Ordinal Arithmetic with List Structures (Preliminary Version)

Nachum Dershowitz*
Edward M. Reingold

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue
Urbana, Illinois 61801, USA
e-mail: {nachum,reingold}@cs.uiuc.edu

Abstract

We provide a set of "natural" requirements for well-orderings of (binary) list structures. We show that the resultant order-type is the successor of the first critical epsilon number.

The checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In this problem the ordinal might be $(n-r)\omega^2+(r-s)\omega+k$. *A less highbrow form of the same thing would be to give the integer* $2^{80}(n-r)+2^{40}(r-s)+k$.

—Alan M. Turing (1949)

1 Introduction

A riddle—consider the Lisp-like function f,

$$
\begin{aligned}
f(a) &= a \\
f(b) &= b \\
f(cons(x,y)) &= \begin{cases}
a & \text{if } x \equiv y \equiv a, \\
cons(cons(\ldots cons(b, f(y)), y) \ldots, y) & \text{if } x \equiv b \text{ and } y \not\equiv b, \\
cons(x, cons(x, \ldots cons(x, cons(f(x), b) \ldots))) & \text{if } y \equiv b, \\
cons(f(x), cons(f(x), \ldots cons(f(x), a) \ldots)) & \text{if } x \not\equiv a, b \text{ and } y \equiv a, \\
cons(x, f(y)) & \text{otherwise.}
\end{cases}
\end{aligned}
$$

that maps binary trees with leaves labeled a or b to themselves. Ellipses represent repetitions of *arbitrary* length, so f is actually a multivalued function. Question: Is there any expression z over a, b, and *cons*, such that $z, f(z), f(f(z)), f(f(f(z))), \ldots$ is an infinite sequence, or must every such sequence $\{f^{(n)}(z)\}_n$ end in all as or bs? This function is depicted in Figure 1, where we use bullets ($\bullet$) for internal nodes ("cons cells") and squares for leaves (atoms).

The surprising answer is that no other infinite sequences are possible.

*Research supported in part by the National Science Foundation under Grants CCR-90-07195 and CCR-90-24271.

In general, such questions can be answered by using the notion of well-ordering, stemming from the fundamental work of Cantor [1915]. Floyd, in his landmark paper [1967], envisioned proving termination of programs by showing that some ordinal-valued function decreases strictly with each repetition of a loop, as did Turing before him (see the quotation above). The well-ordering most commonly used is ω, the natural ordering of the natural numbers [Dijkstra, 1976; Gries, 1981], but lexicographic orderings (ω^n) also play an important part [Manna, 1974]. Occasionally, "larger" orderings have been used (for example, [Dershowitz and Manna, 1979; Dershowitz, 1987]); see [Dershowitz, 1987; Dershowitz and Okada, 1988; Cichon, 1990].

The riddle above is a termination question on binary trees, one of the most pervasive data structures used in computer science. Like numbers, binary trees can be well-ordered in many ways. In this paper, we give "natural" principles that such orderings ought to satisfy. We consider infinite binary trees, and show how a "regular" subclass—the trees representable as list structures in Lisp—more than suffice for all ordinals up to $\epsilon_{\epsilon_{\epsilon_{\cdots}}} + 1$, where $\epsilon_{\epsilon_{\epsilon_{\cdots}}}$ is the first critical epsilon number. (Different notions of "naturalness" of ordinal notations are surveyed in [Crossley and Kister, 1986/1987].) Conversely, ordinals up to and *including* $\epsilon_{\epsilon_{\epsilon_{\cdots}}}$ can be neatly represented by this subclass of infinite binary trees.

In the next section, we consider natural orderings on binary trees, and some (known) consequences of those principles for finite trees. By imposing a lexicographic rule, we get—not surprisingly—an ϵ_0 ordering. Then, in Section 3, we present our main results, the extension of the natural ordering to arbitrary list structures, which correspond to the "rational" subset [Courcelle, 1983] of infinite binary trees. We show that $\epsilon_{\epsilon_{\epsilon_{\cdots}}} + 1$ can be proved well-ordered by the Homeomorphic Embedding Theorem on infinite trees. Section 4 mentions related work on orderings of (finite) ordered trees, leading to orderings of type Γ_0, the first impredicative ordinal; the last section includes a few remarks on implications for program verification.

Nonempty lists are built from "cons" cells $cons(x, y)$ containing two pointers, x and y; pointers may point either to the empty list *nil* or to a cons cell. We use $|l|$ for size of a list structure l, that is, the number of cons cells and nil pointers in l Thus, for example, $|\, nil\,| = 1$, $|\, cons(nil, nil)| = 3$, and $|z| = 2$, when $z \equiv cons(nil, z)$.

The orderings we deal with are really quasi-orderings; that is, they are not anti-symmetric. For a quasi-ordering $\geq$, we use $\simeq$ for the intersection of $\geq$ and its inverse $\leq$; the strict ordering $>$ is $\geq \cap \not\simeq$. We use $\equiv$ for structural equality, and $\not\equiv$ for its complement.

2 Small Ordinals

The ordering principles we propose apply equally well to cyclic and acyclic list structures. We begin, therefore, with the more mundane, acyclic variety—that is, with finite binary trees.

2.1 Axioms of Ordering

Principle 1 (Growth). *A tree is greater than or equivalent to its subtrees; that is,*

$$cons(x, y) \geq x, y,$$

for all trees x, y.

Principle 2 (Monotonicity). *Replacing a subtree by a greater or equivalent one results in a greater or equivalent tree; that is,*

$$x \geq y \Rightarrow \begin{cases} cons(x, z) \geq cons(y, z) \\ cons(z, x) \geq cons(z, y), \end{cases}$$

for all trees x, y, z.

Okada and Steele [1988] relate any ordering on finite trees satisfying such principles to Ackermann's ordinal notation.

By "deleting" in a tree, we mean replacing a subtree by one of its subtrees; "inserting" is the inverse operation.

Lemma 1. *Deleting (inserting) results in a smaller (greater) or equivalent tree.*

Proof. Follows from Growth and Monotonicity. □

So, if t_1 is homeomorphically embedded in t_2, then $t_1 \leq t_2$, where $\leq$ is any ordering satisfying Principles 1 and 2. (A tree t is *homeomorphically embedded* in a tree t' if there's a mapping of nodes of t_1 into nodes of t_2 such that each edge of t_1 corresponds to a disjoint path in t_2.)

Monotonicity implies that if $x' \geq x$ and $y' \geq y$, then $cons(x', y') \geq cons(x, y)$. What, however should the ordering of $cons(x, y)$ and $cons(x', y')$ be when $x' > x$ and $y > y'$? We choose a lexicographic rule in which "left" is more significant than "right". Note, however, that Lemma 1 implies that $cons(x', y') < cons(x, y)$ whenever $y > cons(x', y')$. So, we can't just say that $x' > x$ implies $cons(x', y') \geq cons(x, y)$. Hence, the following lexicographic principle is the strongest that can be formulated without violating our prior principles.

