Reflection for the Masses*

Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt

Vrije Universiteit Brussel
{charlotte.herzeel|pascal.costanza|tjdhondt}@vub.ac.be

Abstract. A reflective programming language provides means to render explicit what is typically abstracted away in its language constructs in an on-demand style. In the early 1980's, Brian Smith introduced a general recipe for building reflective programming languages with the notion of *procedural reflection*. It is an excellent framework for understanding and comparing various metaprogramming and reflective approaches, including macro programming, first-class environments, first-class continuations, metaobject protocols, aspect-oriented programming, and so on. Unfortunately, the existing literature of Brian Smith's original account of procedural reflection is hard to understand: It is based on terminology derived from philosophy rather than computer science, and takes concepts for granted that are hard to reconstruct without intimate knowledge of historical Lisp dialects from the 1960's and 1970's. We attempt to untangle Smith's original account of procedural reflection and make it accessible to a new and wider audience. On the other hand, we then use its terminological framework to analyze other metaprogramming and reflective approaches, especially those that came afterwards.

1 Introduction

Programming languages make programming easier because they provide an *abstract* model of computers. For example, a Lisp or Smalltalk programmer does not think of computers in terms of clock cycles or transistors, but in terms of a virtual machine that understands s-expressions, and performs evaluation and function application, or understands class hierarchies, and performs message sending and dynamic dispatch. The implementation of the particular programming language then addresses the actual hardware: It is the interpreter or compiler that translates the language to the machine level.

Programming languages do not only differ in the programming models they provide (functional, object-oriented, logic-based, multi-paradigm, and so on), but also in fine-grained design choices and general implementation strategies. These differences involve abstractions that are implemented as explicit language constructs, but also "hidden" concepts that completely abstract away from certain implementation details. For example, a language may or may not abstract away memory management through automatic garbage collection, may or may

* In: R. Hirschfeld and K. Rose (Eds.): S3 2008, LNCS 5146, pp. 87–122, 2008.
© Springer-Verlag Berlin Heidelberg 2008.

not support recursion, may or may not abstract away variable lookup through lexical scoping, and so on. The implementation details of features like garbage collection, recursion and lexical scoping are not explicitly mentioned in programs and are thus said to be *absorbed* by the language [1]. Some languages absorb less and reveal more details about the internal workings of their implementation (the interpreter, the compiler or ultimately the hardware) than others.

We know that all computational problems can be expressed in *any* Turing-complete language, but absorption has consequences with regard to the way we think about and express solutions for computational problems. While some kinds of absorption are generally considered to have strong advantages, it is also obvious that some hard problems are easier to solve when one does have control over the implementation details of a language. For example, declaring *weak references* tells the otherwise invisible garbage collector to treat certain objects specially, using the *cut* operator instructs Prolog to skip choices while otherwise silently backtracking, and *first-class continuations* enable manipulating the otherwise implicit control flow in Scheme. When designing a programming language, choosing which parts of the implementation model are or are not absorbed is about finding the right balance between *generality* and *conciseness* [2], and it is hard to determine what is a good balance in the general case.

A reflective programming language provides means to render explicit what is being absorbed in an *on-demand* style. To support this, it is equipped with a model of its own implementation, and with constructs for explicitly manipulating that implementation. This allows the programmer to change the very model of the programming language from within itself! In a way, reflection strips away a layer of abstraction, bringing the programmer one step closer to the actual machine. However, there is no easy escape from the initially chosen programming model and its general implementation strategy: For example, it is hard to turn an object-oriented language into a logic language. Rather think of the programmer being able to change the fine-grained details. For example, one can define versions of an object-oriented language with single or multiple inheritance, with single or multiple dispatch, with or without specific scoping rules, and so on. In the literature, it is said that a reflective language is an entire region in the *design space of languages* rather than a single, fixed language [3].

In order to support *reflective programming*, the implementation of the programming language needs to provide a *reflective architecture*. In the beginning of the 1980's, Smith et al. introduced *procedural reflection*, which is such a reflective architecture that introduces the essential concepts for building reflective programming languages. Since the introduction of procedural reflection, many people have used, refined and extended these concepts for building their own reflective programming languages. As such, understanding procedural reflection is essential for understanding and comparing various metaprogramming and reflective approaches, including macro programming, first-class environments, first-class continuations, metaobject protocols, aspect-oriented programming, and so on. Unfortunately, the existing literature of Smith's original account of procedural reflection is hard to understand: It is based on terminology derived from

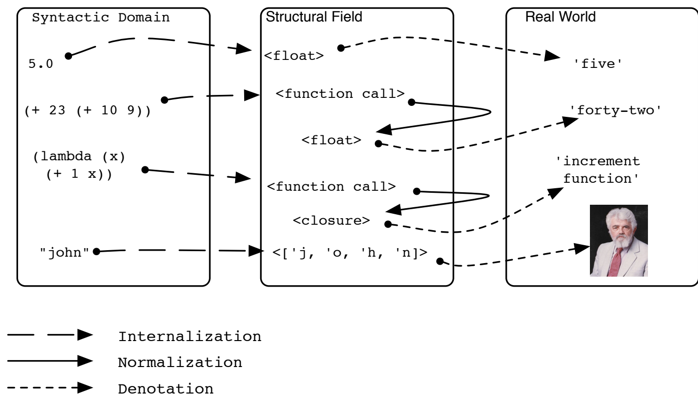

Fig. 1. Smith's theory of computation.

philosophy rather than computer science, and takes concepts known in the Lisp community in the 1960's and 1970's for granted that are hard to reconstruct without intimate knowledge of historical Lisp dialects.

In this paper, we report on our own attempt to untangle and reconstruct the original account of procedural reflection. Our approach was to actually reimplement a new interpreter for 3-Lisp, using Common Lisp as a modern implementation language. In our work, we also build upon experiences and insights that came after the introduction of procedural reflection in [1], and also add some refinements based on our own insights. Additionally we use the concepts introduced by procedural reflection to analyze various other metaprogramming and reflective approaches, especially those that came after Smith's conception of procedural reflection.

2 Self representation for a programming language

By introducing *procedural reflection*, Smith introduced a general framework for adding reflective capabilities to programming languages. In order to turn a programming language into a reflective version, one must extend the language with a model of the same language's implementation. That self representation must be *causally connected* to the language's implementation: When we manipulate the self representation, this should be translated into a manipulation of the *real* implementation. What parts of the implementation are possibly relevant to include in the model can by large be derived from mapping the non-reflective version of the programming language to Smith's theory of computation.

2.1 A model of computation

According to Smith [1], computation can be modeled as a relational mapping between three distinct domains: a *syntactic domain*, an internal representational domain (the *structural field*) and the "*real world*". The syntactic domain consists of program source code. The structural field consists of all runtime entities that make up the language implementation. In extremis, this includes the electrons and molecules of the hardware on which the interpreter runs. However, for the purpose of procedural reflection, the structural field typically consists of high-level data structures for implementing the values native to the programming language, such as numbers, characters, functions, sequences, strings, and so on. The real world consists of natural objects and abstract concepts that programmers and users want to refer to in programs, but that are typically out of reach of a computer.

Fig. 1 shows a graphical representation for computation as a mapping from programs, to their internal representation, to their meaning in the real world. The three labelled boxes represent each of the domains. The collection of arrows depict the mapping. The figure is meant to represent a model for computation rendered by a Lisp interpreter, written in C – hence the s-expressions as example elements of the syntactic domain. The `<type>` notation is used to denote an instance of a particular C type or struct. For example, the "number 5" is written as "5.0" in the programming language. Internally, it is represented by a `C` value of type `float` and it means, as to be expected, "number 5".

Each mapping between two domains has a different name. Mapping a program to its internal representation is called *internalization*, also typically called *parsing*. The mapping from one element in the structural field to another one in the structural field is called *normalization*. Normalization plays the role of *evaluation* and reduces an element in the structural field to its "simplest" form. For example, the s-expression `(+ 23 (+ 10 9))` is internalized to some structure representing a "procedure call". When that element is normalized, it is mapped to 42 – or better: To the internal representation of "42". The mapping from an internal representation to its meaning in the world is called *denotation*. In general, we cannot implement an interpreter that performs denotation. This is rather something only human beings can do. What real implementations do instead is to mimic denotation, which is called *externalization* [4]. For example, Common Lisp is built around the concept of "functions", and Common Lisp programmers do not want to be bothered by the internal implementation of functions as instances of a C struct for closures. Thus, when a function is printed, something like "<function:f>" is displayed. By limiting the kinds of operations available in Common Lisp for manipulating closure values and making sure they are printed in a way that does not reveal anything about their implementation, Common Lisp provides programmers the illusion of "real" functions. Such absorption and the provision of an "abstract" model of the hardware is the entire purpose of programming languages. By adding reflection to the language, we (purposefully) break that illusion.

2.2 Reflection

Reflection allows programmers to program as if at the level of the language's implementation. Smith distinguishes between two kinds of reflection. *Structural reflection* is about reasoning and programming over the elements in the internal domain, i.e. about inspecting and changing the internal representation of a program. On the other hand, *procedural reflection*, later also called *behavioral reflection*, is concerned with reasoning about the normalization of programs. The former allows treating programs as regular data and as such enables them to make structural changes to programs. The latter also provides access to the execution context of a program, which can then be manipulated to influence the normalization behavior of a program. Adding structural and procedural reflection to a programming language requires embedding a self representation of the language in the language, which is essentially a model of its implementation.

In relation to Smith's model of computation, the way to add structural reflection to any programming language is by extending the language with constructs for denoting and manipulating elements in the internal domain. The parts of the implementation that constitute the internal domain can be identified by looking for the structures in the implementation that implement the possible outcomes of internalization and normalization. So in other words, we should extend the language with means to denote the internal representation of numbers, characters, procedures, and so forth. Adding behavioral reflection requires providing means to influence the normalization process, e.g. by making it possible to call the interpreter or compiler from within the language.

