quoted. The current example using macros does work in R5RS SCHEME. Thanks to John Clements for pointing out my earlier error, and for Robby Findler, Matthew Flatt, and Matthias Felleisen for an engaging email discussion about the meaning of this example.

reflected programs are represented directly using variables in meta-programs (as opposed to, for instance, de Bruijn indices or other numbering schemes) greatly eases the burden of operating on reflected program texts, but poses challenges for static typing [51, 38, 34]. A great variety of other representation techniques has been considered, for example in the solutions to the POPLmark challenge [2]. The details of these are not relevant to the current work, other than to note: while HOAS can introduce technical complications, particularly for typed languages, it is widely acknowledged to be the most concise and convenient variable representation technique.

2.2 Reflective Theorem Proving

Another kind of reflection is studied in Mathematical Logic and Automated Theorem Proving [17, 14, 5, 8]. There, the object language is a logical theory. The meta-level language is the language in which this theory is defined. Object language entities such as formulas and proofs may refer to similar entities at the meta-level, via an encoding of meta-level syntax and semantics in the theory. The most famous example of this is Gödel's use of reflection in the development of his incompleteness theorems. He encodes formulas and proofs as natural numbers, with the provability predicate then expressed via a logical formula. In this way, one can prove in the theory various formulas of the form *Provable* $\ulcorner\phi\urcorner$, where ϕ is a formula provable in the theory, and $\ulcorner\phi\urcorner$ is its encoding as a number. Gödel's results establish fundamental limits on logical reflection: consistent theories of suitable strength cannot reflect meta-theoretic proofs of their own consistency. Hence, they can express only proper fragments of their meta-theories.

The principle use of reflection in automated theorem proving is to enable safe extension of a trusted core prover by more efficient, untrusted reasoning principles. An early example is the work by Davis and Schwartz on using reflection to allow safe extension of an arithmetic theory [8]. This requires encoding formulas and proofs as numbers, as in Gödel's work. Modern theorem provers like Coq, HOL Light, and others support richer notions of data type than just natural numbers [15, 45]. Such provers can support reflective activities with a lower encoding penalty than that incurred by encoding formulas and proofs as natural numbers, since their datatypes enable much more natural encodings. Nevertheless, there is still an encoding penalty to pay. Users wishing to do reflective proofs must work with some class of encoded objects, although some systems allow shallower encodings than others. In Coq, for example, reflective tactics typically must encode the syntactic entities such as formulas upon which they will operate (cf. [37]). They typically do not need to encode the code for the tactic itself, however. The mapping from Coq formulas to encoded formulas is implemented in Coq's tactic language (Ltac), and is not susceptible to formal reasoning, or even type checking. In provers like HOL Light, tactics written in ML are type checked, but are otherwise outside the direct scope of the prover's proof system (although see [16]). In ACL2, meta-functions operating on non-trivially encoded terms may be added to the trusted core once proved

correct [17]. Non-trivially encoded terms are also used heavily in the Maude term rewriting system [4].

3 The Archon Project

As stated above, the work reported in this paper is the first part of a project called ARCHON, which aims to unite reflective programming and reflective theorem proving. The two main goals of ARCHON are formal verification of structurally reflective programs and directly reflective theorem proving. Achieving these goals is both work in progress and beyond the scope of the current paper. Nevertheless, we consider them briefly, to help motivate the design of the ARCHON programming language.

3.1 Verification of Meta-Programs

Formal verification of structurally reflective programs written in existing languages is a daunting prospect. Consider the LISP family of languages. The imperative nature of these languages already greatly increases the difficulty of formal reasoning, since program state (the environment, in LISP terminology) must be handled explicitly. One could work in just a pure subset, as done in ACL2, but then one does not support the full language, which may be a drawback for reasoning about existing programs [20]. Furthermore, the level of abstraction of program texts is quite low. As noted above, the encoding of program texts as data in languages like SCHEME leaves some low-level details (e.g., the string representations of variables) of program texts unabstracted. The lack of scope safety poses another challenge, since in general, formal reasoning then requires reasoning about the scope of variables.

Verification of meta-programs is motivated by the observation that there are meta-programs such as compilers which are important targets for verification (see, e.g., [25, 26]). There is also a deeper motivation. Static type systems and other analyses can greatly aid formal verification. If a program is typable in some non-trivial type system, there is a non-trivial theorem that holds about it, which can be used as a lemma during verification of more general properties (see, e.g., [48]). To obtain such lemmas in a machine-checked setting, it is necessary to reason formally about the behavior of such analyses. Since these analyses are meta-programs (computing a type, for example, from a program text encoded as a piece of data in the language), formal verification of meta-programs is needed.

3.2 Directly Reflective Theorem Proving

The second goal of ARCHON is directly reflective theorem proving. Recall that existing reflective provers require an encoding of at least some meta-level objects like formulas and proofs. Directly reflective theorem proving is reflective proving without any encoding. Instead of operating on encodings of theorem prover structures like formulas or terms, programs like tautology checkers may operate

on such structures directly. Safe extension of the prover's trusted core then becomes more feasible, since there is no non-trivial encoding, as implemented by the Ltac code used in Coq, which falls outside the realm of the prover's reasoning system. Reasoning about reflective operations is also easier, since it is not necessary to account for a non-trivial encoding.

3.3 The Problem and Further Outline

The crucial first step to meeting the above goals is to devise a programming language with direct structural reflection, in which programs can operate directly on other programs, at an appropriate level of abstraction. The level of abstraction of the reflective encoding in LISP is too low, because it results in a contextual equivalence that is too fine. Recall that two program terms are contextually equivalent iff no context can distinguish them by behaving in observationally different ways (e.g., returning true for the one term and false for the other). Programs in LISP languages can distinguish between terms which are α-equivalent (equivalent up to capture-avoiding renaming of bound variables). For example, building on the code given in Section 2.1.2 for extracting the bound variable from a lambda abstraction, we can easily write a macro which evaluates to `#t` if it is applied to `(lambda (x) x)` and `(lambda (x) x)`, and `#f` if it applied to `(lambda (x) x)` and `(lambda (y) y)`:

```
(define-syntax complam
  (syntax-rules () ((complam arg1 arg2) (eqv? (exv arg1) (exv arg2)))))

(complam (lambda (x) x) (lambda (x) x))

(complam (lambda (x) x) (lambda (y) y))
```

The first expression evaluates to `#t`, and the second to `#f`. [2]

With a good higher-level encoding scheme, α-equivalence of program texts should certainly imply contextual equivalence of their encodings, and the language should be scope safe. Going further, we can observe that for a general meta-programming language, α-equivalence of programs should coincide with contextual equivalence of their encodings. For this implies that meta-programs have an intensional enough view of encoded programs to be able to make all the distinctions we consider relevant at the meta-level. For ease of formal reasoning, the language should be pure, with no imperative features. Finally, it is desirable for the encoding of programs to use higher-order abstract syntax (HOAS) where applicable, since this is by far the most convenient variable representation technique (as discussed in Section 2.1.2).

An important step towards these goals has been taken by Wand, who achieves a minimalistic pure programming language with a reflective encoding meeting the above criteria [50]. Unfortunately, Wand's language is not suitable for general meta-programming (though it does not need to be, for the theoretical results

[2] This example has been added here, and is not contained in the published HOSC version.

of his paper). This matter is discussed in detail in the next three sections. The syntax and operational semantics of ARCHON are presented next (Section 7), as well as examples demonstrating ARCHON's utility for meta-programming (Section 8). Section 8.4 shows how to implement the reflective primitives of Wand's language in ARCHON, thus further establishing the new language's expressiveness. A prototype implementation of ARCHON is discussed and evaluated in Section 9.

4 Wand's System

Wand defines operations **fexpr** (for quotation) and **eval** which map program terms to their HOAS-based Mogensen-Scott encodings (reviewed below) and back [50]. He proves that α-equivalence of programs coincides with contextual equivalence of their encodings. This achieves the goal mentioned above for a higher-level encoding of terms. The motivation for Wand's work is, in a sense, a negative one: he wishes to point out the difficulties which reflective capabilities raise for traditional approaches to applications like compilation. These rely on coarser notions of contextual equivalence than just α-equivalence, for example to do source-to-source optimizations. Such optimizations are intended, of course, to result in terms which are not α-equivalent to the unoptimized terms. It is not Wand's aim to give an account of general meta-programming in his proposed system, since for his purposes, it is enough to establish the *triviality* of contextual equivalence (namely, that it coincides with α-equivalence). Indeed, Wand's system cannot be used for general meta-programming, at least not in any obvious way. We explain why (in Section 6 below), after first reviewing his system.

The syntax for Wand's system is the following (cf. [50, pages 90–91]):

$$T ::= x \mid \lambda x.T \mid T\ T \mid \mathbf{fexpr}\ T \mid \mathbf{eval}\ T$$

Here and below we use standard abbreviations from lambda calculus including left associativity of application, abbreviated notation for consecutively nested lambda abstractions, and lambda abstraction with scope extending as far as syntactically possible to the right. To define the operational semantics of this language, we first designate reduction contexts R and values V, as follows:

$$\begin{array}{lcl} R & ::= & [] \mid (R\ M) \mid ((\lambda x.M)\ R) \mid (\mathbf{fexpr}\ R) \mid (\mathbf{eval}\ R) \\ V & ::= & \lambda x.M \mid (\mathbf{fexpr}\ V) \end{array}$$

We use a standard notion of reduction contexts R. Recall that these contain exactly one occurrence of the hole $[]$, which may be filled with a term M with the notation $R[M]$. Reduction contexts indicate, in a compact way, the evaluation order of the language. The operational semantics is now defined by the following reduction rules:

$$\begin{array}{lcl} R[((\lambda x.M)\ V)] & \rightarrow & R[[V/x]M] \\ R[((\mathbf{fexpr}\ V)\ M)] & \rightarrow & R[(V\ \ulcorner M \urcorner)] \\ R[(\mathbf{eval}\ \ulcorner M \urcorner)] & \rightarrow & R[M] \end{array}$$

