Appendix 1

An overview of the λ-calculus is given, showing how this can be used to model most mathematical and computational functions and structures. In particular, a technique is presented for modelling objects as records in which the order of fields is not significant. This requires a greater degree of subtlety than simply treating objects as tuples. A problem with Cook's record combination operator is discovered and fixed. Finally, techniques are considered for augmenting the λ-calculus with types and assignment.

A.1 λ-Calculus

The λ-calculus was developed by Alonzo Church in the 1930s as a pure calculus of functions. Along with Kleene's recursive functions and Turing's universal machine, the λ-calculus is considered one of the three fundamental models of computation.

A.1.1 λ-Abstraction

The λ-calculus is based on the notion of function abstraction and application. A term in the λ-calculus is an expression made up of functions applied to values. In the pure λ-calculus, there are no other primitive values apart from functions; instead, all values are modelled by simple λ-expressions.

Each function abstracts over a single argument, introduced by λ. The body of a function follows the "dot" and is an expression in which the function argument may occur free. Thus: λ*x.x* is the identity function, abstracting over a single argument *x* and whose body is *x*. We refer to *x* as the function's *bound*

variable; when considering larger functions of the form: $\lambda x.(y\ (z\ x))$, *y* and *z* are *free* variables in this scope. The name of the bound variable is of no consequence, except where substitutions might lead to aliasing (see below).

A.1.2 β-Reduction

The principal rule of the λ-calculus is called β-*reduction*. This is the process whereby expressions are simplified as a result of values being substituted into function arguments. Thus: $(\lambda x.x\ a)$ reduces to *a*, as a result of the substitution *a/x*. This can be thought of as applying the identity function to the value *a*, which returns *a*. More formally, we say that the argument expression *a* is substituted into the bound variable *x* of the function expression $\lambda x.x$, the λ-abstraction is discharged, yielding the body of the function expression in which all occurrences of the bound variable have been replaced by the argument expression. An important property of the λ-calculus is that all expressions are simplified to a unique irreducible normal form, no matter in which order the β-reduction steps are applied to any sub-expression. The order of evaluation of sub-expressions is of no consequence.

A.1.3 Currying

The λ-calculus can only represent single-argument functions. Multi-argument functions are transformed into equivalent nested single-argument functions; this is known as *currying* (named after H B Curry). A function which returns the first of two arguments is written: $\lambda x.\lambda y.x$ and this is understood to have the meaning: $\lambda x.(\lambda y.x)$, in other words, the body of the outermost function $\lambda x.(...)$ is the innermost function: $\lambda y.x$. The application of this function to two values *a* and *b* is illustrated using parentheses to indicate the natural order of application, $\Rightarrow$ to indicate a single β-reduction step and the style *a/x* to indicate the substitutions performed:

$$((\lambda x.\lambda y.x\ \ a)\ \ b)$$
$$a/x \Rightarrow (\lambda y.a\ \ b)$$
$$b/y \Rightarrow a.$$

In the first β-reduction step, the function binds *x* to *a* and returns: $\lambda y.a$, which is applied in the second β-reduction step to the value *b*. The effect of currying is to cause functions to bind arguments sequentially and return functional values which are a continuation of the task in hand. Expressions in the λ-calculus are β-reduced an arbitrary number of times until they reach normal form, so the above expression reduces finally to the value *a*. This behaviour is equivalent to the non-curried form of the function which binds its two arguments *x* and *y* in parallel and returns the first bound value in one evaluation step.

Since the λ-calculus is left-associative, parentheses are typically dropped, except where they are required to alter the precedence of sub-expressions. The above expression may be written as:

$(\lambda x.\lambda y.x\ \ a\ \ b) \Rightarrow a$ or even: $\lambda x.\lambda y.x\ \ a\ \ b \Rightarrow a$

A.1.4 α-Conversion

A second, minor rule of the λ-calculus is called α-conversion. This is where syntactic variables are renamed in order to avoid unintended aliasing. Without this rule, the expression: *(λx.λy.(x y) y b)* reduces inadvertently to *(b b)*. The semantic error is introduced by a simple-minded carrying out of syntactic substitutions:

$(\lambda x.\lambda y.(x\ y)\ y\ b)$
$y/x \Rightarrow (\lambda y.(y\ y)\ b)$
$b/y \Rightarrow (b\ b)$ error! the two occurrences of *y* are distinct.