3 Embedding a model of Lisp in Lisp

One of the most difficult things about reflection to deal with is the fact that it "messes up" the way programmers think. It does so because reflection strips away the abstract model offered by the programming language. When using reflection, the programmer is thinking in terms of the language's implementation, and no longer in terms of the programming model behind the language. In a way, learning about reflection is similarly shocking as it was finding out that Santa Claus isn't real, but that it is your parents who give you your presents. When you first found out about this, you were very upset and you resented your parents for putting up a charade like that. After a while, though, you came to realize that it was actually nice of them to introduce you to the illusion of a Santa Claus, and that it is in fact still nice. Once you know your parents are the ones giving the presents, you can even influence them to make sure you get the gifts you really want. Similarly, reflection allows you to influence the language implementation to make sure you get the programming language you really want.

In what follows, we illustrate how to turn Lisp into a reflective variant. To be more precise, we turn the "prototypical" Lisp or "ProtoLisp" into its reflective variant. Here, ProtoLisp is what we like to call the subset of Lisp presented in textbooks to learn about implementation strategies for Lisp [5].

3.1 The world in terms of ProtoLisp (externalization in ProtoLisp)

The ProtoLisp programmer thinks in terms of s-expressions, evaluation and procedure application. An s-expression constitutes a program to be evaluated by the computer. The syntax of s-expressions is parenthesized prefix notation: S-expressions look like lists of symbols. The first symbol designates an operator or *procedure*, while the rest designates the operands or *arguments*. For example, the s-expression `(+ 1 2)` has an equivalent mathematical notation `1 + 2` and evaluates to `3`. An important procedure in ProtoLisp is `lambda`, which can be used to create new procedures.[1] It takes two arguments: a list, representing the procedure's arguments, and a body, an s-expression. For example, `(lambda (x) (+ 1 x))` creates a procedure that takes one argument `x` and adds `1` to it.

There are more kinds of objects we can talk about in ProtoLisp. The categories of the kinds of such objects are the programming language's *types*. ProtoLisp's types include numbers, truth values, characters, procedures, sequences, symbols and pairs (a pair denotes a procedure call).[2] The procedure `type` can be used to get hold of the type of a particular object.

In reality, the interpreter does not know about "numbers" or "procedures" or any of those other types. The programmer only thinks that what the interpreter shows him is a number or a procedure, though in reality, it shows him something completely different, like instances of classes implementing closures, arrays of characters representing strings, and other bits and bytes. We only choose to treat them as numbers and procedures and characters and so on. A language's types, the built-in operators, the way objects are printed, is what implements externalization. Reflection provides a look at the *actual* implementation of those objects. To know what exactly constitutes the implementation of such objects, we have to delve into the implementation of the ProtoLisp interpreter.

[1] In ProtoLisp, we call `lambda` a procedure because we do not distinguish between procedures and special forms, as is traditionally done in Lisp dialects.

[2] In this paper, we use the term *type* in the tradition of dynamically typed languages, where values are tagged with runtime type information. Various Lisp dialects differ in the diversity of the types they provide.

```
(defun read-normalize-print ()
  (normalize (prompt&read) *global* (lambda (result!)
                                      (prompt&reply result!)
                                      (read-normalize-print))))

(defun prompt&read ()
  (print ">")
  (internalize (read)))
```

Fig. 2. The ProtoLisp read-eval-print loop or repl.

3.2 Internalization in ProtoLisp

ProtoLisp is an interactive language, implemented in this paper in Common Lisp using the Common Lisp Object System (CLOS). It consists of an endless read-evaluate-print-loop (repl), which repeatedly asks the programmer to type in an s-expression, performs evaluation of the s-expression and prints the result of the evaluation. The code for this is depicted in Fig. 2. In the code, we use the terminology by Smith: so *normalize* and *internalize* instead of the more traditional terms *evaluate* and *parse*. Note that the interpreter is implemented in continuation-passing style, meaning the control-flow is implemented by explicitly passing around "continuations". Continuations are implemented here as functions that take one argument. The third argument passed to `normalize` is a function that encodes the continuation of what needs to happen after an s-expression is evaluated, namely printing the result and starting the loop again.

A program is initially just a string of characters. The ProtoLisp interpreter needs to parse a program string to something structured before it can be manipulated for evaluation. This structure is what comes out of the call to `prompt&read` in Fig. 2 (the first argument to `normalize`), which calls `internalize` to create that structure. From now on we refer to instances of those classes as *internal structures*. Depending on what the program looks like, a different internal structure is created. For example, if what is read is a digit, then an instance of the class `numeral` is created, if what is read is something that starts with a left-brace `(` and closes with a right brace `)`, then an instance of the class `pair` is created, and so on. In ProtoLisp, there is a syntax specific for each of the language's types. The `internalize` function dispatches on that syntax and creates instances of the appropriate classes.
Fig. 3 aligns all of the ProtoLisp types (third table) with examples showing their notation (first table). The second table lists the CLOS classes to which those notations internalize. Note that there is no specific syntax for denoting procedures. They are created by `lambda`, as discussed in the previous section.

3.3 Normalization in ProtoLisp

The `normalize` function depicted in Fig. 4 is the heart of the ProtoLisp interpreter. It implements how to simplify an s-expression to its normal form, which is something that is self-evaluating, like a number. The `normalize` function takes three arguments, namely an *s-expression*, an *environment* and a *continuation*:

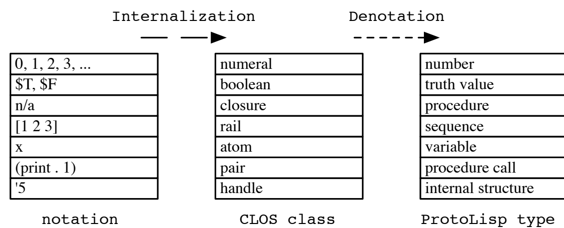

Fig. 3. Mapping notations to internal structures

- The s-expression is the ProtoLisp program to be evaluated. It is an instance of any of the classes listed in the second table of Fig. 3.
- The environment parameter is needed to keep track of the *scope* in which `normalize` is evaluating an s-expression. Environments are instances of the class `Environment` and map variables to bindings. There are two functions defined for environments: `binding` takes an environment and an atom, and returns the object bound to the atom in the given environment. `bind` takes an environment, an atom and an internal structure, and creates a binding in the environment, mapping the atom to the internal structure. The `normalize` function is initially called with a *global environment*, which is bound to the variable `*global*` and provides bindings for all primitive procedures defined in the language (like `+`).
- Since the interpreter is written in continuation-passing style, the control-flow is explicitly managed by passing around *continuations*. Continuations are implemented as Common Lisp functions that take one argument, which must be an internal structure. As an example of a continuation, consider the `lambda` form in the source code of the `read-normalize-print` function in Fig. 2, which takes one argument `result!`, prints it on the screen and calls itself. This function is the continuation for the call to `normalize` in Fig. 2, which means that the function is called by the `normalize` function instead of returning to the caller.[3]

The `normalize` function distinguishes between four cases, the branches of the conditional form `cond` in Fig. 4. Depending on the type of the s-expression being normalized, a different evaluation strategy is taken.

An s-expression *in normal form* (`normal-p`) is self-evaluating. This implies that when normalizing it, it can simply be returned. This is shown in line 2 of Fig. 4, where the continuation of `normalize` is called with an s-expression in normal form. Examples of self-evaluating s-expressions are instances of the classes `numeral` and `closure`, which implement numbers and functions.

[3] We assume that our implementation language Common Lisp supports tail recursion. Smith shows that this is ultimately not necessary, since all recursive calls are in tail position in the final 3-Lisp interpreter, resulting in a simple state machine [4].

```
;structure = numeral | boolean | closure | rail | atom | pair | handle

;normalize: structure, environment, function -> structure

01. (defun normalize (exp env cont)
02.   (cond ((normal-p exp) (funcall cont exp))
03.         ((atom-p exp)
04.           (funcall cont (binding exp env)))
05.         ((rail-p exp)
06.            (normalize-rail exp env cont))
07.         ((pair-p exp)
08.            (reduce (pcar exp) (pcdr exp) env cont))))

;normalize-rail: rail, environment, function -> rail

11. (defun normalize-rail (rail env cont)
12.   (if (empty-p rail)
13.     (funcall cont (rcons))
14.     (normalize (first rail)
15.                env
16.                (lambda (first!)
17.                  (normalize-rail (rest rail)
18.                                  env
19.                                  (lambda (rest!)
20.                                    (funcall cont (prep first! rest!))))))))

;reduce: atom, rail, environment, function -> structure

21. (defun reduce (proc args env cont)
22.   (normalize proc env
23.              (lambda (proc!)
24.                (if (lambda-p proc!)
25.                  (reduce-lambda args env cont)
26.                  (normalize args env
27.                             (lambda (args!)
28.                               (if (primitive-p proc!)
29.                                  (funcall cont (wrap (apply (unwrap proc!) (unwrap args!))))
30.                                  (normalize (body proc!)
31.                                             (bind-all (pattern proc!)
32.                                               args! (environment proc!))
33.                                             cont)))))))))

;reduce-lambda: rail, environment, function -> closure

41. (defun reduce-lambda (args env cont)
42.   (let ((argument-pattern (first args))
43.         (body (second args)))
44.     (funcall cont
45.              (make-closure
46.                :body body
47.                :argument-pattern (make-rail :contents (unwrap argument-pattern))
48.                :lexical-environment env))))
```

Fig. 4. The continuation-passing-style interpreter for ProtoLisp.

An atom, which denotes a variable (atom-p), is normalized by looking up its binding in the environment with which normalize is called, and returning this binding. Line 4 in Fig. 4 shows this: It displays a call to the continuation with the result of a call to binding as an argument. The latter searches the binding for the atom exp in the environment env.

A rail (rail-p) is normalized by normalizing all of its elements. Fig. 4 displays the source code for normalize-rail, which does this. It shows that a new rail is constructed out of the normalized elements of the original rail. empty-p is a function that checks whether a given rail has zero elements. rcons creates a new rail as an instance of the class rail. first and rest return the first element and all the other elements in a rail respectively. prep prepends an internal structure to a rail. So, for example, when one types [1 (+ 1 1) 3] in the repl, then [1 2 3] is displayed in return as the result of normalization.

A pair, denoting a procedure call (pair-p), is normalized by calling reduce[4], whose source code is also shown in Fig. 4. It is passed as arguments – besides the environment and the continuation of the call to normalize – the name of the procedure being called and the procedure call's argument list. They are obtained by calling the functions pcar and pcdr on the pair respectively (see line 8).