The second and third reduction rules rely on an encoding function $\ulcorner \cdot \urcorner$, defined just below. We first observe that intuitively, ((**fexpr** V) M) calls V on the encoding of M, where M need not be a value. Indeed, the whole point is that arbitrary program terms M, including ones which would otherwise reduce, are frozen by **fexpr** and presented to V. Frozen terms may then be thawed using **eval**. As Wand explains in an endnote, it is necessary to include **eval** as a primitive in the language, due to the way the encoding works. We will return to this point below. The encoding is defined as follows:

$$\begin{array}{lcl}
\ulcorner x \urcorner & = & \lambda a\ b\ c\ d\ e.a\ x \\
\ulcorner M\ N \urcorner & = & \lambda a\ b\ c\ d\ e.b\ \ulcorner M \urcorner\ \ulcorner N \urcorner \\
\ulcorner \lambda x.M \urcorner & = & \lambda a\ b\ c\ d\ e.c\ \lambda x.\ulcorner M \urcorner \\
\ulcorner (\textbf{fexpr}\ M) \urcorner & = & \lambda a\ b\ c\ d\ e.d\ \ulcorner M \urcorner \\
\ulcorner (\textbf{eval}\ M) \urcorner & = & \lambda a\ b\ c\ d\ e.e\ \ulcorner M \urcorner
\end{array}$$

Wand calls this encoding a Mogensen-Scott encoding, since Mogensen defines an encoding with the essential features of the above. Mogensen's encoding extends an encoding attributed to Dana Scott with the use of HOAS (higher-order abstract syntax) in the clause for encoding lambda abstractions [29]. Since these encodings appear to be less well known than, say, the Church encoding, and since they play a crucial role both in understanding the limitations of Wand's system and the contribution of the present work, we pause now to consider them (readers familiar with these encodings may, of course, wish to skip the following Section).

5 The Mogensen-Scott Encoding

There are a variety of ways to encode inductive datatypes using lambda calculus terms (see, e.g., [49] or [7, Chapter 13]). Informally, the basic problem is to define the fundamental operations of an inductive datatype as pure lambda terms. So constructors (e.g., in SCHEME, `'()` and `cons`), testers (e.g., `null?`) and selectors (e.g., `car` and `cdr`) are to be defined in pure lambda calculus, instead of taken as primitive, as they are in LISP languages.

5.1 The Church Encoding

Before considering the Scott encoding and its extension by Mogensen, let us recall the Church encoding, which may be more familiar to some readers. We encode the operations of an inductive datatype in the following way. Suppose the datatype has n constructors, where the i'th constructor C_i has arity $a(i)$. We encode C_i by the following lambda term, where we write $\bar{c}$ for $c_1\ \ldots\ c_n$:

$$\lambda x_1\ \ldots\ x_{a(i)}.\lambda c_1\ \ldots\ c_n.c_i\ (x_1\ \bar{c})\ \ldots (x_{a(i)}\ \bar{c})$$

This term takes in the $a(i)$ arguments to constructor C_i as $x_1, \ldots,\ x_{a(i)}$. It then returns a lambda abstraction (let us call it M) which accepts n arguments

$c_1, \ldots, c_n$, one for each constructor of the datatype. These arguments are iterators, which will be applied according to the structure of the data. This M then applies the i'th iterator to, not literally the input arguments $x_1, \ldots, x_{a(i)}$, but rather to what we might think of as the *transposition* of those inputs to the iterators $c_1, \ldots, c_n$. That is, each Church-encoded piece of data, such as the arguments $x_1, \ldots, x_{a(i)}$, begins by accepting n iterators to apply according to the structure of the data. Our term M transposes the arguments to the constructor, so that they apply the given iterators $c_1, \ldots, c_n$, rather than whatever other iterators they would have taken in.

A simple example helps demonstrate this encoding. Consider the inductive datatype of the natural numbers in unary notation. There are two constructors, S (successor) and Z (zero), where the former has arity 1 and the latter has arity 0. They are Church encoded like this:

$$\begin{array}{lcl} S & := & \lambda x_1.\lambda s\ z.s\ (x_1\ s\ z) \\ Z & := & \lambda s\ z.z \end{array}$$

With this encoding, the first few numerals are defined as follows, and satisfy the stated β-equivalences:

$$\begin{array}{lcll} 0 & := & Z & =_\beta \quad \lambda s\ z.z \\ 1 & := & S\ Z & =_\beta \quad \lambda s\ z.s\ z \\ 2 & := & S\ (S\ Z) & =_\beta \quad \lambda s\ z.s\ (s\ z) \\ 3 & := & S\ (S\ (S\ Z)) & =_\beta \quad \lambda s\ z.s\ (s\ (s\ z)) \end{array}$$

5.2 The Scott Encoding

As the source for the Scott encoding, the Computer Science literature depends on the following citation from Curry et al.: "Dana Scott, A system of functional abstraction. Lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University, September 1963, furnished by author in 1968" [7, page 504]. This work appears not to have been subsequently published. Disregarding this bibliographic issue, we consider the Scott encoding as follows. Given an inductive datatype presented as at the start of the previous section, we encode constructor C_i by the following lambda term:

$$\lambda x_1\ \ldots\ x_{a(i)}.\lambda c_1\ \ldots\ c_n.c_i\ x_1\ \ldots\ x_{a(i)}$$

The crucial difference between this term and the corresponding term from the Church encoding is that here, the inputs $x_1, \ldots, x_{a(i)}$ are not "transposed" to use the iterators $c_1, \ldots, c_n$. This means that lambda abstractions occur all throughout the encoding of a compound piece of data, unlike in Church-encoded data, where such lambda abstractions occur only at the very top of the term. For the natural number datatype, the definition specializes as follows:

$$\begin{array}{lcl} S & := & \lambda x_1.\lambda s\ z.s\ x_1 \\ Z & := & \lambda s\ z.z \end{array}$$

With this encoding, we may obtain the first few numerals by call-by-value reduction (denoted here by $\Downarrow_{cbv}$) using S and Z:

$$\begin{array}{lcll} 0 & := & Z & \Downarrow_{cbv} \quad \lambda s\ z.z \\ 1 & := & S\ Z & \Downarrow_{cbv} \quad \lambda s\ z.s\ 0 \\ 2 & := & S\ (S\ Z) & \Downarrow_{cbv} \quad \lambda s\ z.s\ 1 \\ 3 & := & S\ (S\ (S\ Z)) & \Downarrow_{cbv} \quad \lambda s\ z.s\ 2 \end{array}$$

So for example, the Scott encoding of 2 may be written fully as

$$\lambda s\ z.s\ (\lambda s\ z.s\ (\lambda s\ z.z))$$

In contrast, the Church encoding of 2 is β-equivalent to

$$\lambda s\ z.s\ (s\ z)$$

Where Church-encoded data accept iterators to apply according to the structure of the term, Scott-encoded data may be viewed as accepting continuations. The appropriate continuation will be called with the immediate subdata of a given piece of data. Thus, the Scott encoding encodes data as case statements, instead of as iterations. For example, we can define the predecessor function for Scott-encoded numerals as follows (where zero is taken to be its own predecessor):

$$\lambda x.x\ (\lambda p.p)\ 0$$

A Scott-encoded numeral x is accepted as input by this lambda term. This x is then immediately applied to two continuations. The first continuation will be used when x is a successor number. The second will be used when x is zero. Whichever continuation is called, it will be called with the immediate subdata of x. If x is zero, it has no subdata, so the continuation 0 is called with no arguments (i.e., simply returned). If x is a successor number, its immediate subdata is its predecessor p. This will be given as the input to the continuation $\lambda p.p$, which simply returns it, thus producing the predecessor.

5.3 Comparison of the Church and Scott Encodings

The Church encoding seems to be more familiar to computer scientists than the Scott encoding. For example, the Church encoding is presented in detail in standard programming languages textbooks like Pierce's, while the Scott encoding is not mentioned [35]. The Church encoding does have some advantages over the Scott encoding. As shown just above, Church-encoded data are generally more concise than Scott-encoded data (though not asymptotically so). The main advantage and perhaps the reason they are better known is that Church-encoded data and many common operations on them are typable in System F [12]. Thanks to strong normalization of System F, the large class of programs expressible as well-typed operations on Church-encoded data (including non-trivial examples like list-sorting functions) are statically guaranteed to terminate.

Scott-encoded data, in contrast, is not in any obvious way typable in pure System F. They appear to require both universal and recursive types for typability. Since strong normalization fails in the presence of recursive types, we cannot establish termination of programs manipulating Scott-encoded data just by static typing in a traditional type system. Nevertheless, the Scott encoding enjoys two critical advantages over the Church encoding, which should make them preferable for general programming. First, constructor terms (like $S\ (S\ Z)$) evaluate to their intended encodings in call-by-value lambda calculus. This is not the case with the Church encoding, where a constructor term like $(S\ Z)$ evaluates to a value in one step as follows in the call-by-value strategy:

$$(S\ Z) \quad \rightarrow \quad \lambda s\ z.s\ (Z\ s\ z)$$

In order to obtain the intended encoding $\lambda s\ z.s\ z$, two β-reductions would need to be performed beneath a lambda abstraction, which is not allowed with the call-by-value or call-by-name strategy.

The second advantage of the Scott encoding is that constant-time selector functions are easily definable, as discussed above. With the Church encoding, in contrast, known implementations of predecessor are rather complicated, and run in time linear in the size of the input numeral.

5.4 Mogensen's Addition to the Scott Encoding

Mogensen extends the idea of the Scott encoding using HOAS to encode untyped lambda terms [29]:

$$\begin{array}{lcl} \ulcorner x \urcorner & = & \lambda a\ b\ c.a\ x \\ \ulcorner M\ N \urcorner & = & \lambda a\ b\ c.b\ \ulcorner M \urcorner\ \ulcorner N \urcorner \\ \ulcorner \lambda x.M \urcorner & = & \lambda a\ b\ c.c\ \lambda x.\ulcorner M \urcorner \end{array}$$

The crucial difference from a Scott encoding as defined in the previous section is in the third clause of this definition. Lambda abstractions are encoded by using a lambda abstraction for the bound variable (in $\lambda x.\ulcorner M \urcorner$). So variables are represented using variables. We will not review here the general advantages of this representation (see the works cited in Section 2.1.2 above), but consider next the specific effect of this representation in Wand's system.