The α-conversion rule checks before each β-reduction step whether the value about to be substituted is re-bound inside the function body. If so, the name of the bound variable must be changed throughout the body before substitution can take place:

$(\lambda x.\lambda y.(x\ y)\ y\ b)$
$= (\lambda x.\lambda z.(x\ z)\ y\ b)$ α-conversion step
$y/x \Rightarrow (\lambda z.(y\ z)\ b)$
$b/y \Rightarrow (y\ b)$.

and this prevents unintended aliasing of variable names. Technically, α-conversion ensures that the same variable does not occur bound and free in the same context.

A.1.5 Partial Application

A function transformed by currying has the property that it may be applied to fewer arguments than it expects and still return a meaningful value. This is known as *partial application*. A two-argument function may be applied to one or two arguments:

$(\lambda x.\lambda y.x\ \ a\ b) \Rightarrow ... \Rightarrow a$ complete application

$(\lambda x.\lambda y.x\ \ a) \Rightarrow \lambda y.a$ partial application

Partial application is a technique that is used to delay the release of information tied up in the body of a function. In the second case above, the value $\lambda y.a$ returned by the partial application is a function which, no matter to what value it is eventually applied, will always return *a*. We say that *a* is *protected* by the abstraction λy; it is clear that $\lambda y.a$ delays the release of *a*. In general, *a* may be an arbitrarily complex expression.

A.1.6 Primitive Structures

Functional forms representing primitive data structures may be created using partial application. A 2-tuple constructor for building pairs may be written as:

make-pair = λx.λy.λf.(f x y)

The symbol "=" is used above to introduce symbolic names like *make-pair*; these are merely syntactic abbreviations which may be replaced at any time by the λ-expressions for which they stand. *Make-pair* is a three-argument function, which is typically applied to two values to create pairs:

pair = (make-pair a b)
= (λx.λy.λf.(f x y) a b)
a/x ⇒ (λy.λf.(f a y) b)
b/y ⇒ λf.(f a b)

The resulting *pair* wraps up two values, *a* and *b*, which are protected by the abstraction λ*f*. The values stored in *pair* are released when the abstraction λ*f* is discharged. By careful choice of the value supplied for λ*f*, the first or second element of the pair may be selected:

first = λx.λy.x second = λx.λy.y

First and *second* are functions of two arguments which return their first or second bound value respectively. By applying *pair* to one of these functions, the first or second projection from the pair is obtained:

(pair first) = (λf.(f a b) first)
first/f ⇒ (first a b) = (λx.λy.x a b) ⇒ ... ⇒ a

(pair second) = (λf.(f a b) second)
second/f ⇒ (second a b) = (λx.λy.y a b) ⇒ ... ⇒ b

A.2 Models for Mathematics

All logical and mathematical values and operations may be modelled as terms in the λ-calculus. The choice of construction is essentially arbitrary, provided that the operations exhibit the appropriate behaviour. Values are encoded in such a way as to facilitate the simplest designs for operations.

A.2.1 Booleans and Selection

The encoding of choice for Boolean values is determined by the role they play in the selection of sub-expressions. A selection function *if* expects three arguments: a Boolean-expression, a *then*-expression and an *else*-expression:

if = λb.λt.λe.(b t e)

According to this, Boolean values are ideally represented by the first and second projection functions:

true = first = $\lambda x.\lambda y.x$ false = second = $\lambda x.\lambda y.y$

It is clear that supplying one of these values to *if* will select the appropriate *then* or *else* sub-expression from the body of *if*:

(if true then else) = ($\lambda b.\lambda t.\lambda e.$(b t e) true then else)
$\Rightarrow ... \Rightarrow$ (true then else) = ($\lambda x.\lambda y.$x then else)
$\Rightarrow ... \Rightarrow$ then

(if false then else) = ($\lambda b.\lambda t.\lambda e.$(b t e) false then else)
$\Rightarrow ... \Rightarrow$ (false then else) = ($\lambda x.\lambda y.$y then else)
$\Rightarrow ... \Rightarrow$ else

In fact, it is not even necessary to define a distinct *if* function where the Boolean values alone may be used to select sub-expressions. Given a representation for Boolean values, the standard Boolean operations may be defined:

not = $\lambda x.$(x false true) and = $\lambda x.\lambda y.$(x y false)

or = $\lambda x.\lambda y.$(x true y) implies = $\lambda x.\lambda y.$(x y true)

These functions rely on the ability of Booleans to select the appropriate sub-expression from their bodies. A simple example illustrates:

(or false true) = ($\lambda x.\lambda y.$(x true y) false true)
$\Rightarrow ... \Rightarrow$ (false true true) = ($\lambda x.\lambda y.$y true true)
$\Rightarrow ... \Rightarrow$ true