The source code of the reduce function, which is of course also written in continuation-passing style, is listed in Fig. 4 as well. Through a call to normalize, it looks up the binding for the procedure call's name proc (an atom). The continuation of that normalize call implements the rest of the reduce logic. It gets called with an instance of the class closure, which implements procedures (see below). If the latter closure represents the lambda procedure, checked on line 24 using the predicate lambda-p, then reduce-lambda is called, otherwise the rail of arguments is normalized, and depending on whether the closure represents a primitive procedure or not, reduce proceeds appropriately.

In case the procedure being called is a primitive procedure, as checked with primitive-p on line 28 in Fig. 4, a procedure call is interpreted by deferring it to the Common Lisp implementation. Using unwrap, the closure object bound to proc! is translated to a Common Lisp function that implements the same functionality. This function is called with the arguments of the procedure call (bound to args) after mapping them to matching Common Lisp values. The result of this call is returned after turning it into an equivalent internal structure again by means of wrap, since the ProtoLisp interpreter is defined only for such structures. The details of wrap and unwrap are discussed in a following section.

Finally, when the procedure being called is user-defined, reduce proceeds by normalizing the procedure's body with respect to the procedure's environment, extended with bindings for the procedure's variables (see lines 30–32).

reduce-lambda, also listed in Fig. 4, takes as arguments the rail of arguments of the lambda procedure call being normalized, an environment and a continuation. It creates an instance of the class closure (see make-closure on line 45), where the slots body, argument-pattern and lexical-environment are bound to a body, a rail of variables and the environment passed to reduce-lambda re-

[4] reduce is traditionally called apply.

spectively. The body is what is obtained by selecting the second element in `args`, while the argument pattern is obtained by selecting the first element from `args`. For example, when reducing the procedure call denoted by `(lambda (x) (+ x 1))`, then upon calling `reduce-lambda`, `args` will be bound to the rail denoted by `[(x) (+ x 1)]`, `body` will be bound to the rail denoted by `[(+ x 1)]` and `argument-pattern` to `[x]`.

3.4 Structural Reflection in ProtoLisp

Knowing the implementation of ProtoLisp, we are able to extend the language with reflection. With reflection we should be able to program at the level of the ProtoLisp implementation by writing ProtoLisp programs. To this end, the ProtoLisp language needs to be extended in such a way that it provides the illusion as if the implementation were written in ProtoLisp itself (and not in Common Lisp). In this section, we discuss how to extend ProtoLisp with structural reflection, while the next section discusses adding procedural reflection. ProtoLisp extended with structural reflection is dubbed *2-Lisp* by Smith.

Adding structural reflection requires two extensions: Firstly, we need to extend the ProtoLisp language with the ADTs that make up the ProtoLisp implementation. Secondly, we also need a way to get hold of the internal structure of a ProtoLisp value. The ProtoLisp implementation consists of a number of CLOS classes implementing the ProtoLisp types, and Common Lisp functions to manipulate instances of these classes. The classes are listed in the second table of Fig. 3 and the various functions, such as `normalize`, `reduce`, `pcar`, `binding` and so on, are discussed in the previous section. For example, the CLOS class `pair` and the functions `pair-p`, `pcar` and `pcdr` implement an ADT for representing procedure calls that needs to be added to the ProtoLisp language as a corresponding ProtoLisp type and corresponding ProtoLisp procedures.

Similarly to adding a `pair` type and procedures that work for pairs, we extend ProtoLisp with types and procedures that mirror the CLOS classes implementing the rest of the ProtoLisp types, like numbers, characters, procedures and so on. Some of the implementation functions we need to port to ProtoLisp require types of arguments the ProtoLisp programmer normally does not deal with. The `normalize` function, for example, takes an environment parameter. We need to add types and procedures to ProtoLisp for these classes as well.[5]

We also add `’` to ProtoLisp, which is syntax for denoting the internal structure of an s-expression that is created when the s-expression is internalized.[6] For example, `’(+ 1 1)` denotes the instance of the class `pair` that is obtained when internalizing `(+ 1 1)`. The result can be used as a pair: For example, `(pcar ’(+ 1 1))` returns `’+` and `(pcdr ’(+ 1 1))` returns `’[1 1]`.

`’` allows getting hold only of internal structures of values with corresponding syntax. So for example, `’` does not allow accessing the internal structure of a

[5] A complete listing of these procedures is beyond the scope of this paper. The interested reader is referred to the original 3-Lisp reference manual [4].

[6] `’` is called “handle”, in contrast to Common Lisp’s and Scheme’s “quote”, also abbreviated as `'`.

procedure, because a procedure cannot be created via internalization alone (see Fig. 3). We add the procedure `up` to ProtoLisp that normalizes its argument and then returns its internal structure. Thus typing in `(up (lambda (x) (+ 1 x)))` will return a closure. The `down` procedure is added to ProtoLisp to turn an internal structure obtained via `'` or `up` into a ProtoLisp value. E.g. `(down '3)` returns the "number" 3.

We can now, for example, define a procedure that prints a pair in infix notation. The definition is shown in the following session with the repl:

```
> (set print-infix (lambda (structure)
                     (if (pair-p structure)
                       (begin
                         (print "(")
                         (print-infix (1st (pcdr structure)))
                         (print (pcar structure))
                         (for-each print-infix (rest (pcdr structure)))
                         (print ")"))
                       (print (down structure)))))
<procedure:print-infix>
> (print-infix '(+ 1 1))
(1 + 1)
```

The `print-infix` procedure checks whether its argument is a pair. If it is a pair, the first argument to the procedure call is printed, followed by the procedure's name and the rest of the arguments. Otherwise, it is just printed.

Apart from inspecting internal structures, we can also modify them. For example, the following code shows how to add simple before advice to closures.

```
> (set advise-before
    (lambda (closure advice)
      (set-body closure (pcons 'begin (rcons advice (body closure))))))
> (set foo (lambda (x) (+ x x)))
> (foo 5)
10
> (advise-before (up foo) '(print "foo called"))
> (foo 5)
foo called
10
```

The `advise-before` procedure changes the body of a closure to a sequence that first executes a piece of advice and then the original closure body.

3.5 Procedural Reflection in ProtoLisp

As a final extension we add procedural reflection to ProtoLisp. This extension is dubbed *3-Lisp* by Smith. The goal of procedural reflection is to allow the programmer to influence the normalization process. To this end, ProtoLisp is extended with *reflective lambdas* to access a program's execution context, that is the interpreter's temporary state, which consists of the expression, the environment and the continuation at a particular normalization step. A reflective lambda looks like a regular procedure, with the exception that the length of its argument list is fixed: `(lambda-reflect (exp env cont) (cont (down (1st exp))))`. A reflective lambda has three parameters. When a reflective lambda call is resolved, they are bound to the expression, the environment and the continuation at that normalization step.

As a first example, consider implementing a `when` construct. This construct is similar to `if`, but `when` has only one consequential branch. The code below shows how to implement it in 3-Lisp.

```
(set when (lambda-reflect (exp env cont)
            (normalize (cons 'if (up [ (down (1st exp))
                                       (down (2nd exp))
                                       $F ]))
                       env cont)))
```

`when` is defined as a reflective lambda. When the interpreter normalizes a pair with `when` as procedure name, it transforms the pair into an `if` pair and normalizes that one instead. The body of the `when` construct consists of a call to `normalize`. The first argument is the `if` pair that is constructed out of the `when` pair, the second and the third argument are just the same environment and continuation for normalizing the `when` pair. For example, the following two expressions are equivalent:

```
(when (= (mod nr 1000) 0)
  (print "Congratulations! You win a prize."))

(if (= (mod nr 1000) 0)
  (print "Congratulations! You win a prize.")
  $F)
```

As a second example, consider one wishes to implement a `search` procedure. It takes as arguments a test and a list of elements, and returns the first occurrence in the list that satisfies the test. For example:

```
>(search number-p [ 'licensed 2 'kill ])
2
```

A procedure like `search` can easily be implemented, but consider that in doing so, we want to reuse the library procedure `for-each`, which applies a given procedure to *each* element in a given list:

```
(set for-each (lambda (f lis)
                (when (not (empty lis))
                  (begin (f (1st lis)) (for-each f (rest lis))))))
```

The code below shows an implementation of `search` in terms of `for-each`. The trick is to implement `search` as a reflective lambda: We call `for-each` with the test passed to `search`, and when this test succeeds, it calls the continuation `cont`. This is the continuation of normalizing the `search` pair: As such, we jump out of the `for-each` loop as soon as an element satisfying the test is found.

```
(set search (lambda-reflect (exp env cont)
              (normalize (2nd exp) env
                (lambda (list!)
                  (for-each (lambda (el)
                              (when ((down (binding (1st exp) env)) el)
                                (cont el)))
                            (down list!))))))
```

```
(defclass pair (handle) ...)
(defun pcar (p) ...)
(defun pcdr (p) ...)

(defvar *global* (make-environment))

(bind *global* (make-atom :name (quote pcar))
               (make-closure :ctype *primitive-tag*
                             :body (function pcar)
                             :lexical-environment *global*))
```

Fig. 5. Extending ProtoLisp with procedures mirroring the implementation functions.

3.6 Implementing reflection

Meta types & procedures Adding reflection to ProtoLisp requires extending the language with types and procedures that mirror the ADTs making up its implementation. In our implementation of ProtoLisp, the latter ADTs are implemented using CLOS classes and functions. For the better part, porting them to ProtoLisp is a matter of extending the language with new *primitive* procedures and types that simply wrap the corresponding CLOS classes and functions.

For example, in Fig. 5 we list definition skeletons of the class `pair` and its functions `pcar` and `pcdr`. The code also shows how we extend ProtoLisp's global environment with a definition for the ProtoLisp procedure `pcar`. The latter definition is implemented as a closure tagged "primitive", with a reference to the Common Lisp function `pcar` as its body. So when the procedure `pcar` is called in ProtoLisp, the interpreter recognizes it as a "primitive" closure object, defers the call to the Common Lisp implementation, and ultimately calls the Common Lisp function stored in the closure object.