6 Meta-Programming in Wand's System

The use of the Mogensen-Scott encoding of lambda terms in Wand's system results in a HOAS-based encoding of lambda terms which satisfies the goals stated in Section 3.3 above [50]. He thus accomplishes his stated objective of showing the triviality of contextual equivalence, and to highlight the challenges this poses for applications like source-to-source optimization.

Certain kinds of meta-programs can be written in Wand's system, as demonstrated below (Section 8.1). But Wand's language is inadequate for meta-programs which generate code, or even encodings of code, as indeed hinted

by Endnote 2 of his paper. Wand there explains that he was led to include **eval** as a primitive operation in the language because it is not in any obvious way definable, given the Mogensen-Scott encoding. This limitation is not unique to **eval**. There is no obvious way to define any non-trivial code-generating meta-programs (like **eval**) in Wand's system. The problem is that there is no way to compute beneath a lambda abstraction. For example, as mentioned in Wand's endnote, suppose we try to implement **eval**. Mogensen gives code for **eval** that works, but only if arbitrary β-reduction is used. In the call-by-value setting of Wand's language, we could try something like the following:

$$\begin{array}{rll} \textbf{eval} & := & \mathit{fix}\ \lambda \mathit{eval}\ m. \quad m\ (\lambda x.x) \\ & & \qquad\qquad\quad (\lambda m\ n.(\mathit{eval}\ m)\ (\mathit{eval}\ n)) \\ & & \qquad\qquad\quad (\lambda m.\lambda x.\mathit{eval}\ (m\ x)) \end{array}$$

where *fix* is a standard call-by-value fixpoint operator, which may be defined as:

$$\mathit{fix} \quad := \quad \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y))$$

The problem, noted by Wand, is in the third line of the term for **eval** above, where we try to evaluate a lambda abstraction m by evaluating, essentially, its body, beneath a new lambda abstraction ("$\lambda x.\mathit{eval}\ (m\ x)$"). With call-by-value or call-by-name evaluation, the redex $\mathit{eval}\,(m\ x)$ cannot be evaluated here, since it is buried beneath a lambda abstraction. This same problem afflicts any non-trivial meta-program generating code or encodings of code. Meta-programs which need to recursively compute the body of a lambda abstraction cannot be implemented (in any obvious way). For to do so as in existing approaches, they would either compute the body using a free variable which would then be captured, or else they would compute beneath the lambda abstraction. The former option is disallowed by scope safety, and the latter by the evaluation strategy. Hence, this language, despite the desirable properties of its reflective encoding, is not suitable for general meta-programming. We can conclude additionally that it is not sufficient, for the purposes of general meta-programming, to have α-equivalence of programs coincide with contextual equivalence of their encodings.

7 The Archon Programming Language

We turn now to the definition of the Archon programming language, which solves the problems discussed above with existing structurally reflective languages. Archon is pure and scope safe. We have α-equivalence of programs coinciding with contextual equivalence of their encodings, which use HOAS. These qualities are shared with Wand's language. Like Wand's system, we are striving in Archon for something like minimality or independence: no language construct is directly definable using the others, in the sense that any use of one operator can be replaced by some constant-time computable term with the same extensional behavior. If scope safety is a form of soundness, we are

also interested in completeness. We might take completeness in this context to mean that every scope-safe function implementable in an untyped lambda calculus with more general, non-scope-safe meta-programming constructs (e.g., LISP) can be implemented in the scope-safe language. While independence and completeness are both desired properties of the language, proving or disproving these must remain to future work.

ARCHON is suitable for general meta-programming, as demonstrated by several examples in Section 8 below. The shortcoming of Wand's language, namely the inability to compute beneath lambda binders, is repaired in ARCHON. Orthogonally, ARCHON opts to place the burden of reflection on the programming language, rather than the reflective encoding. ARCHON adopts the identity function for the encoding, at the cost of introducing new reflective programming constructs for operating on program terms. Hence, ARCHON meta-programs operate directly on raw program terms, without any encoding. This shift of the burden of reflection from the encoding to the language is justified as follows. First, the language must already be extended in some way to allow computation beneath lambda abstractions. Once we have begun extending the language to allow for reflection, we can achieve a simpler language by shifting all reflective burden from the encoding to the language. This simplification has practical benefits. In reasoning (either formally or informally) about the behavior of meta-programs, we can consider the manipulated programs directly, in their natural form as programs of the language, and not via a non-trivial encoding. In contrast, with Wand's language, we must contend not only with additional reflective language constructs, but with the overhead of the Mogensen-Scott encoding.

One quantitative measure of the simplicity of the language design is its *locality*, in the following sense. Wand's reduction rules for **fexpr** and **eval** require recursive meta-level computation to apply or undo the Mogensen-Scott encoding, for a single reduction step. In contrast, ARCHON's reflective constructs require only constant-time meta-level computation for a single reduction step. A single step of evaluation, without any recursive meta-level evaluation, suffices to eliminate any use of ARCHON's reflective constructs, in favor of constructs of untyped lambda calculus.

7.1 Syntax

The syntax of ARCHON terms T (we also write M, N, R) appears in Figure 1. We write x (as well as y and z) only for variables, drawn from some countably infinite set. We then have call-by-value λ-abstractions and call-by-name $\bar{\lambda}$-abstractions, and applications (written using juxtaposition as usual). When a call-by-value lambda abstraction appears in an application, its argument must be evaluated before the application can be β-reduced. In contrast, arguments are passed to call-by-name lambda abstractions unevaluated. This is useful in meta-programming for passing around raw program terms, which would otherwise evaluate.

For meta-programming purposes, it seems required in practice to allow com-

$$
\begin{array}{rcl}
T & ::= & x \mid \lambda x.T \mid \bar{\lambda} x.T \mid T\ T \mid \textbf{open}\ T\ T \mid \textbf{vcomp}\ T\ T \\
& & \mid \textbf{swap}\ T \mid T : T\ T\ T\ T\ T\ T\ T
\end{array}
$$

Figure 1: The Syntax of ARCHO Terms

putation with open terms (i.e., terms which contain free variables). So in ARCHON, (free) variables evaluate to themselves. In LISP languages, in contrast, evaluation of a free variable is not allowed (and typically causes evaluation to abort with an error). Applications like $a\ b$, where a and b are free variables, also evaluate to themselves. Indeed, the set of values V in ARCHON is somewhat more complex than usual in call-by-value or call-by-name lambda calculus:

$$
\begin{array}{rcl}
V & ::= & \lambda x.T \mid \bar{\lambda} x.T \mid A \\
A & ::= & x \mid A\ V
\end{array}
$$

The remaining four constructs of the language are the reflective constructs, informally explained next (the formal operational semantics is given in the next Section). We rely on no parsing conventions for ARCHON below, except the (standard) ones mentioned above from lambda calculus.

7.1.1 Opening a Lambda Abstraction

The **open** construct "opens" a lambda abstraction to allow computation on its bound variable and its body. An **open** expression has two subexpressions. The first is the term to attempt to open. The bound variable and the body of this term, if it is a lambda abstraction, are both passed to the second argument of **open**. For an artificial but representative example, the following term opens $\lambda x.x$ and applies its body (x, passed as the argument for variable b) to itself:

$$\textbf{open}\ (\lambda x.x)\ (\lambda v\ b.(b\ b))$$

Since ARCHON is intended to be scope safe, **open** cannot simply return $x\ x$ in this case, for then the bound variable x would escape its scope. The solution adopted in ARCHON is simply to rebind the variable around the result of the computation, using the same kind of abstraction (call-by-name or call-by-value) as the opened lambda abstraction. So the above example term evaluates to $\lambda x.x\ x$. The behavior of **open** when its first subexpression is not a lambda abstraction is not important. Below, we choose to have the **open** expression evaluate to a rather arbitrary term. It would also be reasonable to have it abort the computation. This construct has some features in common with a recently proposed reflective construct called **map-closure** [40]. The difference is that **map-closure** operates on environments in closures, not lambda abstractions. Hence, it is a computationally rather than structurally reflective construct, and not suitable in any obvious way for general meta-programming.

7.1.2 Comparing Variables

For some meta-programs it is necessary to test whether two free variables are the same or different. This is done in ARCHON using the **vcomp** construct. The term **vcomp** T_1 T_2 evaluates to *true* if T_1 and T_2 are identical free variables. Otherwise it evaluates to *false*. Here and below, the expressions *true* and *false* are abbreviations for Scott-encoded call-by-name booleans $\bar{\lambda}x.\bar{\lambda}y.x$ and $\bar{\lambda}x.\bar{\lambda}y.y$. Note that the Scott and Church encodings coincide on enumerated datatypes like the boolean datatype. Note also that using call-by-name abstractions here (and in other Scott-encoded data) means that unused cases are not executed when a piece of data is used as a case construct.

7.1.3 Swapping Lambda Abstractions

The operation **swap** is used to swap the order of consecutive lambda bindings. For example, our operational semantics will give us the following evaluation (where $\Downarrow$, defined below, is the evaluation relation):

$$\textbf{swap}\ (\lambda x\ y.x) \Downarrow \lambda y\ x.x$$

It may not be obvious to the reader why this operation is practically useful, but it turns out to be necessary for a class of practical meta-programs, in particular those which must recursively traverse ARCHON terms to compute some resulting term, using the decomposition operator discussed next. An example of such a meta-program is given later, in Figure 9.

It is not clear if **swap** can be defined using other language features. One might consider defining it by opening the lambda abstraction twice, and then using a recursive function to traverse the body and swap the variables (aided, for example, by variable comparison) wherever they occur. But such traversals, as just mentioned, appear to need **swap** already. So it may not be possible to define **swap** in terms of other operations. This question must be left open. Just as for **open**, the behavior of **swap** when its subexpression does not evaluate to a consecutively nested lambda abstraction is not important, and is chosen to be a rather arbitrary term.

7.1.4 Decomposition

The final construct of Figure 1 is called decomposition. This is an intensional case-analysis construct, which takes apart raw (i.e., unevaluated) program terms, and passes their immediate subterms as arguments to the appropriate branch of the case. In the decomposition

$$T : T\ T\ T\ T\ T\ T\ T$$

the first term is the one being decomposed, and the remaining seven terms are the terms to use for each of the seven kinds of constructs, in the order in which they appear in Figure 1. Observe that ARCHON does have exactly seven constructs, if we lump call-by-value and call-by-name lambda abstractions

together, as we do. They are distinguished in decompositions by passing a Scott-encoded boolean to the case for lambda abstractions, which is *true* for call-by-value, and *false* for call-by-name. For a simple example of the use of decomposition, the following term evaluates to $b\ a$ (here writing $\_$ in the unused cases for an arbitrary lambda term):

$$(a\ b)\ :\ \_\ \_\ (\lambda x\ y.y\ x)\ \_\ \_\ \_\ \_$$

A final note on decomposition is that decomposing a lambda abstraction does not access its subexpressions, since this functionality is already provided by **open**.