A.2.2 Natural Numbers

Different encodings for the Natural numbers have been proposed. The standard Church-Turing encoding allows addition to be defined non-recursively, but has the disadvantage that numerical representations are not unique, making it more difficult to define the notion of equality. Here, an encoding is adopted which reflects the Peano axioms directly. We assume that a distinguished value *zero* exists and that every other number can be constructed by repeated application of a successor function. S*ucc* is designed such that it wraps its argument in another λ-abstraction:

succ = $\lambda n.\lambda f.$(f n)

one = (succ zero) = ($\lambda n.\lambda f.$(f n) zero) $\Rightarrow \lambda f.$(f zero)

two = (succ one) = ($\lambda n.\lambda f.$(f n) one) $\Rightarrow \lambda f.$(f one) = $\lambda f.$(f $\lambda f.$(f zero))

three = (succ two) = ... = λf.(f λf.(f λf.(f zero)))

It should be obvious that the magnitude of a number is represented by the number of λ-abstractions surrounding zero. A predecessor function removes one level of abstraction:

pred = λn.(n λx.x)

(pred one) = (λn.(n λx.x) one) ⇒ (one λx.x)
= (λf.(f zero) λx.x) ⇒ (λx.x zero) ⇒ zero

It is clear that *pred* works simply by applying its argument, a number, to the identity function. This releases the predecessor number bound internally inside the body of any number encoding. An encoding for *zero* itself is now required. The choice of encoding is determined by the need to distinguish *zero* from all other numbers. Let us define a predicate *is-zero* which returns *false* for non-zero numbers:

is-zero = λn.(n λx.false)

(is-zero one) = (λn.(n λx.false) one)
⇒ (one λx.false) = (λf.(f zero) λx.false)
⇒ (λx.false zero) ⇒ false

The body: *(n λx.false)* of *is-zero* relies on the uniform structure of numbers to achieve its result. For all constructed numbers of the form: *n* = λ*f.(f m)* it is true that *(n λx.false)* ⇒ *(λx.false m)*, which will always bind the value *m* and return *false*. Since the structure of a number directly determines the selection of *false*, this suggests that any encoding for *zero* should select *true* in the same context. An obvious encoding for *zero* is therefore:

zero = λy.true

since this function will bind the value λ*x.false* in the body of *is-zero* and still return *true*:

(is-zero zero) = (λn.(n λx.false) zero) ⇒ (zero λx.false)
= (λy.true λx.false) ⇒ true

A.2.3 Equality and Recursion

The function *is-zero* is the only primitive predicate for numbers. Determining numerical equality is handled by removing layers of λ-abstraction from a pair of numbers until *zero* is detected. *Equal* is defined recursively:

```
equal = λx.λy.(if (is-zero x)
                  (if (is-zero y) true false)
                  (if (is-zero y) false (equal (pred x)(pred y))))
```

such that: *(equal n m)* is only *true* if recursive calls eventually terminate in the base case: *(equal zero zero)*. The occurrence of *zero* in either argument causes the function to terminate. Testing both arguments is necessary to prevent the unwanted invocation of *(pred zero)* which has no interpretation as a Natural number.

We assume the standard fixpoint approach to motivate the existence of well-defined recursive functions. Initially, the functional ϕ*equal* is defined to abstract over the point of recursion:

ϕequal = λf.λx.λy.(if (is-zero x)
(if (is-zero y) true false)
(if (is-zero y) false (f (pred x)(pred y))))

and then the recursive form of *equal* is established by application of the fixpoint finder $\mathbf{Y}$, which itself has a non-recursive definition in the λ-calculus:

equal = ($\mathbf{Y}$ ϕequal)

where $\mathbf{Y}$ = λf.(λs.(f (s s)) λs.(f (s s)))

The expansion of ($\mathbf{Y}$ ϕ*equal*) is left as an exercise for the interested reader, who may determine that this yields the recursive function *equal*.

A.2.4 Natural Arithmetic

The functions *succ* and *pred* allow a simple form of arithmetic. To handle general arithmetic, *add* may be defined in terms of *succ* and *pred*:

add = λx.λy.(if (is-zero x) y (succ (add (pred x) y)))

Add is defined recursively. It works by repeatedly applying *pred* to its first argument *x* until this reaches *zero*. As the recursion unwinds, the result is constructed by as many repeated applications of *succ* to the second argument *y*. All the familiar arithmetical operations can eventually be constructed using this approach. For example, the general multiplication case is easily constructed as a recursive function that repeatedly applies *add*:

mult = λx.λy.(if (equal x one) y (add y (mult (pred x) y)))

which is then protected against degenerate cases by incorporating tests for *zero* in either argument:

multiply = λx.λy.(if (or (is-zero x)(is-zero y)) zero (mult x y))

Subtract may be defined in the same style as *add*; *quotient* in the same style as *multiply* (making appropriate allowances for Natural arithmetic).