Wrapping & unwrapping Recall line 29 of the `reduce` function in Fig. 4, which handles calls to primitive procedures: A Common Lisp function is fetched from the primitive closure object and then called with the provided arguments. However, it is necessary that these arguments are first mapped onto "equivalent" Common Lisp values by means of `unwrap`. For example, given an instance of the class `numeral`, `unwrap` returns the equivalent Common Lisp `number`.

When there is a one-to-one mapping between simple Common Lisp and ProtoLisp values, like between ProtoLisp numbers and Common Lisp numbers, the implementation of `unwrap` is straightforward. Unwrapping a closure which is tagged "primitive" is also straightforward: In this case, `unwrap` simply returns the Common Lisp function stored as the closure's body. Unwrapping other closures is more complicated, because their bodies contain ProtoLisp source code that Common Lisp functions cannot directly deal with, so they need to be handled specially when Common Lisp code wants to call them directly. Therefore, such closures are unwrapped by creating a special function object, which stores a reference to the original closure and a Common Lisp function handling it (see Fig. 6). Common Lisp code that wants to call such function objects needs to

```
(defmethod unwrap ((closure closure))
  (cond ((primitive-p closure) (body closure))
        ((reflective-p closure)
         (make-function :closure closure
                        :lambda (lambda (&rest args)
                                  (error "Cannot call reflective closure from CL."))))
        (t
         (make-function :closure closure
                        :lambda (lambda (args)
                                  (reduce closure
                                          (make-rail :contents (list args))
                                          *global*
                                          (lambda (result!)
                                            (prompt&reply result!)
                                            (read-normalize-print))))))))
```

Fig. 6. Unwrapping closure objects to Common Lisp functions.

invoke the stored function instead.[7] What special action is performed by such a function depends on whether the original closure is tagged as "non-reflective" or "reflective".

When `unwrap` is called with a "non-reflective" closure, we can map it to a Common Lisp function that simply invokes the ProtoLisp interpreter by calling `reduce` with the closure and the arguments received by that Common Lisp function. We also need to pass an appropriate environment and continuation to `reduce`. Since the arguments passed to the unwrapped procedure are already normalized, it does not really matter which environment to pass, so passing the global environment here is as good as any other choice. We can answer the question which continuation to pass by making the following observation: In our implementation of ProtoLisp, the only higher-order primitive procedures that take other procedures as arguments and eventually call them are `normalize` and `reduce`, which are the two main procedures of the interpreter and receive procedures as their continuation parameters. See, for example, the call to `normalize` in the definition of `search` in the previous section, which receives a non-reflective ProtoLisp procedure as a continuation. These continuation procedures are expected to call other continuations, which will ultimately end up in displaying a result in the repl and waiting for more s-expressions to evaluate, because the repl is where *any* evaluation originates from. However, to ensure that the repl is not accidentally exited by continuation procedures which simply return a value – for example when calling `(normalize '(+ 1 1) global (lambda (res) res))` – we pass a "fallback" continuation to `reduce` that simply ends up in the repl as well (see Fig. 6).[8]

We stress, however, that unwrapping "non-reflective" closures in this way is based on the assumption that the only higher-order primitive procedures which

[7] In our implementation, we have used *funcallable objects*, as provided by CLOS [3], to avoid having to update all the places in the code where functions are called.

[8] If ProtoLisp is not used as a repl, but for example as an extension language inside other applications, we have to use other "fallback" continuations here, which would simply return the value to the original call site.

call their procedure arguments are indeed `normalize` and `reduce`. If we want to provide other higher-order procedures as primitives, like for example `mapcar`, we need to pass other environments and continuations in `unwrap` as special cases. Fortunately, this is not necessary because such higher-order procedures can always also be implemented in ProtoLisp itself.[9]

As a final case in `unwrap`, we need to consider how to map closures tagged as "reflective" onto something appropriate in Common Lisp. However, there is no way we can map a reflective closure to a Common Lisp function with the same behavior because it would require that there are already similar procedurally reflective capabilities available in Common Lisp itself, which is not the case. Therefore, we just ensure that calling an unwrapped reflective closure signals an error. We can still pass reflective procedures as arguments to primitive ProtoLisp procedures, but only to eventually receive them back in ProtoLisp code again, where they have well-defined semantics. Since `normalize` and `reduce` are the only higher-order primitive procedures in ProtoLisp that actually call their procedure parameters, this is no big harm: We consider the possibility to reflect on the program text, the environment and the continuation at some arbitrary implementation-dependent place in the interpreter to be highly questionable.[10]

The inverse of the `unwrap` function is the `wrap` function, which maps a Common Lisp value to the corresponding ProtoLisp value (an internal structure). The `wrap` function is used to turn the result of interpreting a primitive procedure into a proper internal structure (see line 29 in Fig. 4). For example, given a Common Lisp number, `wrap` returns an instance of the class `numeral`. When passed a Common Lisp list, `wrap` returns a rail object. Given a Common Lisp function, `wrap` returns a closure object which is tagged "primitive". Given a function object wrapping a closure, `unwrap` returns the closure. Similarly `wrap` maps other Common Lisp values to appropriate internal structures.

Up, down & ' As discussed in the section on structural reflection, ProtoLisp is extended with `'`, `up` and `down` for denoting internal structures. The procedure `up` returns the internal structure of its argument, and is implemented as a primitive procedure (see above) that calls the function `wrap`. The procedure `down`, which returns the ProtoLisp value matching the given internal structure, is implemented as a primitive procedure that calls the function `unwrap`. `'` is syntax added for returning the result of internalizing a ProtoLisp s-expression.

`wrap` and `unwrap` also work with instances of the class `handle`. Handle objects "wrap" structures that are already internal, by just storing references to the wrapped objects. If `wrap` receives an internal structure, it just wraps it in a handle object. If `unwrap` receives an instance of the class `handle`, it returns whatever the handle object holds.

---

[9] Primitive higher-order procedures may be interesting for efficiency reasons, though.

[10] Note, however, that passing reflective procedures to the underlying implementation can be supported by a "tower" of interpreters, and is actually one motivation for the notion of reflective towers, which we discuss in the next section.

```
(defun reduce-reflective (proc! args env cont)
  (let ((non-reflective-closure (de-reflect proc!)))
    (normalize (body non-reflective-closure)
               (bind-all (lexical-environment non-reflective-closure)
                         (argument-pattern non-reflective-closure)
                         (make-rail :contents (list args env cont)))
               (lambda (result!)
                 (prompt&reply result!)
                 (read-normalize-print)))))
```

Fig. 7. Interpreting calls to `lambda-reflect`.

Lambda-reflect We also extend ProtoLisp with `lambda-reflect` to render the temporary state of the interpreter explicit. As discussed, `lambda-reflect` resembles `lambda`: When a call to `lambda-reflect` is interpreted, a closure is created that is tagged "reflective". When a call to a reflective procedure is interpreted, it is turned into an equivalent non-reflective procedure which is called instead. Furthermore, that procedure is passed the s-expression, the environment and the continuation with which the interpreter was parameterized when interpreting the reflective procedure call.

In the implementation, we extend `reduce` with a case for recognizing reflective procedure calls and passing them on to `reduce-reflective`, whose code is shown in Fig. 7. The function `de-reflect` turns a reflective closure into a regular closure by just changing its tag from "reflective" to "non-reflective". Apart from that, it has the same argument pattern, the same lexical environment and the same body as the original reflective closure. The procedure `bind-all` extends the lexical environment of the closure by mapping its argument pattern to the (unevaluated) arguments, the environment, and the continuation with which `reduce-reflective` is called. Finally, the call to `normalize` triggers evaluating the body of the closure in terms of the extended environment. The continuation passed to `normalize` is the repl's continuation from Fig. 2. When the body of the reflective procedure contains a call to the continuation it receives as an argument, then the continuation in Fig. 7 will actually never be called. However when the body of the reflective procedure does not call its continuation, then the repl continuation will be (implicitly) called. Again, this is to avoid that the repl is accidentally exited (like when unwrapping non-reflective closures).

3.7 The tower model

One of the most debated ideas of Smith's account on reflection is the tower model. In this model, 3-Lisp is implemented as an infinite stack of meta circular interpreters running one another. This solves some conceptual and technical problems that arise when implementing procedural reflection in 3-Lisp. One such problem is the following: Consider a reflective lambda where the continuation passed to it is ultimately not called. In a naive implementation of 3-Lisp, evaluating a call to such a reflective lambda would result in exiting the 3-Lisp repl and falling back to the implementation level, since the passed continuation includes the continuation of the ProtoLisp repl. Generally speaking, this is not

the desired functionality. Using an interpreter that implements the tower model, calling a reflective lambda can be resolved by the interpreter that is running the repl the programmer was interacting with at the time of the call. The only way the programmer will get back to the original interpreter is when the continuation is called inside the reflective lambda's body. When the continuation is not called, then the programmer just stays in the upper interpreter. This is referred to as "being stuck" at a meta level [4]. As such there is no danger the programmer falls back to the implementation level [6].

Problems like those can occur when there is *reflective overlap*, i.e. when a reflective language construct renders some part of the implementation explicit, but also relies on it [2]. In our example, reflective lambdas render the continuation explicit, but when it is not called inside its body, then the entire interpretation process is stopped. Tower models are generally believed to solve problems introduced by reflective overlap. However, they are not the only solution. The Scheme language, for example, is not designed as a tower, and though it introduces `call/cc` to capture a continuation, not calling it will not result in exiting Scheme, but it will just be implicitly called, no matter what. Our implementation behaves similar in that respect.

3.8 Summary – A recipe for reflection

In the previous sections, we extended ProtoLisp with reflection, which makes it possible to program at the level of the ProtoLisp implementation by writing ProtoLisp programs. The steps we took for adding reflection to ProtoLisp can be synthesized to a recipe for adding reflection to *any* programming language.

There are two parts to implementing structural reflection. First of all, one needs to identify the ADTs in the implementation that are used for representing programs, and expose them in the language itself. Secondly, one needs to equip the language with a mechanism to turn programs into first-class entities. Identifying which ADTs are potential canditates for structural reflection is done by looking at the possible outcomes of internalization (parsing) and normalization (evaluation). Note that in porting the ADTs, we need to make it appear as if they were implemented in the language itself. For adding structural reflection to ProtoLisp, this means adding new primitive types and procedures wrapping the ADTs that implement characters, numbers, procedures, and so on. The procedure `up` and the `'` syntax enable getting hold of the internal representations of programs. The procedure `down` turns programs into regular values again. Such mechanisms are implemented by means of a *wrap/unwrap* mechanism that tags internal structures in a way that ensures that the programmer can interface them using the wrapped implementation procedures.