7.2 Operational Semantics

A big-step operational semantics for ARCHON is given in Figures 2, 3, and 4. The judgment $T \Downarrow R$ means that the term T evaluates to R. In the rules E-Lam and E-OpenLam, λ^* is either lambda abstraction operator (λ or $\bar{\lambda}$), as are λ^1 and λ^2 in the rule E-Swap. Variable renaming is accounted for explicitly in E-OpenLam and by capture-avoiding substitution, denoted $[T_2/x]T_1$. It is necessary, in general, to rename the bound variable in E-OpenLam, to avoid clashes with existing free variables.

The choice of $\lambda x.\mathit{false}$ for the result of attempting to open or swap a term which is not a lambda abstraction or a consecutively nested lambda abstraction in rules E-Open and E-Swap, respectively, is not essential. There is also some arbitrariness in whether or not to evaluate the first subexpressions of **open**, **swap**, and **vcomp** expressions. The language is not greatly changed by the choice, for each of those three constructs, of whether to evaluate this subexpression before reduction or not.

Note that evaluation of terms without **open**, **vcomp**, **swap**, and decomposition (Figure 2) is contained in the β-equivalence relation (on open terms) of the standard untyped λ-calculus, if we collapse both lambda abstractions to the standard one. The meta-level function FV used in E-OpenLam is defined in a standard way to give the set of free variables of a term.

7.3 Meta-Theory

The side condition in E-OpenLam prevents confusion of the newly introduced free variable with other free variables. The only other potential opportunity for variable capture is in applying substitutions, but these are capture-avoiding by definition. This fact and the following theorem, easily proved by induction on the structure of computations, establish scope safety of the language:

Theorem 1 *For all terms T and R, if $T \Downarrow R$, then $FV(R) \subseteq FV(T)$.*

The following theorem can also be established. Together with the implementability of a test for α-equivalence in the language (Section 8.2), it suffices to prove that α-equivalence of programs coincides with contextual equivalence.

$$\frac{}{x \Downarrow x}\ \textsf{E-Var}$$

$$\frac{}{\lambda^* x.T \Downarrow \lambda^* x.T}\ \textsf{E-Lam}$$

$$\frac{T_1 \Downarrow \lambda x.R_1 \qquad T_2 \Downarrow R_2 \qquad [R_2/x]R_1 \Downarrow R}{T_1\ T_2 \Downarrow R}\ \textsf{E-AppCbv}$$

$$\frac{T_1 \Downarrow \bar{\lambda} x.R_1 \qquad [T_2/x]R_1 \Downarrow R}{T_1\ T_2 \Downarrow R}\ \textsf{E-AppCbn}$$

$$\frac{T_1 \Downarrow R_1 \qquad T_2 \Downarrow R_2 \qquad R_1 \textit{ not a lambda abstraction}}{T_1\ T_2 \Downarrow R_1\ R_2}\ \textsf{E-App}$$

Figure 2: Evaluation of Standard Constructs

$$\frac{T_2\ y\ [y/x]T_1 \Downarrow R \qquad y \notin FV(\lambda^* x.T_1) \cup FV(T_2)}{\textbf{open}\ (\lambda^* x.T_1)\ T_2 \Downarrow \lambda^* y.R}\ \textsf{E-OpenLam}$$

$$\frac{T_1 \textit{ not a lambda abstraction}}{\textbf{open}\ T_1\ T_2 \Downarrow \lambda x.\mathit{false}}\ \textsf{E-Open}$$

$$\frac{}{\textbf{vcomp}\ x\ x \Downarrow \mathit{true}}\ \textsf{E-VcompEqVars}$$

$$\frac{T_1 \not\equiv T_2 \textit{ or } T_1 \textit{ or } T_2 \textit{ not a variable}}{\textbf{vcomp}\ T_1\ T_2 \Downarrow \mathit{false}}\ \textsf{E-Vcomp}$$

$$\frac{T \Downarrow \lambda^1 x.\lambda^2 y.R}{\textbf{swap}\ T \Downarrow \lambda^2 y.\lambda^1 x.R}\ \textsf{E-SwapLam}$$

$$\frac{T \Downarrow R_1 \qquad R_1 \textit{ not a consecutively nested lambda abstraction}}{\textbf{swap}\ T \Downarrow \lambda x.\mathit{false}}\ \textsf{E-Swap}$$

Figure 3: Evaluation of Swap, Open, and Vcomp

$$\frac{T_1\ z \Downarrow R}{z\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompVar}$$

$$\frac{T_2\ \mathit{true}\ \lambda z.M \Downarrow R}{(\lambda z.M)\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompCbv}$$

$$\frac{T_2\ \mathit{false}\ \bar{\lambda} z.M \Downarrow R}{(\bar{\lambda} z.M)\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompCbn}$$

$$\frac{T_3\ M\ N \Downarrow R}{(M\ N)\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompApp}$$

$$\frac{T_4\ M\ N \Downarrow R}{(\mathbf{open}\ M\ N)\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompOpen}$$

$$\frac{T_5\ M\ N \Downarrow R}{(\mathbf{vcomp}\ M\ N)\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompVcomp}$$

$$\frac{T_6\ M \Downarrow R}{(\mathbf{swap}\ M)\ :\ T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R}\ \textsf{E-DecompSwap}$$

$$\frac{T_7\ M\ M_1\ M_2\ M_3\ M_4\ M_5\ M_6\ M_7 \Downarrow R}{\begin{array}{l}(M : M_1\ M_2\ M_3\ M_4\ M_5\ M_6\ M_7)\\ \quad : T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \Downarrow R\end{array}}\ \textsf{E-DecompDecomp}$$

Figure 4: Evaluation of Decomposition

Theorem 2 *For all α-equivalent terms T and T', if $T \Downarrow R$, then $T' \Downarrow R'$ where R' is α-equivalent to R.*

Proof. The proof is by induction on the structure of $T \Downarrow R$. We consider just the case for E-OpenLam. We have

$$\frac{T_2\ y\ [y/x]T_1 \Downarrow R_1 \qquad y \notin FV(\lambda^* x.T_1) \cup FV(T_2)}{\textbf{open}\ (\lambda^* x.T_1)\ T_2 \Downarrow \lambda^* y.R_1}$$

and also

$$\Gamma \vdash (\textbf{open}\ (\lambda^* x.T_1)\ T_2) =_\alpha (\textbf{open}\ (\lambda^* x'.T_1')\ T_2')$$

From these facts, we get $y \notin FV(\lambda^* x'.T_1') \cup FV(T_2')$. Hence, we have

$$T_2\ y\ [y/x]T_1 \ =_\alpha \ T_2'\ y\ [y/x']T_1'$$

The induction hypothesis may now be applied to obtain $R' =_\alpha R$ with

$$T_2'\ y\ [y/x']T_1' \Downarrow R_1'$$

We can then obtain the desired evaluation judgment by applying E-OpenLam, with $\lambda^* y.R_1$, which is easily seen to be α-equivalent to $\lambda^* y.R_1'$.

8 Meta-Programming Examples

We now consider meta-programming examples in Archon. We may distinguish meta-programs which recursively inspect code from meta-programs which also recursively produce it. The former class can be implemented in Wand's language, since inspection of Mogensen-Scott encoded lambda terms is supported. As discussed in Section 6, the latter class is not in any obvious way implementable, except in trivial cases (e.g., constant functions). Archon can implement both kinds of meta-programs, and has some advantages for implementing code-inspecting meta-programs. So we begin by comparing Archon and Wand's language on a meta-program which is canonical for Wand's purposes, namely testing arbitrary terms for α-equivalence. Note that this is provably not implementable in untyped lambda calculus, so both Wand's language and Archon are more expressive (see, e.g., [23, Section 3.3.4]). Section 8.3 gives another representative meta-program, namely to compute from n and f the function $\lambda x.f^n\ x$, where f^n denotes n-fold application of f.

8.1 Alpha Equivalence Test in Wand's System

In Wand's system, we can define a test *eq* for α-equivalence of encoded terms as a term of the language. This *eq* cannot take in arbitrary terms and then encode them itself using **fexpr**, since evaluation of an application of *eq* to two arguments will cause those arguments to be evaluated. So it must operate on Mogensen-Scott encoded terms (cf. the family *Eq* of terms defined in [50,

Figure 1]). It must furthermore operate only on closed lambda terms, since it has no means to inspect free variables. Indeed, as *eq* traverses terms, it must substitute some entities which it can inspect for all the bound variables retained in the Mogensen-Scott encoding. It may use Scott- or Church-encoded natural numbers for this purpose. The number to introduce for the next bound variable it encounters must be taken as an additional input to *eq*.

8.1.1 Helper Functions

The code for *eq* makes use of several defined helper functions. We use a standard call-by-value fixed point operator *fix*, such as given in Section 6. We assume S is the successor function in whichever encoding of natural numbers we are using, and *eqnat* is the equality test for natural numbers (straightforwardly implementable in both the Scott and Church encodings). We also use *and* as an abbreviation for a standard implementation of conjunction on Scott-encoded booleans. It is helpful also to define several operations on Mogensen-Scott encoded programs ("x") which use one continuation ("t") when encoded data has a certain form (e.g., is an encoded application), and another ("f") when it does not have that form. Writing *F1* as an abbreviation for $\lambda y.f$ and *F2* as an abbreviation for $\lambda y_1\ y_2.f$, we have

$$\begin{array}{lcl} \mathit{on\_app} & := & \lambda x\ t\ f.(x\ \mathit{F1}\ t\ \mathit{F1}\ \mathit{F1}\ \mathit{F1}) \\ \mathit{on\_lam} & := & \lambda x\ t\ f.(x\ \mathit{F1}\ \mathit{F2}\ t\ \mathit{F1}\ \mathit{F1}) \\ \mathit{on\_fexpr} & := & \lambda x\ t\ f.(x\ \mathit{F1}\ \mathit{F2}\ \mathit{F1}\ t\ \mathit{F1}) \\ \mathit{on\_eval} & := & \lambda x\ t\ f.(x\ \mathit{F1}\ \mathit{F2}\ \mathit{F1}\ \mathit{F1}\ t) \end{array}$$

So we have

$$\mathit{on\_app}\ \ulcorner(a\ b)\urcorner\ (\lambda x\ y.x)\ \mathit{false}$$

evaluating to a, and

$$\mathit{on\_app}\ \ulcorner(\lambda x.x)\urcorner\ (\lambda x\ y.x)\ \mathit{false}$$

evaluating to *false*. We also take "$\mathit{let}\ x\ =\ T_1\ \mathit{in}\ T_2$" to abbreviate $(\lambda x.T_2)\ T_1$, as usual.