Principle 3 (Lexicography). *If $x' > x$ and $cons(x', y') \geq y$, then $cons(x', y') \geq cons(x, y)$.*

Let $\geq$ be a minimal ordering satisfying Principles 1, 2, and 3. (A "minimal" ordering is one that, if any pair $s \geq t$ is removed from the ordering, violates one of the principles.)

Theorem 1. *The ordering $\geq$ is total; that is $t_1 \geq t_2$, or $t_2 \geq t_1$, or both. Specifically,*

$$cons(x', y') \geq cons(x, y) \text{ if and only if } \begin{cases} y' \geq y & \text{if } x' \simeq x, \\ cons(x', y') \geq y & \text{if } x' > x, \\ y' > cons(x, y) & \text{if } x' < x. \end{cases}$$

Proof. By induction on size of the trees, this definition—combined with the fact that the empty tree, *nil*, is comparable with all trees (by virtue of the Growth Principle)—gives a total ordering. This ordering clearly satisfies the principles. Furthermore, any ordering satisfying the principles must satisfy the "if" direction, the first case of which follows from Monotonicity; the second, from Lexicography; and the third, from the Growth Principle. □

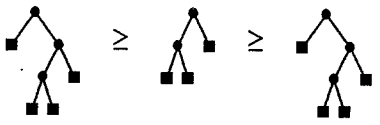
Lemma 2. *For any trees x and y, $cons(x, y) > nil$.*

Proof. Making $cons(x, y) \geq nil \not\geq cons(x, y)$ still gives an ordering satisfying the principles. □

Theorem 2. *Tree comparison of finite trees t_1 and t_2 can be done in time $O(|t_1| \times |t_2|)$.*

Proof. Follows from Theorem 1, Lemma 2, and induction on $|t_1|$ and $|t_2|$. □

The ordering $\geq$ is actually a quasi-ordering, for



because, in general,

Lemma 3. *If $x < y$, then $cons(x, cons(y, z)) \simeq cons(y, z)$.*

Proof. The inequality $cons(x, cons(y, z)) \geq cons(y, z)$ follows from the Growth Principle; the other direction follows from Lexicography, using Lemma 2. □

2.2 Order-Preserving Mapping

One can map finite binary trees, under the given ordering, to ordinals below ϵ_0 in the following straightforward way:

Proposition 1. *There is an order-preserving mapping from trees under $\geq$ to the ordinals up to ϵ_0:*

$$\begin{aligned} [\![nil]\!] &= 0 \\ [\![cons(x,y)]\!] &= \omega^{[\![x]\!]} + [\![y]\!] \end{aligned}$$

In other words, lists $(l_1, \ldots, l_n)$ are interpreted as the noncommutative sum $\omega^{[\![l_1]\!]} + \cdots + \omega^{[\![l_n]\!]}$.

This mapping is not one-to-one; as we just saw, there are equivalent, non-isomorphic trees. It is order-preserving. This means that for two finite binary trees t and t', $t \geq t'$ if and only if $[\![t]\!] \geq [\![t']\!]$. Furthermore, there is a one-to-one correspondence between binary trees and expressions involving (non-commutative) addition and exponentiation. Since such expressions give all ordinals below ϵ_0, our ordering is of order-type ϵ_0, too. Thus, expressions in Cantor Normal Form are in one-to-one correspondence with the equivalence classes on binary trees imposed by $\simeq$.

2.3 Embedding Theorem

As a special case of Higman's Lemma [Higman, 1952], we know that, in any infinite sequence $\{t_i\}_{i<\omega}$ of finite binary trees, there must be two trees t_j and t_k $(j < k)$ such that t_j is homeomorphically embedded in t_k. In other words, t_k can be obtained from t_j by deletion only. By Lemma 1, it follows that $t_j \leq t_k$; hence, an infinite descending sequence of trees is impossible. In other words, our ordering is well-founded. We have already seen that $\leq$ is order-isomorphic to ϵ_0. Since ϵ_0 induction is equivalent to the consistency of Peano Arithmetic, this means that the Embedding Lemma of Higman cannot be proved in Peano Arithmetic [Friedman, 19??].

2.4 Arithmetic

The mapping from ordinals to binary trees gives a convenient data structure for representing ordinals below ϵ_0. Arithmetic operations (commutative addition $\oplus$, commutative multiplication $\otimes$, and exponentiation), and a predecessor operation to get fundamental sequences, are now easy to define; the following correspondences are suggestive:

$$\begin{aligned}
0 &\mapsto nil \\
1 &\mapsto cons(nil, nil) \\
x \oplus nil &\mapsto x \\
cons(x,y) \oplus cons(x',y') &\mapsto cons(x, y \oplus cons(x',y')) && \text{if } x \leq x' \\
x \otimes nil &\mapsto nil \\
cons(x, nil) \otimes cons(x',y') &\mapsto cons(x \oplus x', cons(x, nil) \otimes y') \\
cons(x,y) \otimes z &\mapsto (cons(x, nil) \otimes z) \oplus (y \otimes z) \\
\omega^x &\mapsto cons(x, nil) \\
pred_n(cons(nil, nil)) &\mapsto nil \\
pred_n(cons(x, nil)) &\mapsto cons(pred_n(x), nil) \otimes n && \text{if } x \text{ is a successor ordinal} \\
pred_n(cons(x, nil)) &\mapsto cons(pred_n(x), nil) && \text{if } x \text{ is a limit ordinal} \\
pred_n(cons(x,y)) &\mapsto cons(x, pred_n(y)) && \text{if } y \not\equiv nil
\end{aligned}$$

For example, this binary-tree data structure could be used in implementing the computation of the various extensions of Ackermann's function (see, for example, [Ketonen and Solovay, 1981]). An ordinal-indexed function $A_\alpha(n)$ can be defined for ordinals α and natural numbers n by

$$A_\alpha(n) = \begin{cases} 2n & \text{if } \alpha = 0, n \geq 1, \\ A_\beta^{(n)}(1) & \text{if } \alpha \text{ is a successor ordinal } \beta + 1, \\ A_{pred_n(\alpha)}(n) & \text{if } \alpha \text{ is a limit ordinal.} \end{cases}$$

The computation of this function plays an important role in the unbounded search procedures of Reingold and Shen [1991]. Moreover, these search procedures themselves use ordinals to index the recursive calls.

These operations also make it easy to encode problems like the "Battle of Hydra and Hercules" of Kirby and Paris [1982] as hard-to-prove-well-defined functions on binary trees.

3 Medium Sized Ordinals

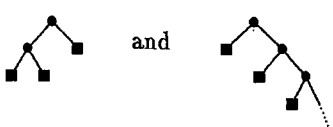
List structures, in general, correspond to "rational" binary trees, which are like ordinary binary trees, but paths may be of length ω, as long as there are only a finite number of distinct subtrees.