Procedural reflection is implemented by adding mechanisms that allow the programmer to influence the normalization of a program at well-defined steps. This includes pausing the normalization, inspecting and changing the processor state at that time, and continuing the normalization. In ProtoLisp, we added `lambda-reflect`, which allows defining reflective procedures. A reflective procedure is passed the state of the processor (an expression, an environment and

a continuation), which can be manipulated as regular data inside its body. To proceed with the computation, the programmer needs to set the interpreter state by calling the `normalize` procedure. In our implementation, `lambda-reflect` is implemented as a special case in the interpreter, and `normalize` as a primitive procedure that calls its counterpart in the actual implementation.

In the next section, we give an overview of some reflective programming languages, including both historical and contemporary approaches. Such approaches are all more or less compatible with the recipe outlined above.

4 Related work

Historical overview Reflective facilities were already part of Lisp before Smith introduced the concept of procedural reflection. Lisp 1.5 [7] provides a quote mechanism that allows constructing Lisp programs on the fly at runtime. For example, the Lisp 1.5 expression `(let ((x 1)) (apply '(lambda (y) (+ x y)) (list 2)))` returns 3: A list whose first element is the symbol `lambda` is interpreted as a function, and since Lisp 1.5 is a dynamically scoped Lisp dialect, the variable references see the dynamic bindings of the respective variables even in the code constructed on the fly. (The quoted lambda expression in the example may as well be the result of a computation.) This ability to construct and execute code on the fly corresponds to Smith's notion of structural reflection, where procedures can be constructed and manipulated via `up` and `down`. Lisp 1.5 also provides the ingredients of procedural reflection: An `fexpr` is a function that gets unevaluated arguments passed as program text, and it is possible to define `fexpr`s as user programs. The `alist` provides access to the environment mapping variables to values,[11] and the "push down list" is the call stack (continuation) that can also be accessed from within Lisp programs. Taken together, these features correspond to the notion of reflective procedures in 3-Lisp. For example, such reflective features were used to introduce the concept of advice [8].

Unfortunately, dynamic scoping is problematic when it is the default semantics for Lisp. Especially it leads to the so-called "upward" and "downard funarg problems" [9]. While they can be solved by dynamic closures and spaghetti stacks to a certain degree, only the introduction of lexical closures in Scheme fully solved all aspects of the underlying issues [10]. Lexical closures got picked up in Common Lisp and most other Lisp dialects thereafter. However, lexical closures make some of the aforementioned reflective features of Lisp 1.5 less straightforward to integrate as well. 3-Lisp can be regarded as a reconceptualization of such reflective facilities in the framework of a lexically scoped Lisp dialect.

Current Lisp dialects, among which Scheme and Common Lisp are the most widely used ones, typically provide only restricted subsets of structural reflection: Scheme's `eval` and Common Lisp's `eval` and `compile` can be used to turn a quoted lambda expression into a function (similar to `down`), but they cannot be enclosed in arbitrary lexical environments, only in global or statically

[11] In Lisp 1.5, only one such environment exists.

predefined environments. There is also typically no construct corresponding to `up` available that would allow retrieving the original definition of a function. In terms of procedural reflection, neither Scheme nor Common Lisp allow defining functions that receive unevaluated arguments as program text, neither Scheme nor Common Lisp specify operators for reifying lexical environments, and only Scheme provides `call/cc` for reifying the current continuation. Macros were introduced into Lisp 1.5 in the 1960's [11], and are considered to be an acceptable and generally preferrable subset of reflecting on source code [12]. The difference in that regard to reflective procedures, `fexpr`, and so on, is that macros cannot be passed around as first-class values and are typically restricted from accessing runtime values during macro expansion. This allows compiling them away before execution in compiled systems, as is mandated for example by current Scheme and ANSI Common Lisp specifications [13, 14]. Useful applications of first-class lexical environments in Scheme have been described in the literature [15, 16], but the only Scheme implementation that seems to fully support first-class environments at the time of writing this paper is Guile, and the only Lisp implementation that seems to do so is clisp in interpreted mode.[12]

Wand and Friedman were the first to follow up on Smith's ideas, and in their research they concluded that the tower model unnecessarily introduces extra complexity to reflective programming. One of their first results is Brown, a reflective variant of Scheme, which shows that implementing procedural reflection does not require a tower architecture [17]. In their explanations, Wand and Friedman introduced the now widely used term *reification*, which is the process of turning implementation structures into first class representations. However, they differ from 3-Lisp by renouncing an explicit `up/down` mechanism. Brown instead relies on Scheme's quoting facilities as a "reification" mechanism, and the `up/down` mechanism is dismissed as a "philosophical concern". However, as we discussed in Section 3.4, this mechanism is necessary for being able to denote the internal representation of program values that cannot be identified by a string of source text alone. Hence, because the `up/down` mechanism is missing, Brown's reification mechanism is restricted: It is for example impossible in Brown to get hold of a closure. The contribution in this paper is the integration of an explicit `up/down` mechanism with a tower-less implementation of 3-Lisp, and in doing so, we introduced a correct semantics for wrapping and unwrapping reflective functions (see Section 3.6).

While solving some problems, the tower also introduces new problems. Suppose, for example, that the programmer modifies the `normalize` procedure so that its calls are logged. If the `normalize` procedure is identical for all of the interpreters in the tower, this implies that each call to `normalize` produces an infinite number of logs. To deal with this, the Blond language, based on 3-Lisp, is implemented as a tower where each interpreter (potentially) has a separate envi-

[12] See http://www.gnu.org/software/guile/guile.html and http://clisp.cons.org/. Some other Scheme implementations as well as the OpenLisp implementation of ISLISP claim to support first-class environments as well, but failed in some of our tests.

ronment [18]. Finally, Asai et al. subsequently proposed a compilation framework based on partial evaluation for implementing the tower model efficiently [19].

Analysis of reflective programming models There exist a great deal of programming languages offering reflective facilities. Depending on the kinds of reflective facilities they offer, and how these are implemented and used, we can classify these facilities as being a variant of either 2-Lisp or 3-Lisp. As we discussed previously, 2-Lisp offers the programmer structural reflection, while 3-Lisp grants procedural reflection. With structural reflection, it is possible to manipulate the internal representation of programs as if these were regular data. Using structural reflection, the programmer can make structural changes to programs, e.g. inline the code for specific function calls in a program or extend the body of a closure with code for caching. What is important to note is that structural changes to programs like these can be done without executing the program. As such, compilation techniques could be employed to implement structural reflection, which is in general believed to be more efficient than a dynamic implementation, as is required for implementing certain kinds of procedural reflection.

3-Lisp extends 2-Lisp by offering the programmer access to the execution context of a program. In 3-Lisp this execution context includes an environment and a continuation. This allows, for example, implementing one's own language constructs for exception handling or a module system as a 3-Lisp program. The execution context of a program is available only during program execution, and this requires an implementation that operates at runtime. In what follows, we give an overview of existing reflective programming languages. We briefly discuss their model and whether they offer structural or procedural reflection. We also investigate if the underlying model makes any explicit assumptions that exclude procedural reflection, e.g. for efficiency reasons.

The CLOS Metaobject Protocol (MOP) is a specification of how major building blocks of the Common Lisp Object System are implemented in terms of itself. It thus provides hooks to modify CLOS semantics by defining methods on subclasses of standard metaclasses. Such meta-level methods get invoked by conforming CLOS implementations, for example during the construction of class and generic function objects, and during slot access and method dispatch. The CLOS MOP is thus a reflective architecture, and is based on the model of procedural reflection in the sense that the hooks get triggered at runtime, while these aspects of a CLOS implementation are actually performed. As such, user-defined meta-level methods can make their results depend on runtime values. The CLOS MOP itself does not provide first-class representations of the execution context (like the current environment or the current continuation), but these could in principle be provided as orthogonal features. Later metaobject protocols tried to push the boundaries closer to structural reflection by computing more and more aspects of the object system before they get used, leading over hybrid systems like Tiny CLOS, towards load-time [20] and compile-time metaobject protocols [21], where all aspects are computed before they are ever used. At closer inspec-

tion, one can notice that the CLOS MOP is already a hybrid system, although with an emphasis on procedural elements.

The CLOS MOP can be understood as a combination of procedural reflection as in 3-Lisp together with Smalltalk's approach to object-oriented programming, where everything is an instance of a class, including classes themselves. Smalltalk's metaclasses provide a form of structural reflection, which for example allows manipulating method dictionaries, but lack meta-level protocols that can be intercepted in a procedurally reflective way (with the handling of the "message not understood" exception being a notable exception) [22]. However, Smalltalk provides first-class access to the current call stack via `thisContext`, which roughly corresponds to a combination of the environment and the continuation parameters in reflective lambdas [23]. In [24] Ducasse provides an overview of techniques, based on Smalltalk's reflective capabilities, that can be used to define a message passing control.

Self provides structural reflection via *mirrors* [25]. It can actually be argued that mirrors are a rediscovery of `up` and `down` from 2-Lisp, but put in an object-oriented setting. However, *mirrors* provide new and interesting motivations for a strict separation into internal and external representations. Especially, *mirrors* allow for multiple different internal representations of the same external object. For example, this can be interesting in distributed systems, where one internal representation may yield details of the remote reference to a remote object, while another one may yield details about the remote object itself. AmbientTalk/2 is based on mirrors as well, but extends them with *mirages* that provide a form of (object-oriented) procedural reflection [26].

Aspect-oriented programming [27] extends existing programming models with the means to "modify program join points". Depending on the aspect model at hand, program join points are defined as points in the execution of a program, or as structural program entities. In an object-oriented setting, examples of the former are "message sends" and "slot accesses", examples of the latter are classes and methods. The idea is that the programmer can make changes to program join points *without* having to change their sources, but by defining distinct program modules called "aspects". This property is called *obliviousness* and is believed to improve the quality of software in terms of better modularity.