8.1.2 The Code for *eq*

The code for *eq* in Wand's system is given in Figure 5. The function *eq* takes input n for the next number to introduce for a bound variable, and inputs s and t for the closed Mogensen-Scott encoded terms to compare for α-equivalence. The basic idea of this code is first to do a case analysis on s, and then, in each case, a subsequent case analysis on t. Recall from Section 5.2 that Scott-encoded data are effectively their own case analyses. So to perform a case analysis on s, we apply it to five continuations, one for each of the five syntactic forms s could have. In each case, we perform the subsequent case analysis on t using the helper functions from the previous Section. When s is a lambda abstraction (in the third case), we apply that lambda abstraction to the next natural number to

use for its bound variable. This, of course, causes the number to be substituted for the bound variable. It can then be inspected in the case for variables (the first case).

8.2 Alpha Equivalence Test in Archon

The ARCHON implementation of a test *eq* for α-equivalence is similar to the implementation above in Wand's system. Here, we factor out recursively checking equality of constructs with two subterms as *eq2*, to keep the code more concise. One advantage of implementing *eq* in ARCHON is that we can test α-equivalence of program terms containing free variables. Furthermore, it is convenient not to have to introduce numbers for bound variables. Also, we do not need to assume that those terms are in an encoded form, since ARCHON meta-programs operate directly on raw program terms.

8.2.1 Helper Functions

We use standard helper functions *and* and *beq* for conjunction and equality test on Scott-encoded booleans. We make use of functions similar in spirit to the functions like *on_app* in the implementation in Wand's system above. Below we write *F1* as an abbreviation for $\bar{\lambda}m.f$, and similarly *F2* for $\bar{\lambda}m\ n.f$, and *F8* for $\bar{\lambda}m\ m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7.f$. The functions are then:

$$\begin{aligned}
\mathit{on_lam} &:= \bar{\lambda}u\ t\ f.u\ :\ \mathit{F1}\ t\ \mathit{F2}\ \mathit{F2}\ \mathit{F2}\ \mathit{F1}\ \mathit{F8}\\
\mathit{on_app} &:= \bar{\lambda}u\ t\ f.u\ :\ \mathit{F1}\ \mathit{F2}\ t\ \mathit{F2}\ \mathit{F2}\ \mathit{F1}\ \mathit{F8}\\
\mathit{on_open} &:= \bar{\lambda}u\ t\ f.u\ :\ \mathit{F1}\ \mathit{F2}\ \mathit{F2}\ t\ \mathit{F2}\ \mathit{F1}\ \mathit{F8}\\
\mathit{on_vcomp} &:= \bar{\lambda}u\ t\ f.u\ :\ \mathit{F1}\ \mathit{F2}\ \mathit{F2}\ \mathit{F2}\ t\ \mathit{F1}\ \mathit{F8}\\
\mathit{on_swap} &:= \bar{\lambda}u\ t\ f.u\ :\ \mathit{F1}\ \mathit{F2}\ \mathit{F2}\ \mathit{F2}\ \mathit{F2}\ t\ \mathit{F8}\\
\mathit{on_decomp} &:= \bar{\lambda}u\ t\ f.u\ :\ \mathit{F1}\ \mathit{F2}\ \mathit{F2}\ \mathit{F2}\ \mathit{F2}\ \mathit{F1}\ t
\end{aligned}$$

We also define *nfix* as follows, for a standard call-by-name fixpoint operator:

$$\mathit{nfix} := \lambda f.(\lambda x.f\ (\bar{\lambda}y.x\ x\ y))\ (\lambda x.f\ (\bar{\lambda}y.x\ x\ y))$$

This operator must be used instead of *fix*, since the ARCHON implementation of *eq* uses call-by-name lambda abstraction to receive the two terms to test for α-equivalence.

The ARCHON code for *eq* also needs a helper function *beta_reducek* for performing a single β-reduction. The basic idea of this function is that given two terms, where the first term M is either of the form $\lambda x.B$ or of the form $\bar{\lambda}x.B$, and the second is N; it should return $[N/x]B$. We cannot return this term directly, without risk that it might reduce. So we return it instead via a continuation, provided as a third argument. Here and below, let I abbreviate $\lambda z.z$. The code for this helper function is the following, which requires some explanation (note that _ is used here just as another variable, with the name chosen to indicate

```
eq  :=  fix λeq n s t.
          (s (λx.eqnat x t)
             (λs1 s2.on_app t (λt1 t2.and (eq n s1 t1) (eq n s2 t2)) false)
             (λs1.on_lam t (λt1.eq (S n) (s1 n) (t1 n)) false)
             (λs1.on_fexpr t (λt1.eq n s1 t1) false)
             (λs1.on_eval t (λt1.eq n s1 t1) false))
```

Figure 5: Test for α-Equivalence in Wand's System

```
eq  :=  nfix λeq.λ̄s t.
          let eq2 = λon_op.λ̄m n.
                      (on_op t (λ̄m2 n2.and (eq m m2) (eq n n2))
                               false) in
          (s : (vcomp s t)
             (λ̄cbv1 s.
                on_lam t (λ̄cbv2 t.
                            and (beq cbv1 cbv2)
                                ((open s λ̄x tb.(beta_reducek t x (eq tb)))
                                 I))
                   false)
             (eq2 on_app)
             (eq2 on_open)
             (eq2 on_vcomp)
             (λ̄m.on_swap t (eq m) false)
             (λ̄m0 m1 m2 m3 m4 m5 m6 m7.
                on_decomp t
                   (λ̄n0 n1 n2 n3 n4 n5 n6 n7.
                      (and (eq m0 n0) (and (eq m1 n1)
                      (and (eq m2 n2) (and (eq m3 n3)
                      (and (eq m4 n4) (and (eq m5 n5)
                      (and (eq m6 n6) (eq m7 n7)))))))))
                   false))
```

Figure 6: Test for α-Equivalence in Archon

informally that it is not subsequently used):

$$\begin{array}{lcl} \mathit{beta\_reducek} & := & \bar{\lambda}M\ N\ k. \\ & & \mathit{let}\ M1\ =\ \mathbf{open}\ M\ \bar{\lambda}x\ B.\lambda z.B\ \mathit{in} \\ & & \mathit{let}\ M2\ =\ \mathbf{open}\ (\bar{\lambda}y.y)\ (\bar{\lambda}\_\ y.M1\ y)\ \mathit{in} \\ & & \mathit{let}\ R\ =\ M2\ N\ \mathit{in} \\ & & ((\mathbf{open}\ R\ \bar{\lambda}\_\ b.k\ b)\ I) \end{array}$$

The action of this code is rather subtle, but it well illustrates the power of the **open** construct. Our idea in computing $[N/x]B$ from $\lambda x.B$ and N is illustrated informally by the following transformation sequence (the case for $\bar{\lambda}x.B$ is similar):

$$(\lambda x.B)\ N\ \rightsquigarrow\ (\lambda x.\lambda z.B)\ N\ \rightsquigarrow\ (\bar{\lambda}x.\lambda z.B)\ N\ \rightsquigarrow\ \lambda z.[N/x]B$$

In more detail, we first wish to insert a dummy binder beneath the λx, so that applying the lambda abstraction to N will not trigger reductions in B. So we first want to compute a term *M1* defined to be $\lambda x.\lambda z.B$. Then we want to convert the call-by-value λx to a call-by-name $\bar{\lambda}x$, so that applying the lambda abstraction to N will not cause N to evaluate. So we next want to compute a term *M2* defined to be $\bar{\lambda}x.\lambda z.B$. Finally, we can achieve our β-reduction by applying this *M2* directly to N, and allowing reduction in Archon to perform the substitution. This results in a term R of the form $\lambda z.[N/x]B$. It suffices now just to open this lambda abstraction and apply the continuation k to its body (which is $[N/x]B$). Whatever result the continuation produces will then be closed beneath a binding λz, since uses of open (such as this one opening R) always rebind the variable of the opened lambda abstraction. To eliminate this dummy binding, we apply the result to an arbitrary value (here, I).

We leave it to the reader to confirm that the code above computes *M1* as specified, and just consider in detail the computation of *M2*. The code for *beta_reducek* computes *M2* ("$\bar{\lambda}x.\lambda z.B$") from *M1* as follows. It opens a new lambda abstraction $\bar{\lambda}y.y$, using $\bar{\lambda}\_\ y.M1\ y$. The latter term accepts the bound variable (y, received via variable _) and the body (also y, received by variable y) of this $\bar{\lambda}y.y$. It then applies *M1* to y. This results in $\lambda z.[y/x]B$. The result of this computation is then closed beneath a rebinding $\bar{\lambda}y$, since this is always how evaluation of an **open** expression finishes. So the result is $\bar{\lambda}y.\lambda z.[y/x]B$, which is α-equivalent to the desired term $\bar{\lambda}x.\lambda z.B$. The use of variable y here is purely for readability of the example. Thanks to the scope safety of Archon, the code works just as well if we use x instead of y.

This example provides some small evidence for completeness. It shows that an operation, namely changing a lambda abstraction from call-by-value to call-by-name, which one might otherwise doubt possible in Archon, is in fact implementable. Similar code can convert a call-by-name to a call-by-value abstraction.