3.1 Axioms of Ordering

All the principles of Section 2.1 apply to this case as well, but an infinite number of deletions could increase a tree without violating Principles 1–3. So, we take the following extension of Principle 2 as axiomatic:

Principle 4 (Continuity). *Replacing infinitely many subtrees by greater or equivalent ones results in a greater or equivalent tree.*

Principles 1–4 do not, however, give a total ordering. We do not, for example, know how to order



An additional principle is called for:

Principle 5 (Dominance). *If* $x > y_i$, *for all* $i = 1, 2, \ldots$, *then* $cons(x, nil) \geq cons(y_1, cons(y_2, \ldots))$.

For finite trees, this is a direct consequence of Theorem 1.

3.2 Order-Preserving Mapping

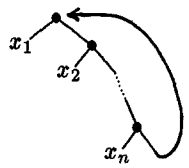
It turns out that we can restrict ourselves to the class of list structures in which there are no cycles except self-loops. Call such a list *normalized.*

Theorem 3. *For every rational binary tree* t *there is a normalized list* ℓ *such that* $t \leq \ell \leq t$.

When comparing structures, like ℓ, under $\leq$, we mean to compared its (possibly) infinite tree expansion.

Proof. All cycles in the graph representation of a rational tree can be reduced to self loops as follows: If a full binary tree is homeomorphically embedded in t, then t is equivalent to the structure z such that $z \equiv cons(z, z)$, which is just a double self-loop:

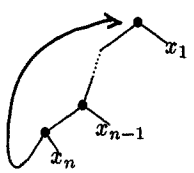
Consider a cyclic graph $z \equiv cons(x_1, cons(x_2, \ldots, cons(x_n, z)))$:



If any of the x_k contains all of z as a subterm, then z both contains the full binary tree (obtained by deleting all other x_i and pruning x_k to what is left of z) and is contained by it (as are all binary trees). Hence, z is equivalent to the full binary tree.

If none of the x_i have z as a subterm, then, by induction on $|l|$, we can suppose that there is a normalized list among the x_i that has a maximal ordinal assignment. We have z less than or equal to the structure $z' \equiv cons(\max x_i, z')$ by Monotonicity, and z greater than or equal to z' by Continuity. Hence, we can replace loop in z with the self-loop of z'.

Similarly, $z \equiv cons(\ldots(cons(cons(z, x_n), x_{n-1}), \ldots), x_1)$, that is,



can be replaced by the double-self-loop corresponding to the full tree or by a self-loop $z' \equiv cons(\max\{x_i\}, z')$. □

An attempt to prove a result like Theorem 3 appears in [Brown, 1979].

Proposition 2. *There is an order-preserving mapping from normalized lists, under the above ordering, onto the ordinals up to and including* $\epsilon_{\epsilon_{\epsilon_{\ldots}}}$.

Proof. The mapping from lists to ordinals is:

$$
\begin{array}{lcll}
[\![nil]\!] & = & 0, & \\
[\![t \text{ such that } t \equiv cons(t, x)]\!] & = & \epsilon_{[\![x]\!]}, & \\
[\![t \text{ such that } t \equiv cons(x, t)]\!] & = & \omega^{[\![x]\!]+1} & \text{if } x \not\equiv t, \\
[\![t \text{ such that } t \equiv cons(t, t)]\!] & = & \epsilon_{\epsilon_{\epsilon_{\ldots}}}, & \\
[\![cons(nil, y)]\!] & = & 1 + [\![y]\!], & \\
[\![cons(x, nil)]\!] & = & \omega^{[\![x]\!]} & \text{if } [\![x]\!] \text{ is a limit ordinal}, \\
[\![cons(x, nil)]\!] & = & \omega^{\beta+1} + 1 & \text{if } [\![x]\!] = \beta + 1, \\
[\![cons(x, y)]\!] & = & \epsilon_\alpha & \text{if } [\![x]\!] = \epsilon_\alpha, [\![y]\!] \leq \alpha, \\
[\![cons(x, y)]\!] & = & \epsilon_\alpha + \beta & \text{if } [\![x]\!] = \epsilon_\alpha, [\![y]\!] = \alpha + \beta, \\
[\![cons(x, y)]\!] & = & \omega^{[\![x]\!]} + 1 + [\![y]\!] & \text{if } [\![x]\!] \text{ is not an epsilon number}, x, y \not\equiv nil.
\end{array}
$$

and its inverse is:

1. $\langle 0 \rangle = nil$

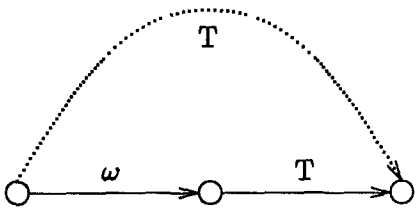


Fig. 2. Result Propagation for Changeable Jungles

edge: if there is a further change of state, the unmarked result is still valid while the marked result information expires. Therefore we have an additional set of result edges R'_J created in situations as above.

For the implementation this suggests the need to provide space for two result edges, a marked and an unmarked one, but this is not really necessary. If we restrict $\trianglerighteq$ to the case where R'_J is empty, then chains of (non-expired) result edges in the normal forms of $\triangleright$ have length at most 2.

Intuitively, this means the following. Suppose we have an evaluation sequence $t_0 \Rightarrow_1 t_1 \Rightarrow_2 \ldots \Rightarrow_n t_n$. We draw an unstamped result edge from t_0 to the last t_i, such that all the $\Rightarrow_j, j \leq i$ are applicative, and mark t_i as an *applicative normal form*. If $i < n$, we draw a stamped result edge from t_i to t_n, provided no $\Rightarrow_l, l \leq n$ changed the state.

Notice that this affects the notion of value: in addition to weak head normal forms there are now *applicative normal forms*.

The concept of a monolithic state is a bit strict, because it does not reflect locality of variables, e.g. (in SML):

```
fun fac n =
    let val p = ref (n,1) in
        while #1(!p) > 0
        do p := (#1(!p)-1, op * (!p));
        #2(!p)
    end;
```

The lifetime of the variable p does not exceed any call of `fac` and it is not accessible outside of `fac` – a dataflow analysis could easily detect this. We could exploit information of this kind for a more sophisticated concept of time and time stamp, but this goes beyond the scope of this paper.

7 Compilation

One subtask of compiling a function definition in a language that supports pattern matching is the management of a symbol table for the pattern variables. It assigns to each variable name a relative address (relative to the stack) and can furthermore be used to detect free variables, anonymous variables and non-linear patterns. Non-linear patterns are forbidden in most languages (not all), but even when they are

allowed, the second occurrence of a variable in a pattern has to be treated differently from its first occurrence.