One of the most influential aspect languages is AspectJ [28], which facilitates adding methods to classes, but also supports advising methods with logging code. Aspects are defined in terms of pointcut-advice pairs: Pointcuts are declarative queries over program join points, whereas advice consists of pieces of Java code that need to be integrated with the join points matched by a pointcut. AspectJ's pointcut language is a collection of predicates for detecting structural patterns in source code, like the names of classes or methods, where code needs to be inserted. AOP is a reflective approach in the sense that aspects are expressed as programs about programs, but unlike reflection, conventional AOP leaves out a model of the language implementation, which greatly reduces its expressiveness.

5 Conclusions

In this paper, we have given an overview of the notion of computational reflection, as introduced by Brian Smith in the early 1980's. His contributions include the notions of structural and procedural reflection, as well as a recipe for adding reflection not only to Lisp dialects, but to programming languages in general. We have a reconstructed an implementation of all the major elements of 3-Lisp, a procedurally reflective laguage, based on this recipe. We have chosen Common Lisp as the basis for our implementation, a modern Lisp dialect, which should make these contributions more accessible to a new and wider audience. We have finally used the terminological framework of computational reflection to analyze other metaprogramming and reflective approaches.

History shows that programming language designers always struggle to find the right balance between hiding language implementation details and making them accessible from within those languages. Apparently, it is inevitable that reflective features find their ways into programming languages, whether their designers are aware of this choice or not, because reflection is indeed a good compromise between hiding and revealing such details. On the other hand, designers and implementors have also always strived to minimize the cost of reflection by providing only subsets of such features. As we discussed in our overview of reflective programming models, most approaches offer only restricted subsets of structural reflection, because efficiency seems more straightforward to be achieved in this case using compilation techniques. Nevertheless, procedurally reflective features also always sneak in, but typically severely scaled down and poorly integrated with structural reflection. Due to such compromises, however, the complexity of writing reflective programs seems to increase, which in turn fortifies the view that reflection is 'dangerous' and should be provided in only small doses. Unfortunately, Smith's original account of computational reflection has been lost along the road.

We are convinced that it is time to rediscover and rethink reflection from the ground up, without any such compromises. Modern hardware is finally fast enough to effectively reduce the necessity to focus on efficiency alone. The rise in popularity of scripting languages in the last two decades shows that programmers are more and more interested in the flexibility offered by dynamicity and reflection, rather than the limiting constraints of staticity and encapsulation. To the contrary, we need more, not less reflection: For example, parallel programming models for multicore processors and distributed systems require new ways to toggle between absorbing and revealing details of language constructs, such that programs can better reason about events in the past, present and future of a computation in process. Brian Smith's notion of computational reflection is an invaluable basis to start from for these future investigations.

6 Acknowledgments

We thank Wolfgang De Meuter, Kris Gybels, Jorge Vallejos and the anonymous reviewers of S3 for their valuable comments on previous drafts of this paper.

Charlotte Herzeel is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

1. Smith, B.C.: Reflection and semantics in lisp. In: POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM (1984) 23–35
2. Steyaert, P.: Open Design of Object-Oriented Languages. PhD thesis, Vrije Universiteit Brussel (1994)
3. Kiczales, G., Rivieres, J.D., Bobrow, D.: The art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA (1991)
4. Smith, B.C., Rivieres, J.D.: Interim 3-Lisp Reference Manual. Intelligent Systems Laboratory, PALO ALTO RESEARCH CENTER. (June 1984)
5. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. second edn. MIT Press (July 1996)
6. Wand, M., Friedman, D.P.: The mystery of the tower revealed: a non-reflective description of the reflective tower. In: Proceedings of the 1986 ACM Symposium on LISP and Functional Programming. (August 1986) 298–307
7. McCarthy, J.: LISP 1.5 Programmer's Manual. The MIT Press (1962)
8. Teitelman, W.: PILOT: A Step Toward Man-Computer Symbioses. PhD thesis, Massaschusetts Institute of Technology (1966)
9. Moses, J.: The function of function in lisp or why the funarg problem should be called the environment problem. SIGSAM Bull. (15) (1970) 13–27
10. Steele, G.L., Sussman, G.J.: The art of the interpreter or, the modularity complex (parts zero, one, and two). Technical report, Cambridge, MA, USA (1978)
11. Hart, T.P.: Macro definitions for lisp. Technical report, Cambridge, MA, USA (1963)
12. Pitman, K.M.: Special forms in lisp. In: LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming, New York, NY, USA, ACM (1980)
13. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A., Kelsey, R., Clinger, W., Reese, J., Findler, R.B., Matthews, J.: Revised^{6} report on the algorithmic language Scheme (September 2007)
14. Pitman, K.M., ed.: ANSI INCITS 226-1994 (formerly ANSI X3.226:1994) American National Standard for Programming Language Common LISP is the official standard. (1994)
15. Queinnec, C., de Roure, D.: Sharing code through first-class environments. SIGPLAN Not. **31**(6) (1996) 251–261
16. Gelernter, D., Jagannathan, S., London, T.: Environments as first class objects. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM (1987) 98–110
17. Friedman, D.P., Wand, M.: Reification: Reflection without metaphysics. In: Conference Record of the 1984 ACM Symposium on LISP and Functional Programming. (August 1984) 348–355
18. Danvy, O., Malmkjær, K.: A blond primer. Technical Report DIKU Rapport 88/21, DIKU (October 1988)
19. Asai, K., Masuhara, H., Matsuoka, S., Yonezawa, A.: Partial evaluation as a compiler for reflective languages. Technical report, University of Tokyo (1995)

20. Bretthauer, H., Davis, H.E., Kopp, J., Playford, K.J.: Balancing the EuLisp Metaobject Protocol. In: Proc. of International Workshop on New Models for Software Architecture, Tokyo, Japan (1992)
21. Chiba, S.: A metaobject protocol for C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95). SIGPLAN Notices 30(10), Austin, Texas, USA (October 1995) 285–299
22. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the rescue. In: ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1998) 396–417
23. Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
24. Ducasse, S.: Evaluating message passing control techniques in smalltalk. Journal of Object-Oriented Programming (JOOP) **12**(6) (1999) 39–44
25. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2004) 331–344
26. Mostinckx, S., Cutsem, T.V., Timbermont, S., Éric Tanter: Mirages: behavioral intercession in a mirror-based architecture. In: DLS '07: Proceedings of the 2007 symposium on Dynamic languages, New York, NY, USA, ACM (2007) 89–100
27. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241., Berlin, Heidelberg, and New York, Springer-Verlag (1997) 220–242
28. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (2001) 327–353