A.2.5 Integer Numbers

It is of greater interest to expand the number system to include Integer values. An Integer may be encoded as a pair of Naturals such that the first number in the pair represents a negative quantity and the second number a positive quantity:

-n = λf.(f n zero) +m = λf.(f zero m) 0 = λf.(f zero zero)

The advantage of this encoding is that 0 has a unique representation, unlike sign-bit representations which construct a pair from a Boolean and a Natural: the latter have distinct positive and negative versions of 0. The sign of an Integer may be tested using the predicates:

zero-pos = λz.(is-zero (z first)) zero-neg = λz.(is-zero (z second))

positive = λz.(not (zero-neg z)) negative = λz.(not (zero-pos z))

Clearly, a construction of the form: *λf.(f n m)* only has a legal Integer interpretation if at least one out of *m* or *n* is *zero*. A constructor function for Integers may be defined to preserve this as an invariant:

```
make-int = λn.λm.(if (or (is-zero n)(is-zero m))
                     (make-pair n m)
                     (make-int (pred n)(pred m)))
```

No matter what Natural values are supplied for *n* and *m*, the canonical Integer representation is constructed by recursively decrementing *n* and *m* until one is *zero*.

A.2.6 Integer Arithmetic

This constructor facilitates the development of Integer arithmetic using existing functions for Natural arithmetic. Integer *plus* and *minus* may be constructed in terms of Natural addition, with simplification:

```
plus = λx.λy.(make-int (add (x first)(y first))
                       (add (x second)(y second)))

minus = λx.λy.(make-int (add (x first)(y second))
                        (add (x second)(y first)))
```

Notice the pleasing symmetry in the two definitions. Integer *times* may be defined in a similar vein to calculate the cross-product of two pairs:

```
times = λx.λy.(make-int (add (multiply (x first)(y second))
                             (multiply (x second)(y first)))
                        (add (multiply (x first)(y first))
                             (multiply (x second)(y second))))
```

This definition performs more *multiply* operations than strictly necessary, although the unnecessary calls terminate immediately with *zero*. An alternative definition, for *integers in canonical form* (i.e. with one projection equal to *zero*) deals with sign and magnitude independently:

```
times = λx.λy.(if (same-sign x y)
                  (make-pair zero (multiply (add (x first)(x second))
                                            (add (y first)(y second))))
                  (make-pair (multiply (add (x first)(x second))
                                       (add (y first)(y second))) zero))
```

which only invokes *multiply* once, but depends on the further definition:

```
same-sign = λx.λy.(or (and (zero-pos x)(zero-pos y))
                      (and (zero-neg x)(zero-neg y)))
```

Integer *divide* may be constructed in a similar fashion, using the Natural *quotient* as its base function.

A.2.7 Further Mathematics

The pair construct may be used to model other numerical types. For example:

```
real = λf.(f exponent mantissa)
complex = λf.(f real imag)
fraction = λf.(f numerator denominator)
```

Each of these representations is amenable to the design of arithmetical functions which use more basic functions defined for simpler types. In Real arithmetic, the Integer mantissae of any two Real operands must be scaled before they are combined and the result must be scaled again to fit the normal range for a mantissa. Complex functions exhibit a symmetry in the way they act on the real and imaginary parts, which are both Reals. In particular, Complex multiplication must use the cross-product strategy illustrated above. In Fraction arithmetic, the Integer operands must be scaled to the lowest common denominator before combination and the result simplified afterwards.

A.3 Models for Data Structures

All computational data structures, constructor and accessor functions may be modelled as terms in the λ-calculus. Again, the ideal choice of encoding is usually the simplest that exhibits the appropriate behaviour.

A.3.1 Tuples and Records

A generalisation of the *pair* construct is the arbitrary *n*-tuple. Each *n*-tuple has its own *n+1* argument constructor function and *n* different *n*-place projection functions to select items. For example, a *person* record may be modelled as a 4-tuple whose fields store the surname, forename, sex and age of a person:

make-person = λw.λx.λy.λz.λf.(fw x y z)

person = (make-person smith john male 25)
⇒ λf.(f smith john male 25)

To access fields of this record, the four projection functions are given:

surname = λw.λx.λy.λz.w
forename = λw.λx.λy.λz.x
sex = λw.λx.λy.λz.y
age = λw.λx.λy.λz.z

(person forename) = (λf.(f smith john male 25) forename)
⇒ (forename smith john male 25)
= (λw.λx.λy.λz.x smith john male 25)
⇒...⇒ john

It should be obvious that the role of projection functions is to bind to the *n* values stored in an *n*-tuple and return the *jth* value. Such *n*-place functions will only project from an *n*-tuple of the same arity.