8.2.2 The Code for *eq*

Figure 6 gives the ARCHON code for the test *eq* for α-equivalence. Its basic structure resembles that of the code in Wand's language (Figure 5). Some differences have already been noted. The top-level case analysis is performed using ARCHON's decomposition operator, instead of by application of a Mogensen-Scott encoded term. The first case, for variables, is implemented using **vcomp**, as expected. The case for lambda abstractions (the second case) is the most complex. Naturally, for two lambda abstractions to be α-equivalent, they must either both be call-by-value or both call-by-name. This is checked by *beq* cbv_1 cbv_2. To continue the comparison, we wish to compare the bodies. But we must make sure the bodies are expressed using the same variable. So we open one lambda abstraction ("*s*"), obtain its bound variable ("*x*") and do a single β-reduction of t on x (using *beta_reducek*, defined in the previous Section). This causes uses of t's bound variable to be replaced in its body ("*tb*") by x. We may then compare the two bodies. The lambda binder placed around the (boolean) result is then removed by applying to I (defined, as stated above, to be $\lambda z.z$).

8.3 Statically Iterated Function Application

We implement a familiar meta-programming example in ARCHON, namely statically iterated function application. The problem is to compute from n and f the function $\lambda x.f^n\ x$, where f^n denotes n-fold application of f. We rely here on the helper function *fix*, defined exactly as above for Wand's language; as well as *zero*, *succ*, and *plus* for Scott-encoded unary numbers. A rather simple function can build $f^n\ x$ from f, n, and x. If the function f is a lambda abstraction (as in general we would expect it to be), then some care must be taken to return $f^n\ x$ in such a way that it does not prematurely reduce. The code *iter_h* in Figure 7 achieves this using the following idea. We return code from *iter_h* underneath a dummy lambda abstraction. To access the result of a recursive call, we must then open this dummy lambda abstraction, access the body ("*rb*"), and then place the result beneath a new dummy lambda abstraction ("$\lambda d.f\ rb$"). Since opening the first dummy lambda abstraction rebinds the dummy variable, we end up with two dummy lambda abstractions at the start of the term. We can eliminate the first one simply by applying the result of the open to some arbitrary value (here, I). Then to build the function $\lambda x.f^n\ x$, the ARCHON term *iter* in the figure just needs to open a new lambda abstraction ("$\lambda x.x$") and use *iter_h* to build $\lambda d.f^n\ x$. When the open returns, we have $\lambda x.\lambda d.f^n\ x$. To eliminate the dummy binding of d, *iter* just swaps the two lambda abstractions and applies the result to an arbitrary value ("*I*"). A similar idea is used in the third piece of code in the figure, for multiplication by statically iterated addition (cf. [32, Section 2]).

$$
\begin{array}{l}
\mathit{iter\_h} \;:=\; \mathit{fix}\;\lambda \mathit{iter\_h}\;x\;n\;f. \\
\quad (n\;(\lambda p.\mathit{let}\;r \;=\; \mathit{iter\_h}\;x\;p\;f\;\mathit{in} \\
\qquad\quad ((\mathbf{open}\;r\;\bar{\lambda} x\;rb.\lambda d.f\;rb)\;I)) \\
\qquad \lambda d.x)
\end{array}
$$

$$
\mathit{iter} \;:=\; \lambda n\;f.(\mathbf{swap}\;(\mathbf{open}\;(\lambda x.x)\;\lambda x\;\_.\mathit{iter\_h}\;x\;n\;f))\;I
$$

$$
\mathit{mult} \;:=\; \lambda n.(\mathbf{swap}\;(\mathbf{open}\;(\lambda m.m)\;\lambda m\;\_.\mathit{iter\_h}\;\mathit{zero}\;n\;(\mathit{plus}\;m)))\;I
$$

Figure 7: Statically Iterated Function Application in ARCHON

8.4 Encoding and Decoding in Archon

We can implement Mogensen-Scott encoding and decoding functions (corresponding to **fexpr** and **eval** in Wand's language) in ARCHON as follows. Figure 8 first defines the Mogensen-Scott encoding for ARCHON terms, and then gives an ARCHON term which when applied to a term t, evaluates to $\ulcorner t \urcorner$ (see Figure 8). The crucial ideas are the use of call-by-name lambda abstractions to input raw program terms, and the use of **open** to recurse on the bodies of lambda abstractions. Correctness of the code can be expressed like this:

Theorem 3 *For all terms T, encode $T \Downarrow \ulcorner T \urcorner$.*

Figure 9 defines the decoding function for the Mogensen-Scott encoding of ARCHON terms, and then gives an implementation in ARCHON. Notice that where the encoding function uses ARCHON's decomposition construct, the decoding function may just apply the Scott-encoded data as a case statement. As for the example of iterated function application above, the crucial idea is to return expressions beneath a dummy lambda abstraction (binding variable d), in order to prevent decoded expressions from reducing. The bodies of lambda abstractions are accessed using $\mathbf{open}^*$, defined as follows:

$$
\mathbf{open}^* \quad := \quad \lambda x.\bar{\lambda} y.\mathbf{open}\;x\;y
$$

This term works just like **open** except that it evaluates its first argument. It is used in the code for *decode* to cause the recursive calls to execute instead of intensionally analyzing them. (Recall from Section 7.2 that for inessential reasons, we have designed the operational semantics of **open** so that it does not evaluate its first argument.) Also, recall that, as stated above, I is defined to be $\lambda z.z$.

The only complication that arises is for the case of lambda abstractions (the second case in the ARCHON code of Figure 9). After the body has been decoded and evaluation of the **open** expression completes, we have a dummy lambda abstraction trapped beneath the binding that has been replaced by the evaluation of **open**. This is similar to the situation with *iter* in Section 8.3. We just swap the two bindings using **swap**. For example, the Mogensen-Scott

$$
\begin{array}{rcl}
\ulcorner x \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.V\ x \\
\ulcorner \lambda z.T \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.L\ \mathit{true}\ \lambda z.\ulcorner T \urcorner \\
\ulcorner \bar{\lambda} z.T \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.L\ \mathit{false}\ \bar{\lambda} z.\ulcorner T \urcorner \\
\ulcorner T_1\ T_2 \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.A\ \ulcorner T_1 \urcorner\ \ulcorner T_2 \urcorner \\
\ulcorner \mathbf{open}\ T_1\ T_2 \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.O\ \ulcorner T_1 \urcorner\ \ulcorner T_2 \urcorner \\
\ulcorner \mathbf{vcomp}\ T_1\ T_2 \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.C\ \ulcorner T_1 \urcorner\ \ulcorner T_2 \urcorner \\
\ulcorner \mathbf{swap}\ T \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D.S\ \ulcorner T \urcorner \\
\ulcorner T : T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7 \urcorner & = & \lambda V\ L\ A\ O\ C\ S\ D. \\
 & & \quad D\ \ulcorner T \urcorner\ \ulcorner T_1 \urcorner\ \ulcorner T_2 \urcorner\ \ulcorner T_3 \urcorner \\
 & & \quad\ \ulcorner T_4 \urcorner\ \ulcorner T_5 \urcorner\ \ulcorner T_6 \urcorner\ \ulcorner T_7 \urcorner
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{encode} & := & \mathit{nfix}\ \lambda \mathit{encode}.\bar{\lambda} t.t : \\
 & & \quad (\lambda x.\lambda V\ L\ A\ O\ C\ S\ D.V\ x) \\
 & & \quad (\lambda b\ F.\ \mathit{let}\ r\ =\ \mathbf{open}\ F\ (\lambda x.\bar{\lambda} F'.\mathit{encode}\ F')\ \mathit{in} \\
 & & \qquad \lambda V\ L\ A\ O\ C\ S\ D.L\ b\ r) \\
 & & \quad (\bar{\lambda} M\ N.\ \mathit{let}\ r\ =\ \mathit{encode}\ M\ \mathit{in} \\
 & & \qquad \mathit{let}\ s\ =\ \mathit{encode}\ N\ \mathit{in}\ \lambda V\ L\ A\ O\ C\ S\ D.A\ r\ s) \\
 & & \quad (\bar{\lambda} M\ N.\ \mathit{let}\ r\ =\ \mathit{encode}\ M\ \mathit{in} \\
 & & \qquad \mathit{let}\ s\ =\ \mathit{encode}\ N\ \mathit{in}\ \lambda V\ L\ A\ O\ C\ S\ D.O\ r\ s) \\
 & & \quad (\bar{\lambda} M\ N.\ \mathit{let}\ r\ =\ \mathit{encode}\ M\ \mathit{in} \\
 & & \qquad \mathit{let}\ s\ =\ \mathit{encode}\ N\ \mathit{in}\ \lambda V\ L\ A\ O\ C\ S\ D.C\ r\ s) \\
 & & \quad (\bar{\lambda} M.\mathit{let}\ r\ =\ \mathit{encode}\ M\ \mathit{in}\ \lambda V\ L\ A\ O\ C\ S\ D.S\ r) \\
 & & \quad (\bar{\lambda} M\ M_1\ M_2\ M_3\ M_4\ M_5\ M_6\ M_7. \\
 & & \qquad \mathit{let}\ r\ =\ \mathit{encode}\ M\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_1\ =\ \mathit{encode}\ M_1\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_2\ =\ \mathit{encode}\ M_2\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_3\ =\ \mathit{encode}\ M_3\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_4\ =\ \mathit{encode}\ M_4\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_5\ =\ \mathit{encode}\ M_5\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_6\ =\ \mathit{encode}\ M_6\ \mathit{in} \\
 & & \qquad \mathit{let}\ r_7\ =\ \mathit{encode}\ M_7\ \mathit{in} \\
 & & \qquad \lambda V\ L\ A\ O\ C\ S\ D.D\ r\ r_1\ r_2\ r_3\ r_4\ r_5\ r_6\ r_7)
\end{array}
$$