A Source code

```
;;; config.lsp

(defpackage 3-proto-lisp
  (:use common-lisp clos)
  (:shadow boolean atom length prep first rest nth body reduce)
  (:export read-normalize-print normalize-from-string)
  (:nicknames 3pl))

#|
  Copyright (c) 2007, 2008 Charlotte Herzeel

  Permission is hereby granted, free of charge, to any person
  obtaining a copy of this software and associated documentation
  files (the "Software"), to deal in the Software without
  restriction, including without limitation the rights to use,
  copy, modify, merge, publish, distribute, sublicense, and/or
  sell copies of the Software, and to permit persons to whom the
  Software is furnished to do so, subject to the following
  conditions:

  The above copyright notice and this permission notice shall be
  included in all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
  EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
  OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
  NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
  HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
  WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
  OTHER DEALINGS IN THE SOFTWARE.
|#
```

```
;;; repl.lsp

(in-package 3-proto-lisp)

(defun read-normalize-print ()
  (declare (special *global*))
  (normalize (prompt&read)
             *global*
             (lambda (result!)
               (prompt&reply result!)
               (read-normalize-print))))

(defun prompt&read ()
  (format t ">")
  (three-lisp-read-and-parse))

(defun prompt&reply (result)
  (format t "~%~a~&" (print-to-string result)))

;; Normalize from string

;; Flag to see if in repl mode

(defparameter *repl-mode* T)

(defun normalize-from-string (string)
  (let ((*repl-mode* nil))
    (normalize (internalize (read-from-string string)) *global* (lambda (result!) result!))))

(defun create-meta-continuation ()
  (if *repl-mode*
      (lambda (result!)
        (prompt&reply result!)
        (read-normalize-print))
    (function unwrap)))

;; Loading ProtoLisp code from a file

(defun load-proto-lisp-file (filename-as-string)
  (declare (special *global*))
  (let ((path (make-pathname :name filename-as-string)))
    (with-open-file (str path :direction :input)
      (loop for line = (read str nil 'eof)
            until (eql line 'eof)
            do (normalize (internalize line) *global* (lambda (result!) result!))))))
```

```
;;; externalization.lsp

(in-package 3-proto-lisp)

;; Externalization

(defmethod external-type ((numeral numeral))
  'numeral)

(defmethod external-type ((number number))
  'number)

(defmethod external-type ((cl-boolean cl-boolean))
  'truth-value)

(defmethod external-type ((closure closure))
  'closure)

(defmethod external-type ((function function))
  'function)

(defmethod external-type ((rail rail))
  'rail)

(defmethod external-type ((wrapped-cl-list wrapped-cl-list))
  'sequence)

(defmethod external-type ((atom atom))
  'atom)

(defmethod external-type ((symbol symbol))
  'symbol)

(defmethod external-type ((pair pair))
  'pair)

(defmethod external-type ((cons cons))
  'procedure-call)

(defmethod external-type ((handle handle))
  'handle)

(defmethod print-to-string ((handle handle))
  (if (eql (class-of handle) (find-class 'handle))
      (format nil "?~a" (print-to-string (cl-value handle)))
    (format nil "~s" (cl-value handle))))

(defmethod print-to-string ((boolean boolean))
  (cond ((eql (unwrap boolean) *cl-true*) "$T")
        ((eql (unwrap boolean) *cl-false*) "$F")
        (t (error "print-to-string: Trying to print erronous boolean."))))

(defmethod print-to-string ((wrapped-cl-list wrapped-cl-list))
  (format nil "~s" (cl-list wrapped-cl-list)))

(defmethod print-to-string ((rail rail))
  (with-output-to-string (s)
    (format s "[ ")
    (loop for handle in (cl-list (unwrap rail))
          do (format s (print-to-string handle)))
    (format s " ]")))

(defmethod print-to-string (smth)
  (format nil "~a" smth))

(defmethod print-to-string ((primitive-closure primitive-closure))
  "<primitive procedure>")
```

```
(defmethod print-to-string ((reflective-closure reflective-closure))
  "<reflective procedure>")

(defmethod print-to-string ((closure closure))
  "<simple procedure>")
```

```lisp
;;; internalization.lsp

(in-package 3-proto-lisp)

;; Added syntax

;; Rails
(set-macro-character
 #\] #'(lambda (stream char)
         (declare (ignore char))
         (read stream t nil t) nil))

(set-macro-character
 #\[ #'(lambda (stream char)
         (declare (ignore char))
         (make-instance 'wrapped-cl-list :cl-list (read-delimited-list #\] stream t))))

;; Booleans

(defmethod identify-cl-boolean ((symbol (eql 'T)))
  *cl-true*)

(defmethod identify-cl-boolean ((symbol (eql 'F)))
  *cl-false*)

(defmethod identify-cl-boolean (smth)
  (error "Error while parsing ~s, $ is reserved syntax for booleans." smth))

(set-macro-character
 #\$ #'(lambda (stream char)
         (declare (ignore char))
         (identify-cl-boolean (read stream t nil t))))

;; Handle (cf ')
;; In order not to confuse Common Lisp, we use ^ instead of ' here.
(set-macro-character
 #\^ #'(lambda (stream char)
         (declare (ignore char))
         (internalize (read stream t nil t))))

;; Internalize wraps CL values to internal structures
(defmethod internalize ((handle handle))
  (make-instance 'handle :cl-value handle))

(defmethod internalize ((number number))
  (make-instance 'numeral :cl-value number))

(defmethod internalize ((cl-boolean cl-boolean))
  (make-instance 'boolean :cl-value cl-boolean))

(defmethod internalize ((wrapped-cl-list wrapped-cl-list))
  (make-instance 'rail :cl-value (make-instance 'wrapped-cl-list
                                                :cl-list (cl-list wrapped-cl-list))))

(defmethod internalize ((symbol symbol))
  (make-instance 'atom :cl-value symbol))

(defmethod internalize ((cons cons))
  (make-instance 'pair :cl-value cons))
```

```lisp
;;; normalization.lsp

(in-package 3-proto-lisp)

;; Normalization

;; Environments

(defclass environment (handle)
  ((bindings :initarg :bindings :initform '() :accessor bindings)))

(defmethod environment-p ((environment environment))
  T)

(defmethod environment-p (smth)
  nil)

(defmethod ccons ((atom atom) (environment environment) (rail rail) pair)
  (case (unwrap atom)
    (simple
     (make-instance 'closure
      :name 'anonymous :lexical-environment environment :argument-pattern rail :body pair))
    (reflect
     (make-instance 'reflective-closure
      :name 'anonymous :lexical-environment environment :argument-pattern rail :body pair))
    (t
     (error "ccons: unknown procedure type: ~s" (unwrap atom)))))

(defmethod print-to-string ((environment environment))
  "<environment>")

(defmethod make-mapping ((atom atom) value)
  (list atom value 'mapping))

(defmethod mapping-value (mapping)
  (if (and (listp mapping) (eql (car (last mapping)) 'mapping))
      (second mapping)
    (error "Mapping expected.")))

(defmethod set-binding ((environment environment) (atom atom) (handle handle))
  (push (make-mapping atom handle) (bindings environment)))

(defmethod add-binding ((environment environment) (atom atom) (handle handle))
  (make-instance 'environment
                 :bindings (cons (make-mapping atom handle) (bindings environment))))

(defmethod bind ((environment environment) (argument-pattern rail) (arguments rail))
  (cond ((empty-p argument-pattern) environment)
        ((= (length argument-pattern) (length arguments))
         (bind (add-binding environment (first argument-pattern) (first arguments))
               (rest argument-pattern) (rest arguments)))
        (t
         (error "bind: Function called with the wrong number of arguments."))))

(defmethod binding ((atom atom) (environment environment))
  (let ((mapping (find (unwrap atom) (bindings environment)
                       :key (lambda (atom+value) (unwrap (car atom+value))))))
    (if (null mapping)
        (error "binding: variable ~s unbound in env." (unwrap atom))
      (mapping-value mapping))))

;; Global environment

(defparameter *global* (make-instance 'environment))
```

```lisp
;; Normalize

(defun normalize (internal-structure environment continuation)
  (cond ((normal-p internal-structure)
         (funcall continuation internal-structure))
        ((atom-p internal-structure)
         (funcall continuation (binding internal-structure environment)))
        ((rail-p internal-structure)
         (normalize-rail internal-structure environment continuation))
        ((pair-p internal-structure)
         (reduce (pcar internal-structure) (pcdr internal-structure)
                 environment continuation))
        (t
         (error "normalize: Error trying to normalize non internal structure ~s"
                internal-structure))))

(defun normal-p (internal-structure)
  (if (rail-p internal-structure)
      (normal-rail-p internal-structure)
    (and (not (atom-p internal-structure))
         (not (pair-p internal-structure)))))

(defmethod normal-rail-p ((rail rail))
  (or (empty-p rail)
      (and (normal-p (first rail))
           (normal-rail-p (rest rail)))))

(defun normalize-rail (rail environment continuation)
  (if (empty-p rail)
      (funcall continuation (rcons))
    (normalize (first rail)
               environment
               (lambda (first!)
                 (normalize-rail (rest rail)
                                 environment
                                 (lambda (rest!)
                                   (funcall continuation (prep first! rest!))))))))


(defun reduce (procedure arguments environment continuation)
  (normalize procedure
             environment
             (lambda (procedure!)
               (cond ((reflective-p procedure!)
                      (reduce-reflective procedure! arguments environment continuation))
                     ((abnormal-p procedure!)
                      (reduce-abnormal procedure! arguments environment continuation))
                     (t
                      (normalize arguments
                                 environment
                                 (lambda (arguments!)
                                   (if (primitive-p procedure!)
                                         (funcall continuation
                                                  (wrap (apply (unwrap procedure!)
                                                               (cl-list (unwrap arguments!)))))
                                     (normalize (body procedure!)
                                                (bind (lexical-environment procedure!)
                                                      (argument-pattern procedure!) arguments!)
                                                continuation))))))))
```

```
(defun reduce-reflective (procedure! arguments environment continuation)
  (let ((non-reflective-closure (de-reflect procedure!)))
    (normalize (body non-reflective-closure)
               (bind (lexical-environment non-reflective-closure)
                     (argument-pattern non-reflective-closure)
                     (wrap (make-instance 'wrapped-cl-list
                                          :cl-list (list environment continuation arguments))))
               (create-meta-continuation))))

(defun reduce-abnormal (procedure! arguments environment continuation)
  (ecase (name procedure!)
    (set (reduce-set arguments environment continuation))
    (lambda (reduce-lambda 'closure arguments environment continuation))
    (lambda-reflect (reduce-lambda 'reflective-closure arguments environment continuation))
    (if (reduce-if arguments environment continuation))
    (apply (reduce-apply arguments environment continuation))
    (apply-abnormal (reduce-apply-abnormal arguments environment continuation))))

(defun reduce-set (arguments environment continuation)
  (declare (special *global*))
  (let ((atom (first arguments)) ;; do not normalize the symbol
        (expression (first (rest arguments)))) ;; normalize the expression
    (normalize expression environment
               (lambda (expression!)
                 (set-binding *global* atom expression!)
                 (funcall continuation (wrap 'ok))))))

(defun reduce-if (arguments environment continuation)