Under Unlimp, we can generalise the symbol table easily by treating not just variables, but arbitrary non-ground expressions. *Easily*, because comparing complex expressions is here not more difficult or expensive than comparing variables, since it is just the comparison of addresses.

The generalisation to non-ground expressions (nge) works as follows:

- An nge is allocated space on the stack, if and only of it occurs more than once in the left-hand or right-hand side of the definition.
- If an nge occurs a second time, we do not count its subterms as second occurrences.

Variables are also nge's, and in this special case the first point is the detection of anonymous variables, because variables occurring only once do not need to be stored on the stack. For composite expressions it is a common subexpression elimination, because we put them onto the stack if they occur more than once, which corresponds to the introduction of a let-expression. An example (in Haskell), taken from [10]:

```
dropWhile p []  = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise = x:xs
```

For the first rule, there are 3 nge's, but none of them requires space on the stack, p occurs only once and is hence anonymous. In the second rule, we have 8 nge's and 5 of them are allocated space on the stack, see table 1.

Table 1. Generalised Symbol Table

nge	occurrences
`dropWhile p (x:xs)`	1
`dropWhile p`	2
`p`	2
`x:xs`	2
`x`	2
`xs`	2
`p x`	1
`dropWhile p xs`	1

In this example, each nge which is to be stored on the stack is a subexpression of the left-hand side of the rule. Hence, when an expression matches the left-hand side, each nge to be stored on the stack is a subterm of this expression and can be stored during the matching process. One can argue about nge's like `dropWhile p`; it depends on other implementation details (representation of function application) whether they should count or not.

Some care is necessary to treat conditional expressions properly, e.g. common subexpressions of the then- and else-parts of a conditional expression are not really common. It is harmless to put them onto the stack, but harmful to expect them to be there.

8 Garbage Collection

... is a weak point of Unlimp.

The problem is that there is very little *proper* garbage. Deallocating an unreferenced cell would also throw away its result edge and hence a bit of useful information, so that only unreferenced weak head normal forms (have no result edge) and former K-redexes[5] (result edge remains NIL under lazy evaluation) are proper garbage. Unfortunately, almost no weak head normal form will be unreferenced, at least there is the result edge from some (perhaps unreferenced) vertex, and K-redexes are more the exception than the rule. Only in the presence of side-effects can we expect some unreferenced weak head normal forms, because the time stamps of the result edges pointing to them may have expired.

For this reason, a garbage collector would need to collect improper garbage, which is against the spirit of Unlimp, of course. Each unreferenced vertex is (im)proper garbage. Even vertices only referenced by result edges could be treated as improper garbage, but this would require some additional administration, e.g. the garbage collection has to be treated as a global side-effect.

9 Programming Style

Working with an Unlimp implementation can influence programming style. First let us look at a similar influence of lazy evaluation.

Lazy and strict evaluation do not have the same computational power (in a *practical* sense), because lazy evaluation can deal with (conceptually) infinite objects, whereas strict evaluation cannot. Thus, when the natural solution of a problem requires the intermediate creation of an object of infinite size, solving the problem with a strict language means looking for a less natural way.

But such an influence on programming style is also present when there is no such principal difference in computational power, because for certain programming styles, strict evaluation is very inefficient. Typical for this are backtracking algorithms, see [20]; one example is the following simplified version (in Haskell) of the pairing algorithm used for Swiss System chess tournaments:

```
type Entry a = (a,[a])
type Pairing a = [(Entry a,Entry a)]
pairing :: (Eq a) => [Entry a] -> Pairing a
pairing table = if allpairs==[] then error "no pairing"
                else head allpairs
                where allpairs = fullpairs table
```

[5] In λ-calculus, $(\lambda x.t)u$ is a K-redex if x is not free in t.

```
fullpairs :: (Eq a) => [Entry a] -> [Pairing a]
fullpairs [] = [[]]
fullpairs (x:xs) = [ (x,y):zs | y <- xs, condition x y,
                                zs <- fullpairs (xs\\[y]) ]
condition :: (Eq a) => Entry a -> Entry a -> Bool
condition (x,xs)(y,ys) = notElem x ys
```

The function pairing is applied to the actual table of the players (which is supposed to be a list of even length) and produces a list of pairs (the pairing for the next round), such that each pair fulfills the condition. Moreover, the table leader should play (if possible) against the second, the third against the fourth, etc. The entries in the table consist of the player and his or her opponents so far, which is sufficient for the condition "haven't already played against each other".

The above algorithm is expressed in terms of computing all possible pairings and then selecting the first one, which – because of the structure of the algorithm – tends to pairs first with second etc. This is fine for lazy evaluation, but under strict evaluation it is very inefficient, because the number of all possible pairings usually (depending on condition) grows very fast. Table 2 shows the number of reduction steps (successful rule applications) executed to evaluate pairing tab, for five different examples[6], depending on whether the evaluation strategy is strict or lazy and whether full memoization is used or not.

Table 2. Reduction Steps for a Backtracking Algorithm

strategy	stab	mtab1	mtab2	ltab1	ltab2
strict, nomemo	347	16,401	18,211	1,776,421	1,865,213
strict, memo	184	2,133	2,068	84,117	84,361
lazy, nomemo	110	134	474	238	2,276
lazy, memo	88	128	240	230	540

Clearly, strict evaluation is inappropriate for this program. Although the algorithm is correct for strict evaluation too, a programmer using a strict language is encouraged to solve the problem on a lower level, e.g. by making the backtracking strategy explicit.

The impact of memoization on the program is characteristic: the "better" the algorithm is, the less is the effect of memoization. It cannot turn a horribly slow program into a fast one, but it can reduce the horror drastically. The drastic improvement under strict evaluation, and the slight but significant improvement for the heavy backtrackers (mtab2 and ltab2) under lazy evaluation are rather surprising, as the pairing program does not appear to be a prime candidate for memoization.

Full memoization can work together with lazy as well as strict evaluation, but it does not affect the computational power of either strategy. Therefore, there is no

[6] The chosen examples were lists of length 6 after 2 rounds (stab), of length 10 after 3 rounds (mtab1 and mtab2), and of length 14 after 4 rounds (ltab1 and ltab2). The examples mtab2 and ltab2 were chosen to require a lot of backtracking, in contrast to mtab1 and ltab1.

principle need to change the programming style when memoization is absent, but we do have similar kinds of unpleasant encouragement to solve problems at a lower level.

Some further examples (and references) can be found in [11]. We do not have to look for examples that are contrived to support this argument – the following piece of program (in SML) to compute the nth prime number was taken from [19]:

```
fun prime n =
    let fun next(k,i) =
            if n<=i then k
            else if divides(prime i,k) then next(k+1,0)
            else next(k,i+1)
    in
        if n=0 then 2
        else next(prime(n-1)+1,0)
    end
```

It was considered there to be "rather inefficient". In a traditional implementation it is indeed, but under Unlimp it turns out to be fairly reasonable, because memoizing `prime` makes the algorithm behave like a (rather naïve) variation of the sieve of Eratosthenes.