Figure 8: Mogensen-Scott Encoding and Its Implementation in ARCHON

$$
\begin{array}{rcl}
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.V\ x \lrcorner & = & x \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.L\ \mathit{true}\ \lambda x.T \lrcorner & = & \lambda x.\llcorner T \lrcorner \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.L\ \mathit{false}\ \bar{\lambda} x.T \lrcorner & = & \bar{\lambda} x.\llcorner T \lrcorner \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.A\ T_1\ T_2 \lrcorner & = & \llcorner T_1 \lrcorner\ \llcorner T_2 \lrcorner \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.O\ T_1\ T_2 \lrcorner & = & \mathbf{open}\ \llcorner T_1 \lrcorner\ \llcorner T_2 \lrcorner \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.C\ T_1\ T_2 \lrcorner & = & \mathbf{vcomp}\ \llcorner T_1 \lrcorner\ \llcorner T_2 \lrcorner \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.S\ T \lrcorner & = & \mathbf{swap}\ \llcorner T \lrcorner \\
\llcorner \lambda V\ L\ A\ O\ C\ S\ D.D\ T\ T_1\ T_2\ T_3 & & \\
\qquad T_4\ T_5\ T_6\ T_7 \lrcorner & = & \llcorner T \lrcorner : \llcorner T_1 \lrcorner\ \llcorner T_2 \lrcorner\ \llcorner T_3 \lrcorner \\
& & \qquad \llcorner T_4 \lrcorner\ \llcorner T_5 \lrcorner\ \llcorner T_6 \lrcorner\ \llcorner T_7 \lrcorner
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{decode} & := & \mathit{fix}\ \lambda \mathit{decode}.\lambda t.t \\
& & \quad (\lambda x\ d.x) \\
& & \quad (\lambda b\ F.\mathbf{swap}\ (\mathbf{open}^*\ F\ (\lambda\_.\lambda F'.\mathit{decode}\ F'))) \\
& & \quad (\lambda M\ N.\ (\mathbf{open}^*\ (\mathit{decode}\ M)\ (\lambda\_.\bar{\lambda} M'. \\
& & \qquad \mathbf{open}^*\ (\mathit{decode}\ N)\ (\lambda\_.\bar{\lambda} N'.\ \lambda d.M'\ N'))))\ I\ I) \\
& & \quad (\lambda M\ N.\ (\mathbf{open}^*\ (\mathit{decode}\ M)\ (\lambda\_.\bar{\lambda} M'. \\
& & \qquad \mathbf{open}^*\ (\mathit{decode}\ N)\ (\lambda\_.\bar{\lambda} N'.\ \lambda d.\mathbf{open}\ M'\ N'))))\ I\ I) \\
& & \quad (\lambda M\ N.\ (\mathbf{open}^*\ (\mathit{decode}\ M)\ (\lambda\_.\bar{\lambda} M'. \\
& & \qquad \mathbf{open}^*\ (\mathit{decode}\ N)\ (\lambda\_.\bar{\lambda} N'.\ \lambda d.\mathbf{vcomp}\ M'\ N'))))\ I\ I) \\
& & \quad (\lambda M.(\mathbf{open}^*\ (\mathit{decode}\ M)\ (\lambda\_.\bar{\lambda} M'.\ \lambda d.\mathbf{swap}\ M'))\ I) \\
& & \quad (\lambda M\ M_1\ M_2\ M_3\ M_4\ M_5\ M_6\ M_7. \\
& & \qquad (\mathbf{open}^*\ (\mathit{decode}\ M)\ \lambda\_.\bar{\lambda} M'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_1)\ \lambda\_.\bar{\lambda} M_1'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_2)\ \lambda\_.\bar{\lambda} M_2'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_3)\ \lambda\_.\bar{\lambda} M_3'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_4)\ \lambda\_.\bar{\lambda} M_4'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_5)\ \lambda\_.\bar{\lambda} M_5'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_6)\ \lambda\_.\bar{\lambda} M_6'. \\
& & \qquad\ \mathbf{open}^*\ (\mathit{decode}\ M_7)\ \lambda\_.\bar{\lambda} M_7'. \\
& & \qquad\quad \lambda d.(M : M_1'\ M_2'\ M_3'\ M_4'\ M_5'\ M_6'\ M_7'))) \\
& & \qquad I\ I\ I\ I\ I\ I\ I\ I)
\end{array}
$$

Figure 9: Mogensen-Scott Decoding and Its Implementation in ARCHON

encoding of $\bar{\lambda}x.x$ is

$$\lambda V\ L\ A\ O\ C\ S\ D.L\ \mathit{false}\ \bar{\lambda}x.\lambda V\ L\ A\ O\ C\ S\ D.V\ x$$

If *decode* is called on this term, the code in the second case of Figure 9 tells us first to open the λx and recursively decode the body (*decode* F'). This gives us just x, frozen beneath a dummy lambda abstraction. When the **open** expression finishes, we thus have $\bar{\lambda}x.\lambda d.x$. To finish decoding the lambda abstraction, we need to pull the λd out from beneath the $\bar{\lambda}x$. This is done (by the code in the second case) using **swap**, which yields $\lambda d.\bar{\lambda}x.x$, as desired.

In all the other cases, a combination of **open** and call-by-name abstraction is used to move live code from under dummy abstractions and reassemble that code under a new dummy abstraction. Since **open** always replaces the bound variable (in this case, a now unused dummy), it is necessary to apply the resulting term to several arbitrary terms (here, I) to eliminate the unused dummies. Correctness of the implementation can be expressed as follows:

Theorem 4 *For all terms T, decode $\ulcorner T \urcorner \Downarrow \lambda d.T$.*

9 Implementation

As pointed out by Wand, in any language where α-equivalence of program terms coincides with contextual equivalence of their encodings, compilation steps like source-to-source optimization are rendered unsound. Sound, efficient implementation of structurally reflective languages like ARCHON is thus a non-trivial issue. It is not an insuperable one, however. For example, source-to-source optimization could be allowed for programs which are not analyzed reflectively, as determined by a static analysis. Furthermore, while source-to-source optimization is unsound in general, it may still be possible to design more efficient abstract machines for ARCHON, incorporating optimizations to the entire operational semantics. Exploring these ideas must remain to future work.

This section discusses initial efforts at an graph-reducing interpreter. Approaches based on compilation to an abstract machine result in large performance gains over interpretation, and are considered necessary for serious implementation. Nevertheless, interpretation is the appropriate starting point for a language with novel constructs like ARCHON's (cf. [19]). The prototype ARCHON interpreter is written in just under 1000 lines of JAVA, version 1.4.2. It may be downloaded from `http://cl.cse.wustl.edu/archon/`, together with all the examples considered in this paper. Since JAVA source code may be compiled to native code using the GCJ compiler, we obtain a reasonably efficient executable. The use of a garbage-collected language eliminates the burden of implementing garbage collection in the interpreter. For this reason, functional languages are also a good choice for implementation of ARCHON.

To reduce a β-redex, a pointer (which we will call the assigned variable pointer) is set from the in-memory representation of the bound variable to the argument value. Most of the computation time in graph reduction is consumed

copying lambda abstractions. Lambda abstractions must be copied, in order to avoid setting the same assigned variable pointer in two different ways for two different applications. We will call the operation of duplicating lambda expressions *cloning*. The ARCHON interpreter uses three optimizations to reduce the amount and cost of cloning:

1. Do not clone lambda abstractions the first time they are applied in a β-reduction. Only clone them for reductions after the first one.

2. Clone lambda abstractions only when following an assigned variable pointer. This is justified because the only point in which sharing is introduced to the term graph is when the assigned variable pointer is set from a variable which is used more than once.

3. Avoid following assigned variable pointers during cloning.

Optimization 1 is easy to implement: one just sets a flag on the in-memory (possibly shared) representation of a lambda abstraction, after the first β-reduction in which it is applied. If another application of that lambda abstraction is β-reduced later, the abstraction must be cloned. Optimization 3 requires more care to implement soundly. When a lambda abstraction binding variable x is cloned, we set the assigned variable pointer to point to the new variable (call it x') which the cloned abstraction will contain. When we encounter a use of any variable y during cloning, we thus should replace it with the expression e that its assigned variable pointer points to, if that expression e is a variable. If e is not a variable, we need not replace y with e. We can dereference the pointer later, if we ever need to. The only issue is that we must be sure that no occurrence of the bound variable x we are currently cloning occurs in e (for then, failing to dereference y will leave behind occurrences of x which should have been replaced with x'). An occurrence of a bound variable in e pointed to by y can arise only during evaluation of an **open** expression, when a β-reduction operating on an open term may cause e to get assigned to y. Hence, for soundness of optimization 3, it is sufficient to dereference all assigned variables in the results of computation of **open** expressions. This is implemented in ARCHON simply by running through such a result after evaluation of an **open** expression finishes, and dereferencing all assigned variable pointers.

9.1 Effect of the Optimizations

Figure 10 shows the effect of two of the three cloning optimizations discussed above, on two benchmarks. The `eq.a` benchmark evaluates "*eq eq eq*", where *eq* is the term defined in Section 8.2 testing α-equivalence of two input terms. In this case, the benchmark is testing whether *eq* is α-equivalent to itself (it is, of course). The second benchmark is computing factorial of the number 6, expressed as a Scott-encoded unary number. All times are the averages of times for three runs of the same benchmark. The machine used has an 800 MHz INTEL PENTIUM III processor with 512 MB main memory and 512 KB

Benchmark	$-1-3$	$-1+3$	$+1-3$	$+1+3$
`eq.a`	53.1 (118,586)	6.8 (202,888)	9.1 (32,690)	3.4 (90,113)
`fact.a`	7.2 (6,563)	0.3 (13,118)	0.2 (1,639)	0.2 (4,102)

Figure 10: Effect of Optimizations on Running Time (Times in Seconds)

cache. The version of GCJ used is 3.4.4. A "+1" or "−1" in the heading indicates whether optimization 1 of the previous Section is enabled or disabled, respectively (similarly for "+3" and "−3"). Optimization 2 is not easy to disable in the implementation, and so its effect is not measured here.

The `eq.a` benchmark takes 295,146 β-reductions for ARCHON to evaluate, and `fact.a` take 12,381. The number of times an expression is cloned is given in Figure 10 in parentheses. Note that with optimization 3 turned off, there are fewer clonings, because each cloning does more work (by following all assigned variable pointers).

9.2 Comparison with Other Interpreters

We compare ARCHON with the GHCI interpreter that ships as part of the GLASGOW HASKELL COMPILER (GHC), and also with the MZSCHEME interpreter, that ships with the DRSCHEME programming environment [47, 46]. Not surprisingly for tools that have been under development for many years, these interpreters are much faster than the ARCHON interpreter. Nevertheless, the gap is not as huge as one might expect.