  (let ((condition (first arguments))
        (consequent (first (rest arguments)))
        (antesequent (first (rest (rest arguments)))))
    (normalize condition environment
               (lambda (condition!)
                 (if (cl-bool (unwrap condition!)) ;; Not Lisp-style bools
                     (normalize consequent
                                environment
                                (lambda (consequent!)
                                  (funcall continuation consequent!)))
                   (normalize antesequent
                              environment
                              (lambda (antesequent!)
                                (funcall continuation antesequent!))))))))

(defun reduce-lambda (closure-class-name arguments environment continuation)
  (let ((argument-pattern (first arguments))
        (body (first (rest arguments))))
    (funcall continuation
             (make-instance closure-class-name
                            :name 'anonymous
                            :body body
                            :argument-pattern
                            (wrap (make-instance 'wrapped-cl-list
                                                 :cl-list (unwrap argument-pattern)))
                            :lexical-environment environment))))

(defmethod de-reflect ((reflective-closure reflective-closure))
  (make-instance 'closure
                 :name (name reflective-closure)
                 :argument-pattern (argument-pattern reflective-closure)
                 :body (body reflective-closure)
                 :lexical-environment (lexical-environment reflective-closure)))

(defun reduce-apply (arguments environment continuation) ;; apply takes a symbol and a rail
  (normalize arguments environment
             (lambda (arguments!)
               (reduce (first arguments!) (unwrap (first (rest arguments!)))
                       environment continuation))))
```

```
(defun reduce-apply-abnormal (arguments environment continuation)
  (normalize arguments environment
             (lambda (arguments!)
               (reduce (first arguments!)
                       (unwrap (first (rest (rest arguments!))))
                       (unwrap (first (rest arguments!)))
                       continuation))))
```

```
;;; structural-field.lsp

(in-package 3-proto-lisp)

;; Structural field

;; Classes implementing the ProtoLisp types.
;; Instances of these classes are the internal representation of ProtoLisp values.
;; Instances of these classes are called internal structures.

(defclass handle ()
  ((cl-value :initarg :cl-value :accessor cl-value)))

;; NUMBER

(defclass numeral (handle)
  ())

;; BOOLEAN

(defclass boolean (handle)
  ())

;; CLOSURE

(defclass closure (handle)
  ((body :initarg :body :accessor body :initform nil)
   (lexical-envrionment :initarg :lexical-environment :accessor lexical-environment :initform nil)
   (name :initarg :name :accessor name :initform nil) ; for debugging
   (argument-pattern :initarg :argument-pattern :accessor argument-pattern :initform nil)))

(defclass primitive-closure (closure)
  ())

(defclass reflective-closure (closure)
  ())

;; Abnormal closures are primitive closures whose arguments are not all normalized

(defclass abnormal-closure (primitive-closure)
  ())

;; RAIL

(defclass rail (handle)
  ())

;; Helper class, for representing rails in CL

(defclass wrapped-cl-list ()
  ((cl-list :initarg :cl-list :accessor cl-list)))

;; ATOM

(defclass atom (handle)
  ())

(defclass pair (handle)
  ())

;; Operations boolean

(defmethod boolean-p ((boolean boolean))
  T)

(defmethod boolean-p (smth)
  nil)
```

```
;; Operations atom

(defmethod atom-p ((atom atom))
  T)

(defmethod atom-p (smth)
  nil)

;; Operations numeral

(defmethod numeral-p ((numeral numeral))
  T)

(defmethod numeral-p (smth)
  nil)

;; Operations pair

(defmethod pair-p ((pair pair))
  T)

(defmethod pair-p (smth)
  nil)

;; Operations closure

(defmethod primitive-p ((primitive-closure primitive-closure))
  T)

(defmethod primitive-p (smth)
  nil)

(defmethod reflective-p ((reflective-closure reflective-closure))
  T)

(defmethod reflective-p (smth)
  nil)

(defmethod abnormal-p ((abnormal-closure abnormal-closure))
  T)

(defmethod abnormal-p (smth)
  nil)

(defmethod closure-p ((closure closure))
  T)

(defmethod closure-p (smth)
  nil)

;; Operations pair

(defmethod pcar ((pair pair))
  (wrap (car (unwrap pair))))

(defmethod pcdr ((pair pair))
  ;; returns a rail
  (wrap (make-instance 'wrapped-cl-list :cl-list (cdr (unwrap pair)))))

(defmethod pcons ((handle1 handle) (handle2 handle))
  (wrap (cons (unwrap handle1) (unwrap handle2))))

;; Operations rail

(defmethod rail-p ((rail rail))
  T)
```

```
(defmethod rail-p (smth)
  nil)

(defmethod rcons (&rest list-of-structures)
  (wrap (make-instance 'wrapped-cl-list :cl-list (mapcar (function unwrap) list-of-structures))))

(defmethod scons (&rest list-of-structures)
  (make-instance 'wrapped-cl-list :cl-list list-of-structures))

(defmethod pcons ((handle handle) (rail rail))
  (wrap (cons (unwrap handle) (cl-list (unwrap rail)))))

(defmethod prep ((handle handle) (rail rail))
  (wrap (make-instance 'wrapped-cl-list :cl-list (cons (unwrap handle) (cl-list (unwrap rail))))))

(defmethod prep (smth (wrapped-cl-list wrapped-cl-list))
  (make-instance 'wrapped-cl-list :cl-list (cons smth (cl-list wrapped-cl-list))))

(defmethod length ((rail rail))
  (cl:length (cl-list (unwrap rail))))

(defmethod nth (nr (rail rail))
  (wrap (cl:nth nr (cl-list (unwrap rail)))))

(defmethod nth (nr (wrapped-cl-list wrapped-cl-list))
  (cl:nth nr (cl-list wrapped-cl-list)))

(defmethod pcar ((rail rail))
  (nth 0 rail))

(defmethod pcdr ((rail rail))
  (tail 1 rail))

(defmethod tail (nr (rail rail))
  (wrap (make-instance 'wrapped-cl-list :cl-list (cl:nthcdr nr (cl-list (unwrap rail))))))

(defmethod tail (nr (wrapped-cl-list wrapped-cl-list))
  (make-instance 'wrapped-cl-list :cl-list (cl:nthcdr nr (cl-list wrapped-cl-list))))

(defmethod first ((rail rail))
  (wrap (cl:first (cl-list (unwrap rail)))))

(defmethod rest ((rail rail))
  (wrap (make-instance 'wrapped-cl-list :cl-list (cl:rest (cl-list (unwrap rail))))))

(defmethod rest ((wrapped-cl-list wrapped-cl-list))
  (make-instance 'wrapped-cl-list :cl-list (cl:rest (cl-list wrapped-cl-list))))

(defmethod empty-p ((rail rail))
  (null (cl-list (unwrap rail))))

(defmethod empty-p ((wrapped-cl-list wrapped-cl-list))
  (null (cl-list wrapped-cl-list)))

;; Equality

(defmethod proto-lisp= ((handle1 handle) (handle2 handle))
  (proto-lisp= (unwrap handle1) (unwrap handle2)))

(defmethod proto-lisp= ((rail1 rail) (rail2 rail))
  (declare (special *cl-false*))
  *cl-false*)

(defmethod proto-lisp= ((handle1 handle) smth)
  (declare (special *cl-false*))
  *cl-false*)
```

```
(defmethod proto-lisp= (smth (handle1 handle))
  (declare (special *cl-false*))
  *cl-false*)

(defmethod proto-lisp= (smth1 smth2)
  (declare (special *cl-true* *cl-false*))
  (if (equal smth1 smth2)
      *cl-true*
    *cl-false*))

;; Internalization & Parsing

;; Reads from the standard input, these are plain CL values, which are mapped onto ProtoLisp values.
;; Most syntax of ProtoLisp overlaps with CL's syntax, for other cases there is a read macro.

(defun three-lisp-read-and-parse ()
  (let ((input (read)))
    (internalize input)))

;; Helper classes because syntax per type is unique in ProtoLisp, but not in CL.

(defclass cl-boolean ()
  ())

(defparameter *cl-true* (make-instance 'cl-boolean))
(defparameter *cl-false* (make-instance 'cl-boolean))

(defmethod cl-bool ((obj (eql *cl-true*)))
  T)

(defmethod cl-bool ((obj (eql *cl-false*)))
  nil)

(defmethod cl-bool (smth)
  (error "cl-bool: ~s is not either *cl-true* or cl-false" smth))

(defun cl->cl-bool (smth)
  (if smth
      *cl-true*
    *cl-false*))

(defmethod proto-lisp= ((smth1 wrapped-cl-list) (smth2 wrapped-cl-list))
  (if (equal (cl-list smth1) (cl-list smth2))
      *cl-true*
    *cl-false*))

;; sequence equivalent
(defmethod length ((wrapped-cl-list wrapped-cl-list))
  (cl:length (cl-list wrapped-cl-list)))

;; Wrapping and Unwrapping

(defclass cl-closure ()
  ((3l-closure :initarg :closure :reader closure))
  (:metaclass funcallable-standard-class))

(defmethod cl-closure-p ((cl-closure cl-closure))
  T)

(defmethod cl-closure-p (smth)
  nil)

(defmethod wrap ((cl-closure cl-closure))
  (closure cl-closure))

(defmethod wrap (smth)
  (internalize smth))
```

```
(defmethod wrap ((function function))
  (make-instance 'primitive-closure :cl-value function))

(defmethod unwrap ((handle handle))
  (cl-value handle))

(defmethod unwrap ((primitive-closure primitive-closure))
  (cl-value primitive-closure))

(defmethod unwrap ((reflective-closure reflective-closure))
  (let ((cl-closure (make-instance 'cl-closure :closure reflective-closure)))
    (set-funcallable-instance-function
     cl-closure
     (lambda (&rest args)
       (declare (ignore args))
       (error "Don't call reflective closures within Common Lisp code.")))
    cl-closure))

(defmethod unwrap ((closure closure))
  (declare (special *global*))
  (let ((cl-closure (make-instance 'cl-closure :closure closure)))
    (set-funcallable-instance-function
     cl-closure
     (lambda (args)
       (reduce closure
               (make-instance 'rail
                              :cl-value (make-instance 'wrapped-cl-list :cl-list (list args)))
               *global*
               (create-meta-continuation))))
    cl-closure))
```

```
;;; primitives.lsp

(in-package 3-proto-lisp)

;; Primitives

(defun set-primitive (name lambda)
  (declare (special *global*))
  (set-binding *global* (wrap name) (wrap lambda)))

(defun set-primitive-abnormal (name lambda)
  (declare (special *global*))
  (declare (special *cl-false*))
  (set-binding *global* (wrap name)
               (make-instance 'abnormal-closure
                              :body lambda :name name :cl-value (wrap *cl-false*))))

;; arithmetic
(set-primitive '+ (function +))
(set-primitive '- (function -))
(set-primitive '* (function *))
(set-primitive '/ (function /))
(set-primitive '= (function proto-lisp=))
(set-primitive '< (lambda (cl-val1 cl-val2) (cl->cl-bool (< cl-val1 cl-val2))))
(set-primitive '> (lambda (cl-val1 cl-val2) (cl->cl-bool (> cl-val1 cl-val2))))
;; printing
(set-primitive 'print (function print))
;; pair
(set-primitive 'pcar (function pcar))
(set-primitive 'pcdr (function pcdr))
(set-primitive 'pcons (function pcons))
;; rails and sequences
(set-primitive 'rcons (function rcons))
(set-primitive 'scons (function scons))
(set-primitive 'prep (function prep))
(set-primitive 'length (function length))
(set-primitive 'nth (function nth))
(set-primitive 'tail (function tail))
(set-primitive 'empty (lambda (rail) (cl->cl-bool (empty-p rail))))
;; closure
(set-primitive 'body (function body))
(set-primitive 'pattern (function argument-pattern))
(set-primitive 'environment (function lexical-environment))
(set-primitive 'ccons (function ccons))
;; atoms
(set-primitive 'acons (function gensym))
;; typing
(set-primitive 'type (function external-type))
(set-primitive 'primitive (lambda (closure) (cl->cl-bool (primitive-p closure))))
(set-primitive 'reflective (lambda (closure) (cl->cl-bool (reflective-p closure))))
;; up & down
(set-primitive 'up (function wrap))
(set-primitive 'down (function unwrap))
;; abnormal primitives, i.e. primitives whose args are not normalized
(set-primitive-abnormal
 'set
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call set")))
(set-primitive-abnormal
 'lambda
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call lambda")))
(set-primitive-abnormal
 'lambda-reflect
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call lambda-reflect")))
(set-primitive-abnormal
 'if
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call if")))
```

```
(set-primitive-abnormal
 'apply
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call apply")))
(set-primitive-abnormal
 'apply-abnormal
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call apply-abnormal")))
;; global environment
(set-primitive 'global *global*)
(set-primitive 'binding (function binding))
(set-primitive 'bind (function bind))
;; normalize
(set-primitive 'normalize (function normalize))
```