10 Speed-Up in the Small

Most examples people mention when they promulgate memoization are like the naïve version of the Fibonacci function or the above version of `prime` - without memoization terribly inefficient and - since they are naïve - only naïve people would write the function this way, unless it is known that the implementation supports memoization[7].

But memoization also has great effects in the small, as in the `pairing` program. Sometimes they appear very unexpectedly, like the following one:

As their favoured benchmark test for functional programs, Jörn von Holten and Richard Seifert at the University of Bremen took arithmetic on natural numbers represented as successor terms. To make the task hard, the following version of arithmetic was used:

```
data Nat    = Z | S Nat
add Z     x = x
add (S x) y = S (add x y)
mul Z     x = Z
mul (S x) y = add (mul x y) y
pow x     Z = S Z
pow x (S y) = mul (pow x y) x
```

This version is supposed to make arithmetic expensive, because (minor reason) `add` is not tail recursive and (major reason) the right-hand sides of the last rules for

[7] Another less well-known example of this kind is model checking with binary decision diagrams, see [3].

`mul` and `pow` have their recursive calls in the first rather than the second argument of `add` and `mul`. Note that `add n m` is linear in `n` and constant in `m`.

However, the response time of an Unlimp implementation turned out to be fairly stable under switching the arguments of `add` and `mul` in the mentioned rules. The reason is that several addition terms reappear in this process, because computing $n + m$ involves also the computation of $k + m$ for all k less than m.

The following table compares the number of evaluation steps to compute 4^2, 4^3, and 4^4. The left figures show the number of steps for the above definition, the right figures refer to the version obtained by switching the arguments of the mentioned calls of `add` and `mul`.

Table 3. Reduction Steps for a Successor Arithmetic

strategy	4^2		4^3		4^4	
strict, nomemo	40	42	554	116	8748	382
strict, memo	22	40	83	109	324	358
lazy, nomemo	96	195	444,678	1,611	too many	13,123
lazy, memo	27	42	192	116	2295	382

The suspected bad behaviour of exponentiation does not appear under memoization and strict evaluation, here it is even slightly better than the ordinary definition. Only for lazy evaluation, memoization cannot fully compensate for the "bad" algorithm.

As in the `pairing` example, we can again observe different kinds of improvement, depending on how "badly" the algorithm behaves. In both cases, the effects appeared in the small, i.e. they had no fancy recursive structure (as the `prime` example), the functions were linear recursive.

11 Conclusion

A unique representations for expressions can affect compilation, execution and usage of functional languages.

We tried to convey the spirit of thinking in unique representations and of exploiting it for different purposes, e.g. for compilation. The given modelling by hypergraphs stays close to the machine level and allows several meta-observations on a rather abstract level. We showed how memoization and side-effects can happily coexist, even in the hypergraph modelling.

The effect of memoization on program execution seems to be well-known, but the analysis of the given examples suggest that it is not well-known enough. When using full memoization, i.e. storing *every* evaluation result, an important and often unexpected phenomenon appears: a cumulative speed-up by saving minor, but numerous computations. This phenomenon encourages a more problem-oriented programming style.

(inc, id) is an internal functor, which also satisfies the commutative diagrams for preserving the identity and the composition arrows. The arrow id is the identity arrow of C_0 and the arrow inc gives us the *inclusion* morphisms. A *directed complete I-category* is given by the same diagrams in the category of dcpo's with continuous maps. Similarly, an *algebraic*, respectively *continuous*, I-category has as its base the category of algebraic posets and continuous, compact point preserving maps, respectively continuous posets and continuous, way-below preserving maps.

An internal functor of I-categories preserves the above diagrams and, hence, preserves the inclusion morphisms; externally we call it a *standard* functor or an *I-functor*. Similarly, an internal functor of directed complete I-categories also preserves directed joins of morphisms and externally we call it a *continuous* I-functor. Any internal I-category can be completed by taking the completion of its objects and arrows to obtain an internal algebraic I-category, which externally we call its *I-completion*. Similarly, we have the notions of *I-adjunction*, *I-retraction*, *I-projection* etc.

Since the main motivation for introducing I-categories has been to obtain a common framework for categories of information systems for domains, we will, in what follows, give an external description of continuous I-categories as ordered categories and then verify that they have various basic properties analogous to continuous posets on the one hand and continuous categories on the other. As in the case of partial orders, many properties of algebraic I-categories are in fact best understood in the more general framework of continuous I-categories. For reason of space, most of the proofs are omitted in this paper.

2 Background: I-categories

In this work, a dcpo is a directed complete partial order with bottom.

Definition 2.1 An *I-category* is a four-tuple $(P, \mathrm{Inc}, \sqsubseteq, \Delta)$ where:

- P is a category,
- $\mathrm{Inc} \subseteq \mathrm{Mor}$ is the subclass of *inclusion morphisms* of P such that in every hom-set, $\hom(A, B)$, there is at most one inclusion morphism, denoted by $\mathrm{in}(A, B)$ or $A \rightarrowtail B$,
- $\sqsubseteq^{A,B}$ is a partial order on $\hom(A, B)$, for all $A, B \in \mathrm{Obj}$,
- $\Delta \in \mathrm{Obj}$ is a distinguished object;

which satisfy:

Ax 1 (i) The subcategory P^i of all objects and inclusion morphisms of P forms a partially ordered class represented as a category.

(ii) $\mathrm{in}(\Delta, A)$ exists, for all $A \in \mathrm{Obj}$ and $\mathrm{in}(\Delta, A) \sqsubseteq f$ for all morphisms $f \in \hom(\Delta, A)$.

Ax 2 $f_1 \sqsubseteq f_2 \;\&\; g_1 \sqsubseteq g_2 \;\Rightarrow\; f_1; g_1 \sqsubseteq f_2; g_2$, whenever the compositions are defined.

We denote the partial order induced by inclusion morphisms on Obj by $\trianglelefteq$. The partial order $\trianglelefteq^m$ on Mor is defined by $f \trianglelefteq^m g$ if $dom(f) \trianglelefteq dom(g)$, $cod(f) \trianglelefteq cod(g)$, and the following diagram weakly commutes:

$$\begin{array}{ccc} \mathrm{dom}(g) & \xrightarrow{\;g\;} & \mathrm{cod}(g) \\ \uparrow & \sqsupseteq & \uparrow \\ \mathrm{dom}(f) & \xrightarrow{\;f\;} & \mathrm{cod}(f) \end{array}$$

We can also define a pre-I-category by requiring in Axiom(i) that (P, Inc) is just a pre-order. It is also possible to dispense with the object Δ and drop Axiom 2.1(ii) to get an *I-category without a least object.* All the results in this paper with no reference to Δ remain valid for I-categories without a least object.