The simple benchmark used is squaring the factorial of 5 (as a Scott-encoded unary number). Thanks to its support for higher-rank polymorphism and recursive types, GHC can type check code operating on Scott-encoded numbers. SCHEME is untyped, and hence easily supports Scott-encoded data. The ARCHON, HASKELL, and SCHEME programs for this benchmark are given in the Appendix. Note that the implementation of multiplication and factorial used in all three languages is deliberately inefficient: the usual, more efficient implementations of both functions pass their arguments to their recursive calls in the opposite order to the one used here. (The version of factorial used for the benchmarks in Section 9.1 is the more efficient one.) The deliberate inefficiency is introduced to get a benchmark that takes long enough for GHCI and MZSCHEME to evaluate, without causing the ARCHON implementation to take too long to complete. The machine used for this experiment has a 1.2 GHz Intel Core Duo processor with 1.5 GB main memory and a 2 MB cache. Even though both MZSCHEME and GHC ship with compilation capabilities, we deliberately compare only with their interpreters. The benchmark takes 28.9 seconds to evaluate with ARCHON, 2.5 seconds with GHCI version 6.6.1, and 3.6 seconds with MZSCHEME version v371, with just-in-time compilation off. So we see ARCHON is around an order of magnitude slower than the other interpreters.

10 Conclusion

This paper has defined the ARCHON directly reflective meta-programming language. This language satisfies a number of desirable properties, including purity and scope safety. Programs are encoded using higher-order abstract syntax (since they are trivially encoded as themselves). ARCHON is also suitable for code-generating meta-programs. The language extends untyped lambda calculus with call-by-value and call-by-name abstractions, as well as novel reflective features for swapping consecutive nested lambda binders, opening lambda abstractions to compute on their bodies, comparing free variables, and decomposing arbitrary program terms. An optimized interpreter has also been presented. As we have seen, ARCHON is substantially closer to completeness than Wand's system, in which code-generating meta-programs are not (in any direct way) generally implementable. Whether or not the proposed language is actually complete is left open.

Acknowledgments. Many thanks to the anonymous reviewers of previous versions of this paper for their thorough reading and insightful criticisms. Their comments on earlier versions helped greatly improve this paper. Thanks to Walid Taha for helpful conversations on meta-programming and the ideas of the current paper. Thanks also to Matthew Bensley for contributions to the ARCHON project including help implementing several of the examples from Sections 8 and 9.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Prentice Hall, 2000.

[2] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

[3] C. Chen and H. Xi. Meta-Programming through Typeful Code Representation. *Journal of Functional Programming*, 15(6):797–835, 2005.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel Computation in Maude. In *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998.

[5] R. Constable. Using reflection to explain and enhance type theory. In *Proof and Computation, NATO ASI Series*. Springer-Verlag, 1994.

[6] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(06):567–600, 2002.

[7] H. Curry, J. Hindley, and J. Seldin. *Combinatory Logic*, volume 2. North-Holland Publishing Company, 1972.

[8] M. Davis and J. Schwartz. Metamathematical Extensibility for Theorem Verifiers and Proof-Checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[9] J. Ferber. Computational reflection in class based object-oriented languages. In *OOPSLA '89: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 317–326, New York, NY, USA, 1989. ACM Press.

[10] S. Ganz, A. Sabry, and W. Taha. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *International Conference on Functional Programming*, pages 74–85, 2001.

[11] J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and Extensible Notations for Modeling Languages. In *Fundamental Approaches to Software Engineering, FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 102–116. Springer Verlag, March 2007.

[12] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[14] J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[15] J. Harrison. The HOL Light System Reference, 2006.

[16] J. Harrison. Towards Self-verification of HOL Light. In *International Joint Conference on Automated Reasoning*, 2006.

[17] W. Hunt, M. Kaufmann, R. Krug, J Moore, and E. Smith. Meta Reasoning in ACL2. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, pages 163–178. Springer-Verlag, 2005.

[18] S. Jefferson and D. Friedman. A Simple Reflective Interpreter. In *IMSA'92 International Workshop on Reflection and Meta-level architecture*, 1992.

[19] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[20] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic, 2000.

[21] R. Kelsey, W. Clinger, J. Rees, et al. Revised^5 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[22] I.-S. Kim, K. Yi, and C. Calcagno. A Polymorphic Modal Type System for Lisp-Like Multi-Staged Languages. In *The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 257–268, 2006.

[23] J. Klop and R. de Vrijer. Examples of TRSs and Special Rewriting Formats. In TERESE, editor, *Term Rewriting Systems*, chapter 3. Cambridge University Press, 2003.

[24] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.

[25] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In S. Peyton Jones, editor, *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2006.

[26] X. Leroy, S. Blazy, and Z. Dargaye. Formal verification of a C compiler front-end. In J. Misra and T. Nipkow, editors, *Proceedings of Formal Methods*, 2006.

[27] B. Lewis, D. LaLiberte, R. Stallman, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual*. GNU Press, 2000.

[28] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[29] T. Mogensen. Efficient Self-Interpretations in lambda Calculus. *Journal of Functional Programming*, 2(3):345–363, 1992.

[30] L. Moreau. A Syntactic Theory of Dynamic Binding. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'97)*, volume 1214, pages 727–741. Springer-Verlag, apr 1997.

[31] A. Nanevski. Meta-programming with names and necessity. Technical Report CMU-CS-02-123R, Carnegie Mellon University, November 2002.

[32] A. Nanevski and F. Pfenning. Meta-programming with names and necessity. *Journal of Functional Programming*, 2005. To appear.

[33] F. Pfenning. *Logical Frameworks*, chapter 21. Volume 2 of Robinson and Voronkov [36], 2001.

[34] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[35] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[36] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier and MIT Press, 2001.

[37] H. Rueß. Computational Reflection in the Calculus of Constructions and its Application to Theorem Proving. In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, pages 319–335. Springer-Verlag, 1997.

[38] C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-Calculus. Functional Programming with Higher-Order Encodings. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, pages 339–353. Springer-Verlag, 2005.

[39] J. Seldin and J. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[40] J. Siskind and B. Pearlmutter. First-Class Nonstandard Interpretations by Opening Closures. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2007.

[41] B. Smith. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[42] G. Steele. *Common LISP: the Language (2nd ed.).* Digital Press, 1990.

[43] B. Stroustrup. *The C++ Programming Language (3rd Edition).* Addison-Wesley, 1997.

[44] W. Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute, November 1999.

[45] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.0*, 2004. http://coq.inria.fr.

[46] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.6.1*, 2007. `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[47] The PLT Group. *PLT DrScheme: Programming Environment Manual*, 2007. `http://download.plt-scheme.org/doc/drscheme/`.

[48] P. Wadler. Theorems for free! In *4th International Conference on Functional Programming and Computer Architecture*, 1989.

[49] C. Wadsworth. *Some Unusual λ-Calculus Numeral Systems*, pages 215–230. In Seldin and Hindley [39], 1980.

[50] M. Wand. The Theory of Fexprs is Trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.

[51] E. Westbrook. Free variable types. In *Seventh Symposium on Trends in Functional Programming (TFP 06)*, April 2006.

[52] A. Wright and R. Cartwright. A Practical Soft Type System for Scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, 1997.

A Haskell Code for the Factorial Benchmark

The type of Scott-encoded unary numbers may be considered to be

$$\mu N.\,(\forall b.\,(N \;\to\; b) \;\to\; b \;\to\; b) \;\to\; N$$

That is, it is a recursive type created from a functional accepting a continuation for successor (of type $N \;\to\; b$) and a continuation for zero (of type b), and computing a value. We can define this recursive type in Haskell with higher-rank polymorphism. Zero and successor numbers are then created from particular functionals. Case analysis is implemented ("`use`" below) simply by extracting the functional and applying it to the branches of the case.

```
module Ns where

data N = Mk (forall b . (N -> b) -> b -> b)
nz = Mk (\ s z -> z)
ns = \ n -> Mk (\ s z -> s n)
use = \ n s z -> case n of { Mk f -> f s z }

plus = \ n m -> use n (\p -> plus p (ns m)) m
mult = \ n m -> use n (\p -> plus (mult p m) m) nz
fact = \ n -> use n (\p -> mult n (fact p)) (ns nz)

showN :: N -> String
showN = \ n -> use n (\ p -> ("S "++(showN p))) "Z"
instance Show N where
  show = showN

test0 = fact (ns (ns (ns (ns (ns nz)))))

test = mult test0 test0
```

B Archon Code for the Factorial Benchmark

Note that the prototype ARCHON implementation uses "^ " for $\bar{\lambda}$ and "\" for λ. Also, to allow simple recursive descent parsing, application is written explicitly, in prefix notation, with "@". Finally "`$ x T1 T2`" means that we define `x` to be the value of `T1` in `T2`. Arguments in applications are sometimes printed directly below the corresponding "@" symbol.

```
$ zero ^ s ^ z z
$ succ \ n ^ s ^ z @ s n
$ one @ succ zero
$ fix \f @ \x @ f \y @ @ x x y
          \x @ f \y @ @ x x y
$ plus @ fix \ plus \ n \ m @ @ n ^ p @ @ plus p
                                    @ succ m
                              m
$ mult @ fix \ mult \ n \ m @ @ n ^ p @ @ plus @ @ mult p m m
                              zero
$ fact @ fix \ fact \ n @ @ n ^ p @ @ mult n @ fact p
                         one
$ show @ fix \ show \ n @ @ n ^ p @ S @ show p
                         Z
$ test0 @ fact @ succ @ succ @ succ @ succ @ succ zero

@ show @ @ mult test0 test0
```

C Scheme Code for the Factorial Benchmark

```
(define (nz)
  (lambda (s z) z))

(define (ns n)
  (lambda (s z) (s n)))

(define (nat-to-string n)
  (n (lambda (n) (string-append "S " (nat-to-string n))) "Z"))

(define (plus n m)
  (n (lambda (p) (plus p (ns m))) m))

(define (mult n m)
  (n (lambda (p) (plus (mult p m) m)) (nz)))

(define (fact n)
  (n (lambda (p) (mult (fact p) n)) (ns (nz))))

(define (test0)
  (fact (ns (ns (ns (ns (ns (nz))))))))

(define (test)
  (let ((x (test0))) (mult x x)))

(define (main argv)
  (print (nat-to-string (test))))
```