A.3.2 Linked Lists

A more flexible strategy is to store data in structures of arbitrary length and write general functions to seek a particular element. It should be immediately obvious how the *pair* construction can be used yet again to build linked lists:

cons = λh.λt.λf.(f h t)

(cons item nil) ⇒ λf.(f item nil)

(cons item2 (cons item1 nil)) ⇒...⇒ λf.(f item2 λf.(f item1 nil))

The constructor *cons* binds two values and protects them behind a further abstraction λ*f*. A list-cell is essentially a 2-tuple whose first projection is the last element stored at the head of the list and whose second projection is the tail of the list. The accessor functions *head* and *tail* may accordingly be written:

head = λm.(m first)
tail = λm.(m second)

All that remains is to find an encoding for *nil*, the empty list. A similar strategy to that chosen to encode *zero* may be followed. First, a test for the empty list is desired:

is-empty = λm.(m λx.λy.false)

The body: *(m λx.λy.false)* of *is-empty* relies on the uniform structure of list cells to achieve its result. For all constructed lists of the form: *m* = λ*f.(f h t)* it is

always true that *(m λx.λy.false)* ⇒ *(λx.λy.false h t)*, which will always bind *h* and *t*, returning *false*. This suggests that any encoding for *nil* should return *true* in the same context. An obvious encoding for *nil* is therefore:

nil = λz.true

in fact, the same value used to encode *zero*. It should be obvious that *(is-empty nil)* ⇒ *(λz.true λx.λy.false)* ⇒ *true*. Further recursive functions may be written for lists, such as *length*, which counts the number of elements, or *append*, which concatenates two lists by deconstructing the first argument and adding its elements onto the second argument. Lists may be used to model sets if *cons* is replaced by a recursive function *include* which tests if a value is present in the list before adding it.

A.3.3 Binary Trees

Other dynamic data structures, such as trees, may be constructed using this approach. Taking a 3-tuple for the basic tree-node, a constructor *make-node* for binary trees may be defined:

make-node = λv.λa.λb.λf.(f v a b)

tree = (make-node val1 (make-node val2 null null)
(make-node val3 null null))
⇒ λf.(f val1 λf.(f val2 null null) λf.(f val3 null null))

where *v* is a value stored at a given node and *a* and *b* are left- and right-subtrees. The functions to access the stored values and subtrees are defined in a manner similar to the *head* and *tail* functions for lists:

node-value = λm.(m λx.λy.λz.x)
left-subtree = λm.(m λx.λy.λz.y)
right-subtree = λm.(m λx.λy.λz.z)

where *x* binds to the value, *y* to the left-subtree and *z* to the right-subtree. Similar techniques are used to construct the test for terminal and non-terminal nodes (perhaps using a value standing for the *null* tree). Further recursive functions may be written for trees which insert values in sorted positions, or which search for values, or which traverse trees.

A.3.4 Objects as Records

The model used by Cook and others [Cook89a, CCHO89a, CCHO89b, CHC90] to represent objects makes the simplifying assumption that the most important aspect to capture is the ability of objects to respond to messages. For this reason, objects are modelled as records of methods. In the model, each object encapsulates a copy of all the functions implementing its behaviour. This would be considered wasteful in a programming language, in which objects typically only store their private state and a pointer to the table of methods shared by all

instances of a class. However, this theoretical model captures the desired style of method invocation in a most intuitive way.

Above, the similarity between method invocation and the use of projection functions to access fields in a tuple was noted:

point.x $\Leftrightarrow$ (point x)

where *point* is defined as a 2-tuple and *x* as a projection function:

point = λf.(f 3 7)

x = λa.λb.a -- *projection functions for 2-tuple*
y = λa.λb.b

This approach assumes objects will be applied to labels to release values from their body. An object is a function from a finite set of labels to values, which may be functions. In general, the functions encapsulated inside an object refer recursively to each other, requiring further abstraction over *self*:

ϕeq-point = λself.λf.(f 3 7 λp.(and (equal (self x)(p x))
(equal (self y)(p y))))

which must later be bound recursively to the object using *eq-point = (***Y** ϕ*eq-point)*. Here *eq-point* is a version of *point* with an *eq* method that tests for equality between two points, using *self*-reference to access the *x* and *y* values. Provided that the labels for *x, y* and *eq* have the form:

x = λa.λb.λc.a -- *projection functions for 3-tuple*
y = λa.λb.λc.b
eq = λa.λb.λc.c

then it is clear that: *((eq-point eq) eq-point)* has the form of a binary method and will return the value *true*. More conventionally: *eq-point.eq(eq-point)* may be written to represent the selection of the *eq* method from the object *eq-point*.