A *directed complete I-category* $(P, \mathrm{Inc}, \sqsubseteq, \Delta)$ is an I-category which satisfies the following three axioms:

Ax 3 $(\mathrm{Mor}, \trianglelefteq^m)$ is a dcpo.

Ax 4 $(\mathrm{Inc}, \trianglelefteq^m)$ is a sub-dcpo of $(\mathrm{Mor}, \trianglelefteq^m)$.

Ax 5 Composition of morphisms is a continuous operation with respect to $\trianglelefteq^m$.

An *(ω-)algebraic I-category* is a directed complete I-category which satisfies the following two axioms:

Ax 6 The dcpo $(\mathrm{Mor}, \trianglelefteq^m)$ is (ω-)algebraic.

Ax 7 The (total or partial) maps:

$$\begin{aligned} \mathrm{dom}, \mathrm{cod} &: \mathrm{Mor} \to \mathrm{Obj} \\ \mathrm{in}(-,-) &: \mathrm{Obj} \times \mathrm{Obj} \to \mathrm{Mor} \\ -;- &: \mathrm{Mor} \times \mathrm{Mor} \to \mathrm{Mor} \end{aligned}$$

preserve compact points in the corresponding dcpo's.

3 Continuous I-categories

3.1 Basics

Definition 3.1 An *(ω-)continuous I-category* is a directed complete I-category which satisfies the following axioms:

Ax 6* $(Mor, \trianglelefteq^m)$ *is an* $(\omega$*-)continuous dcpo.*

Ax 7* The (total or partial) maps:

$$\begin{aligned} \mathrm{dom}, \mathrm{cod} &: \mathrm{Mor} \to \mathrm{Obj} \\ \mathrm{in}(-,-) &: \mathrm{Obj} \times \mathrm{Obj} \to \mathrm{Mor} \\ -;- &: \mathrm{Mor} \times \mathrm{Mor} \to \mathrm{Mor} \end{aligned}$$

preserve the way below relations in the corresponding dcpo's.

Recall that in a dcpo the relation "a is way below b" can be interpreted as "a is a finite approximation to b" ([GHK+80, page xii]). Therefore in a continuous I-category the mappings dom, cod, $\mathrm{in}(-,-)$ and $-;-$ preserve the notion of finite approximation. The way below relation on objects and morphisms are denoted by $\lll$ and $\lll^m$ respectively.

Example 3.2 *Continuous dcpo's.* Let D be a continuous dcpo. Then, considered as a category, $(D, \mathrm{Inc}, =, \bot)$ with Inc = Mor is a continuous I-category. To see this, we first prove that given two morphisms $f = \mathrm{in}(a, b)$ and $f' = \mathrm{in}(a', b')$, we have $f' \lll^m f$ iff $a' \lll a$ and $b' \lll b$. Suppose $f' \lll^m f$ and let $a = \bigvee^{\uparrow}_{i \in I} a_i$. Put $f_i = \mathrm{in}(a_i, a); f$, for each $i \in I$. Then $f' \lll^m f = \bigvee_i f_i$ and, hence, there exists $i \in I$ with $f' \trianglelefteq^m f_i$ which implies $a_i \trianglelefteq a'$, i.e. $a' \lll a$. To show that $b' \lll b$, assume $b = \bigvee^{\uparrow}_{i \in I}$ and let $a = \bigvee^{\uparrow}_{j \in J} a_j$. Then for each $j \in J$, there exists $j' \in I$ with $a_j \trianglelefteq b_{j'}$. Putting $f_j = \mathrm{in}(a_j, b_{j'})$, we have $f' \lll^m f = \bigvee^{\uparrow}_j f_j$. It follows that for some $j \in J$ we have $f' \trianglelefteq^m f_j$, i.e. $b' \trianglelefteq b'_j$ which implies that $b' \lll b$. To prove the converse, assume $a' \lll a$ and $b' \lll b$ and suppose $f \trianglelefteq^m \bigvee^{\uparrow}_{i \in I} f_i$, with $f_i : a_i \to b_i$. Then $a \trianglelefteq \bigvee^{\uparrow}_{i \in I} a_i$ and $b \trianglelefteq \bigvee^{\uparrow}_{i \in I} b_i$, and, hence there exist $i_1, i_2 \in I$ with $a' \trianglelefteq a_{i_1}$ and $b' \trianglelefteq b_{i_2}$. Let $i \in I$ be such that f_i is above f_{i_1} and f_{i_2}, then $f' \trianglelefteq^m f_i$ and therefore $f' \lll^m f$. This proves our claim. The three clauses of Axiom 7* now follow immediately and it is easy to show that $(\mathrm{Mor}, \trianglelefteq^m)$ is continuous.

Given a continuous I-category P, we say that a subcategory Q of P is a basis of P if $(\mathrm{Mor}_Q, \trianglelefteq^m)$ is a basis of $(\mathrm{Mor}_P, \trianglelefteq^m)$. We then obtain the following results.

Proposition 3.3 A continuous I-category with a basis of compact morphisms is algebraic.

Proposition 3.4 An algebraic I-category is continuous.

3.2 Way-below preserving maps

Let $(D, \sqsubseteq)$ be a dcpo, where D is now assumed to be a set. We say a map $f : D \to E$ between two dcpo's is *open* if $A \in \Omega D \Rightarrow \uparrow f(A) \in \Omega E$. We obtain the following topological characterisation of the way below preserving maps.

Proposition 3.5 For a map $f : D \to E$ between two continuous dcpo's the following are equivalent:

(i) f preserves the way below relation.

(ii) f is open.

Proof (i)⇒(ii) Let A be an open set in D. We must show that $\uparrow f(A)$ is inaccessible by directed lubs in E. Suppose, for some $x \in A$, $f(x) = \bigvee^{\uparrow} B \in \uparrow f(A)$. Since D is continuous, x is the directed lub of elements of D way below it and since A is open one of these elements y, say, is in A. Now $y \ll x$ implies $f(y) \ll f(x)$. Hence, there exists $b \in B$ with $f(y) \sqsubseteq b$, and as $f(y) \in f(A)$ we conclude that $b \in \uparrow f(A)$.

(ii)⇒(i) Let $y \ll x$ and $f(x) \sqsubseteq \bigvee^{\uparrow} B$. The set $A = \{z | y \ll z\}$ is open (see [Joh82, page 290]), and $x \in A$. Therefore $\uparrow f(A)$ is open and contains $f(x)$ and, hence, also $\bigvee^{\uparrow} B$. So there exists some $b \in B \cap \uparrow f(A)$. But, $f(y) \sqsubseteq a$ for all elements $a \in \uparrow f(A)$. It follows that $f(y) \sqsubseteq b$. □

Since open sets can be regarded as properties, the above proposition means that way below preserving maps, i.e. maps preserving the notion of finite approximation of points, and property preserving maps are basically the same. We also have:

Proposition 3.6 Let D and E be dcpo's and $f : D \to E$ a continuous map. Then the following are equivalent.