A.3.5 Problems with Records

Although it is intuitive in the λ-calculus to think of an object as a record of methods, where a record is modelled as a function from labels to encapsulated functions, the simple model presented above is inadequate for several reasons. These inadequacies were glossed over in the work of Cook and others. Here, some interesting interactions between method selection, polymorphic inheritance and the taking of fixpoints are uncovered; this merits further attention.

Above, *point* and *eq-point* both use the labels *x* and *y* to select fields. However, it is impossible to denote each label by a single, uniform projection function. This is because projection functions are tied to the size of the tuple for which

they are defined: $x = \lambda a.\lambda b.a$ defined for a 2-tuple *point* may not be used to access the *x* field of a 3-tuple *eq-point*. Applying object tuples to projection functions of the wrong arity results in meaningless constructions. Every time an object is extended, labels must be rebound to new projection functions, defined to suit the size of the tuple. This is not acceptable, since it prevents the uniform invocation of methods: the same label cannot be used to release an appropriate method from objects of different sizes.

A second more serious problem is the supposition that simple records may be combined using an operator $\oplus$. Traditionally, $f = g \oplus h$ represents function overriding, whereby *f* is defined as a new function based on *g*, whose domain is the union $dom(g) \cup dom(h)$ and whose range is $h(x)$ for all $x \in dom(g) \cap dom(h)$ and $g(x)$ otherwise. Considering *f* as a map from domain to codomain sets, this is equivalent to saying that maplets in *h* override maplets in *g* having the same domain value. This style of mathematical construction presupposes that we may reason about functions at a high level of abstraction, using meta-descriptive terms such as *domain* and *range*. In simple λ-calculus models of records, it is impossible to reason about the domain of functions representing records. This is because the set of labels representing the domain is external to the record, which is merely an ordered tuple of values. We may not combine such records and this prevents the operation of inheritance.

A.3.6 Associative Maps

It is possible, using the techniques developed for lists above, to provide an alternative model for objects as associative maps, a more subtle approach than that offered by ordered tuples. An associative map is a function from labels to values in which order is non-significant. Initially, we must store both label and value information in each map.

Using the *pair* construct introduced above, we model a single maplet as an association between a label and a value. Two inspection functions *key* and *value* return the association's label and value, respectively:

$$\text{maplet} = \lambda f.(f\ \text{lab}\ \text{val}) \Leftrightarrow \text{lab} \mapsto \text{val}$$

$$\text{key} = \lambda a.(a\ \text{first})$$
$$\text{value} = \lambda a.(a\ \text{second})$$

and associative maps are constructed as lists of maplets, such that each maplet has a unique key:

$$\text{map} = \lambda m.(m\ \lambda f.(f\ \text{lab1}\ \text{val1})\ \lambda m.(m\ \lambda f.(f\ \text{lab2}\ \text{val2})\ \text{nil})) \Leftrightarrow \{\text{lab1} \mapsto \text{val1}, \text{lab2} \mapsto \text{val2}\}$$

A recursive function is defined to search through a map for a given key. This captures the idea of reasoning about the domain of a function:

```
has-key = λm.λk.(if (is-empty m) false
                    (if (equal (key (head m)) k) true
                        (has-key (tail m) k)))
```

Here, *equal* is used to test one key against another. Keys may be modelled using any values possessing an equality test, such as the Natural numbers.

A recursive function is now defined to combine two associative maps. The first argument is the base map and the second is the extra map, the extension whose maplets should override those in the base map:

```
combine = λb.λe.(if (is-empty b) e
                    (if (has-key e (key (head b)))
                        (combine (tail b) e)
                        (cons (head b)(combine (tail b) e))))
```

The function works by deconstructing the base map and only adding its maplets to the result if the extra map possesses no maplet with the same key. All maplets from the extra map are present in the result. This captures the idea of record combination, or function overriding.

A recursive function is now defined to look up the value for a given key in a map. This function is required because associative maps are essentially unordered and maplets with the same keys may occur in different positions in different maps:

```
lookup = λm.λk.(if (equal (key (head m)) k) (value (head m))
                    (lookup (tail m) k))
```

For simplicity's sake, this function is assumed to be total over maps and keys. It is also possible to define a partial function which rejects undefined keys.

A.3.7 Method Invocation

The change in the representation of objects facilitates reasoning about the domain of functions and enables Cook-style record combination. It also opens the way to polymorphic method invocation, since the same label may be used as a key to methods in different objects and in objects of different sizes. However, such objects may not be applied directly to labels to release values, as above, because a map has the representation of a list, whose only projections are the head and tail fields. This is a serious disadvantage.

This problem may be circumvented by partially applying the lookup function to a map:

```
active-map = (lookup map)

⇒ λk.(if (equal (key (head map)) k) (value (head map))
                    (lookup (tail map) k))
```

$\Rightarrow$ λk.(if (equal lab1 k) val1
(if (equal lab2 k) val2
(lookup (tail (tail map)) k))

In so doing, a function from labels to values is created. This is exactly what is required to model objects, whose internal representation now becomes a nested set of *if*-expressions. However, nested *if*-expressions may not deconstructed and recombined like lists. There is clearly a trade-off between having a simple model for method invocation and retaining the ability to combine objects. This issue was ignored by Cook and others.

The solution given here models object definitions as ordinary associative maps until the moment actual instances are required. This allows object definitions to inherit from other object definitions, using the *combine* function to construct new definitions from base and extension maps. The map is turned into a function from keys to values by partial application of *lookup* at instance-creation time. This process integrates smoothly with Cook's restoration of object-recursion at instance creation time. We illustrate this with an object definition that is recursive:

ϕmap = λself.{lab1 $\mapsto$ val1, lab2 $\mapsto$ val2, ...}

Here, ϕ*map* is a functional abstracting over *self*, a generator for an associative map with *self*-reference. The map is a list of maplets whose values are functions which may be mutually recursive by virtue of accessing other functions through *self*. Given two map generators ϕ*map1* and ϕ*map2* in this form, these may be combined after distributing a new *self*-argument to each:

ϕmap3 = λs.(combine (ϕmap1 s)(ϕmap2 s))

since it is clear that any *(ϕmap s)* has the form of an ordinary associative map. To create an instance from the object definition ϕmap3, the recursion of *self* must first be fixed:

map3 = (**Y** ϕmap3)

and to create an object from this map, the lookup function must be partially applied:

obj3 = (lookup map3) = (lookup (**Y** ϕmap3))

The extra stage, while complicating Cook's model a little, is necessary to describe extensible objects which are applied to labels to release methods.

A.4 Extending λ-Calculus

Finally, ways in which the λ-calculus may be extended to model typed values and languages with assignment are explored. In the simply-typed λ-calculus, the existence of a basic set of types (such as Integer and Boolean) is assumed and more complex types are constructed from these. Here, the preliminaries usually involve assuming a set of typed domains satisfying a set of domain equations. Alternatively, the notion of typed values may be constructed from first principles in the untyped calculus.

A.4.1 Typed Values

To illustrate the latter, consider that a typed value may be represented by a pair constructed from a type tag and an untyped value:

typed-value = λf.(f tag val)

type = λv.(v first)
value = λv.(v second)

Inspection functions *type* and *value* access the tag and untyped value fields. Type tags may be modelled by any suitable values possessing equality, such as the Natural numbers. Equality is needed for the sake of type checking. A possible set of tags is given by:

Boolean = 0; Natural = 1; Integer = 2; ...

and typed values may be declared using:

declare = λt.λv.λf.(f t v)

int3 = (declare Integer 3) ⇒ λf.(f Integer 3)
nat3 = (declare Natural 3) ⇒ λf.(f Natural 3)

Functions may be designed to test the types of their arguments before executing their bodies. The typed function which is usually written as:

```
int-plus (x : Integer; y : Integer) : Integer
    = (plus x y);
```

is just an abbreviation for the expanded form:

```
int-plus = λx.λy.(if (and (equal (type x) Integer)(equal (type y) Integer))
                     (declare Integer (plus (value x) (value y)))
                     (declare Integer error))
```

In this way, we may protect the untyped versions of functions with type checks to prevent the unintended mixing of λ-abstractions and wrap the untyped results of functions with the appropriate type. Note here how the *error* value is also

considered to have the Integer type, for the sake of consistency. Error values are so designed as to suspend computation.