(i) f is an open map.

(ii) The frame map $\Omega f : \Omega E \to \Omega D$ preserves meets.

(iii) The frame map $\Omega f : \Omega E \to \Omega D$, considered as a functor between two categories, has a left adjoint.

4 Some functors on I-categories

Recall that a standard functor between two I-categories is one which preserves inclusion morphisms and a continuous functor between two directed complete dcpo's is one for which the induced mapping on morphisms is continuous. We will now introduce a number of standard functors between I-categories, with which we are able to generalize the basic categorical properties of partial orders.

4.1 I-adjunctions

Definition 4.1 Let P and Q be I-categories. A pair of standard functors $F : P \to Q$ and $G : Q \to P$ forms a *Galois I-connection* if the induced maps on the partial order of morphisms is a Galois connection of partial orders.

The pair (F, G) will then form a *lax* adjunction, in which the unit $\eta : 1_P \to G \circ F$ given by $\eta_A = \mathrm{in}(A, G \circ F(A))$ and the co-unit $\epsilon : F \circ G \to 1_Q$ given by $\epsilon_B = \mathrm{in}(F \circ G(B), B)$ are lax natural transformations, i.e. their naturality diagrams are *weakly* commutative.

In view of this, we call them an *I-adjunction*, F the left I-adjoint and G the right I-adjoint.

4.2 I-completion

Given a partial order $(L, \sqsubseteq)$, where L is a class, its ideal completion, denoted by $(\overline{L}, \subseteq)$, is the class of directed and downward closed *subsets* of L ordered by inclusion.

Let $P = (P, \mathrm{Inc}, \sqsubseteq, \Delta)$ be an I-category; its *I-completion*, $\overline{P} = (\overline{P}, \overline{\mathrm{Inc}}, \subseteq, \overline{\Delta})$ is given by:

- $\mathrm{Obj}_{\overline{P}} = \overline{\mathrm{Obj}_P}$;
- $\mathrm{Mor}_{\overline{P}} = \overline{\mathrm{Mor}_P}$,
  $\mathrm{dom}(\mathcal{F}) = \{\mathrm{dom}(f) | f \in \mathcal{F}\}$, $\mathrm{cod}(\mathcal{F}) = \downarrow\{\mathrm{cod}(f) | f \in \mathcal{F}\}$,
  $\mathrm{Id}(\mathcal{A}) = \downarrow\{\mathrm{Id}(A) | A \in \mathcal{A}\}$,
  $\mathcal{F}; \mathcal{G} = \downarrow\{f; g | f \in \mathcal{F}, g \in \mathcal{G}\}$ (assuming $\mathrm{cod}(\mathcal{F}) = \mathrm{dom}(\mathcal{G})$),
  where $\downarrow S$ is the downward closure of S;
- $\mathrm{in}(\mathcal{A}, \mathcal{B})$ exists iff $\mathcal{A} \subseteq \mathcal{B}$, and when it exists we have:
  $\mathrm{in}(\mathcal{A}, \mathcal{B}) = \downarrow\{\mathrm{in}(A, B) | A \in \mathcal{A}, B \in \mathcal{B}, A \trianglelefteq B\}$;
- the inclusion $\subseteq$ as the partial order on hom-sets;
- $\overline{\Delta} = \{\Delta\}$.

$\overline{P}$ can be shown to be an algebraic I-category. Clearly I-completion is a generalisation of ideal completion and for posets these two notions coincide. On the other hand, I-completion can be regarded as a restricted form of *Ind-completion* of a category: The I-completion $\overline{P}$ of an I-category P is equivalent to the full subcategory of the Ind-completion, Ind-P, of P consisting of those ind-objects which are small diagrams in P^i.

A poset P is a dcpo iff the embedding $\downarrow(-) : P \to \overline{P}$ has a left adjoint (the left adjoint necessarily sends an ideal to its directed lub in P); and a locally small category P has small filtered colimits iff the embedding $y : P \to \text{Ind-}P$ has a left adjoint. Similarly, we have:

Proposition 4.2 An I-category P is directed complete iff the functor $\downarrow(-) : P \to \overline{P}$ has a left I-adjoint.

Furthermore I-completion can be regarded as a functor. Let **I-Cat** be the category of small I-categories with standard functors and **AlgI-Cat** the category of small algebraic I-categories with standard and continuous functors. I-completion extends to a functor $\mathcal{C} : \textbf{I-Cat} \to \textbf{AlgI-Cat}$. We then obtain the following universal property, which is the generalization of the corresponding result for partial orders.

Proposition 4.3 $\mathcal{C}$ is left adjoint to the forgetful functor from **AlgI-Cat** to **I-Cat**.

4.3 I-retractions and I-projections

Definition 4.4 Let P and Q be directed complete I-categories. A standard continuous endofunctor $R : P \to P$ is called an *I-retraction* (*I-projection*) if the induced map on morphisms $R_m : \text{Mor} \to \text{Mor}$ is a retraction (projection). Two standard and continuous functors $R : P \to Q$ and $E : Q \to P$ are said to form an I-retraction (projection) embedding pair if $R_m : \text{Mor}_P \to \text{Mor}_Q$ and $E_m : \text{Mor}_Q \to \text{Mor}_P$ form a retraction (projection) embedding pair.

Note that an I-projection embedding is a Galois I-connection and, hence, a lax adjunction. In complete analogy with the theory of partial orders again, we obtain the following.

Proposition 4.5 (i) The image of an I-retraction is a directed complete I-category.

(ii) The image of an I-projection on a continuous I-category is a continuous I-category.

Theorem 4.6 Any (ω)-continuous I-category is the projective image of an (ω)-algebraic I-category.

A dcpo D is continuous iff $\bigvee^{\uparrow}(-) : \overline{D} \to D$ has a left adjoint (which will then necessarily take any element to the directed set of elements way below it). Johnstone and Joyal's definition of continuous category in [JJ82] is in fact a generalisation of this. We analogously have:

Theorem 4.7 Let P be a directed complete I-category. Then P is continuous iff the functor $\bigvee^{\uparrow}(-) : \overline{P} \to P$ has a left I-adjoint.

It is well known that a retract of a continuous dcpo is a continuous dcpo. A similar result holds for I-categories. The proof of the following result, in effect, uses the analogue of the "adjoint lifting" lemma [Joh81] for I-categories.

Theorem 4.8 Let P be a continuous I-category and P' a directed complete I-category. Suppose $R : P \to P'$ and $E : P' \to P$ form an I-retraction embedding pair. Then P' is continuous.

5 Effectiveness

In [ES91a], an effective theory for ω-algebraic I-categories was developed and an effective version of the initial algebra theorem was presented. We generalise these to ω-continuous I-categories. To do this, we need to formulate a suitable notion of effectively given ω-continuous dcpo's, as the existing formulations in [Smy77] and [Tan74] do not fit in our context.