A.4.2 Type Functions

Polymorphism is introduced in the second-order λ-calculus by allowing functions which accept types as arguments, as well as ordinary typed values. An example of this is the constructor for a simple typed point. Here, the upper-case Λ is used to distinguish type abstraction from value abstraction:

make-typed-point = Λt.λ(a:t).λ(b:t).{x ↦ a, y ↦ b}

This constructor accepts a type argument *t* and builds a record of values in the type supplied:

int-point = (make-typed-point Integer 3 7)
⇒ {x ↦ 3, y ↦ 7} : {x : Integer, y : Integer}

The polymorphic constructor may also be applied partially to a type, yielding a particular typed point constructor:

make-nat-point = (make-typed-point Natural)
⇒ λ(a : Natural).λ(b : Natural).{x ↦ a, y ↦ b}

The function *make-nat-point* will only accept arguments *a* and *b* if they are Natural numbers. The workings of *make-typed-point* and its derived functions may be understood from the following expansion:

```
make-typed-point = Λt.λa.λb.(declare {x : t, y : t}
                                (if (and (equal (type a) t)(equal (type b) t))
                                    {x ↦ (value a), y ↦ (value b)}
                                    error))
```

The result is either a well-formed record or an error in the type {x : t, y : t}. The constructor tests that the value-arguments are in the same given type *t*, then constructs an untyped record which is paired with a record type. Record access functions must obtain both the types and values of the fields they select.

A.4.3 Modelling Assignment

Earlier, the λ-calculus was extended with assignment, in order to model objects with mutable states. Assignment is not strictly part of a pure functional language. However the effect of assignment may be approximated using a global set of variable bindings, called the *environment*, which is passed from function to function. The environment is an associative map from variable names to their bound values.

During program execution, certain statements may update the global environment. Modifications do not literally change the state of the environment, since we are working in a pure functional language without side-effects; instead, they construct new environments in which appropriate changes have been made. The environment must therefore be passed in and out of each function, since assignments may occur at any point and their effect must be recorded in the caller. Every function accepts the environment as an extra first argument. Likewise, every function returns a packaged result, which is a pair of the environment and the function's usual return value. The caller of a function must unpack the returned result in order to update its own environment and access the ordinary return value.

Initially, the environment contains maplets representing the bindings of global program variables. The environment may grow or shrink in size, as a consequence of functions which *add* or *remove* maplets. Maplets are added every time new local variables come into scope; they are removed when local variables go out of scope. The maplets in environments do not have unique keys; this allows an *add* operation to include at the head of an environment a maplet which temporarily masks an existing maplet, which has the same key, further down in the list. The *assign* function replaces the value stored against the first occurrence of a given key. The *remove* function removes the first occurrence of a maplet with a given key. In this way, assignments affect only the most locally scoped version of a variable and the scope of the calling-context is restored when local variables go out of scope.

On entry to a function, the environment must first be extended with any new local variable declarations, suitably initialised, then the body of the function executes. In the function body, access to a variable is modelled by looking up the value stored against the first occurrence of a given key in the environment. This ensures that the most local binding is used. Repeated assignment to a variable is modelled using override operations which construct new associative maps in which the value stored against the first occurrence of the key is replaced. The sequential effect of such assignments must be modelled by nesting the operations that update environments in the desired order. When a function terminates, any maplets representing local variable bindings must be removed from the environment before this is packaged with the usual return value.

Functions typically call sub-functions. At the call-site, the current environment is passed into the sub-function. During its execution, a sub-function may update its copy of the environment, which is different from that held in the caller. When the sub-function returns, the caller unpacks the sub-function's result and its copy of the environment. The caller must now update its own copy of the environment to reflect the changes made in the sub-function. This is done using the *combine* operation for associative maps, which overrides maplets in the caller's environment with those in the sub-function's environment. A necessary consequence of this is that all sub-functions must be called in an explicit sequence, determined by the nesting of operations to update the caller's environment each time a sub-function returns.

To model operations on environments, our existing *lookup* function may be used to perform *env-access*. The new functions *env-add, env-remove* and *env-assign* are defined below:

```
env-access = lookup

env-add = λm.λk.λv.(cons (make-pair k v) m)

env-remove = λm.λk.(if (is-empty m) m
                       (if (equal (key (head m)) k) (tail m)
                           (cons (head m) (env-remove (tail m) k))))

env-assign = λm.λk.λv.(if (is-empty m) m
                          (if (equal (key (head m)) k)
                              (cons (make-pair k v) (tail m))
                              (cons (head m) (env-assign (tail m) k v))))
```

A.4.4 Epilogue

This appendix has presented an overview of constructions in the λ-calculus. A solution has been developed to certain technical problems preventing the simultaneous successful operation of Cook's record combination and the application of objects to labels. For this, techniques were presented for modelling list data structures in the λ-calculus, in particular associative maps. The introduction of associative maps also coincided with the notion of environments, used to explain how assignment may be handled in a pure functional language.