We define the notion of effectively given ω-continuous cpo by putting a suitable recursive structure on a basis of it. Our treatment resembles that of Plotkin in [Plo81] for ω-algebraic cpo's. ϕ_n denotes the n^{th} partial recursive function in the standard enumeration and $\langle m_1, \cdots, m_n \rangle$ is the n-tupling function from $\mathbb{N}^{\kappa}$ to $\mathbb{N}$.

Let E be an ω-continuous cpo with a countable basis B and let $e : \mathbb{N} \to B$ be an onto map i.e. an enumeration of the basis elements of E. We say E is effectively given w.r.t. e with index $\langle a, s, t, u \rangle$ if

(i) $e_a = \bot$.

(ii) The predicate $e_m \uparrow e_n$ (i.e. e_m and e_n are bounded above) is recursive in m and n with index s.

(iii) $e_m \sqsubseteq e_n$ is recursive in m and n with index t.

(iv) $e_m \ll e_n$ is recursive in m and n with index u.

An effective chain w.r.t. B of E with index l w.r.t. e is an increasing chain $\langle e_{\phi_l(n)} \rangle_{n \geq 0}$ with $e_{\phi_l(n)} \ll \bigsqcup_i e_{\phi_l(i)}$ for all $n \geq 0$. An element $d \in E$ is *computable* w.r.t. B and e if there exists an effective chain with lub d. The index of d w.r.t. e is defined to be the index of this effective chain. We then show that every non-negative integer gives rise to an effective chain and we can therefore define a *computable* function in the usual way [Plo81].

Having obtained a suitable theory of an effectively given $\omega-$continuous dcpo, the rest of our work is similar to the case of effectively given algebraic I-categories[ES91a].

Definition 5.1 Let P be an ω-continuous I-category. An *enumeration* (A, f) of a basis, Q, of P is given by an enumeration $A : \mathbb{N} \to \mathrm{Obj}_Q$ and an enumeration $f : \mathbb{N} \to \mathrm{Mor}_Q$ of objects and morphisms of Q respectively. P is said to be *effectively given* with respect to (A, f) if $(\mathrm{Mor}_P, \trianglelefteq^m)$ is effectively given and the functions dom, cod, $\mathrm{in}(-,-)$ and $-;-$ are computable.

Proposition 5.2 Let P be an ω-continuous I-category effectively given with respect to (A, f).

(i) $(\mathrm{Obj}_P, \trianglelefteq)$ is effectively given with respect to A.

(ii) The mappings dom, cod : $(\mathrm{Mor}, \trianglelefteq^m) \to (\mathrm{Obj}, \trianglelefteq)$ and the mapping $\mathrm{Id} : (\mathrm{Obj}, \trianglelefteq) \to (\mathrm{Mor}, \trianglelefteq^m)$ are computable.

(iii) If $B, C \in \mathrm{Obj}_P$ are computable and $B \trianglelefteq C$, then $\mathrm{in}(B, C)$ is computable.

(iv) If $g, h \in \mathrm{Mor}_P$ are computable with $\mathrm{cod}(g) = \mathrm{dom}(h)$, then $g; h$ is also computable.

Theorem 5.3 Let F be a computable endofunctor on an effectively given ω-continuous I-category P. Then P has a computable initial algebra $(D, \mathrm{Id}(D))$, an index for which is effectively obtainable from one for F. Moreover if (E, k) is a computable F-algebra, the unique morphism h satisfying $h = F(h); k$ is computable and an index for h can be effectively obtained from one for F and one for (E, k).

6 Locally quasi-compact topological spaces

Consider a locally quasi-compact space X. It is well known that the lattice of open sets of X is a continuous lattice and given two open sets A and B, we have $A \ll B$ iff A is relatively compact w.r.t. B (i.e. every open covering of B has a finite subcovering of A) [GHK+80, Page 42]. X in fact gives rise to a non-trivial continuous I-category as follows. Let K be the category whose objects are the open sets of X and whose morphisms are the continuous functions between these open sets. Then one can show that $(K, \mathrm{Inc}, =, \emptyset)$, where Inc is the set of inclusions, is a continuous I-category, in which we have $f \trianglelefteq^m g \Rightarrow f = g\restriction_{\mathrm{dom}(f)}$ and

$$f \lhd\!\lhd^m g \quad \Longleftrightarrow \quad f \trianglelefteq^m g \ \&\ \mathrm{dom}(f) \lhd\!\lhd \mathrm{dom}(g) \ \&\ \mathrm{cod}(f) \lhd\!\lhd \mathrm{cod}(g).$$

The above result can be generalised to **LocCom**, the category of locally quasi-compact spaces and continuous maps between them. If A and B are locally quasi-compact spaces, we put $A \trianglelefteq B$ if A is a subspace of B and the inclusion is an open map. Then $(\mathbf{LocCom}, \mathrm{Inc}, =, \emptyset)$ is a continuous I-category.

7 Categories of continuous posets

Categories of continuous posets with continuous functions, e.g. **CSDom**, are not in general equivalent to continuous I-categories: In fact, one can show that there are not enough objects way below a given object to generate it. However, we can show by refining the category **CS-ISys** in [ES91a], that the category of continuous Scott (bounded complete) domains with strict, continuous open maps is a continuous I-category, in which the inclusions correspond to *rigid* projection-embeddings and hom-sets are discretely ordered. The result can be extended to other categories of continuous domains including the retracts of SFP or L-domains. However, the restriction to rigid projection-embeddings and open maps seems to be essential in capturing categories of continuous domains as continuous I-categories.

8 Categories of continuous information systems

We develop an effective theory for categories of continuous information systems and for solving domain equations effectively over continuous domains.

Let $\mathcal{C}$ be an effectively given ω-algebraic–I-category. Then the Karoubi envelope of $\mathcal{C}$, $\mathrm{Kar}(\mathcal{C})$ is a complete I-category but is not in general ω-algebraic. However, consider the *category of arrows*, $\mathcal{C}^{\rightarrow}$ of $\mathcal{C}$; its objects are arrows of $\mathcal{C}$ and its morphisms are pairs of arrows of $\mathcal{C}$ of the form $(a, b) : f \to g$ with $f; b \sqsubseteq a; g$. Given objects $f : A \to B$ and $f' : A' \to B'$, we put $f \trianglelefteq_{\mathcal{C}^{\rightarrow}} g$ iff $f \trianglelefteq^{m}_{\mathcal{C}} g$ with $\mathrm{in}(f, f') = (\mathrm{in}(A, A'), \mathrm{in}(B, B'))$. Furthermore homsets of $\mathcal{C}^{\rightarrow}$ inherit the pointwise ordering from the homsets of $\mathcal{C}$ and we have:

Theorem 8.1 The arrow category of a (complete, ω-algebraic, effectively given) I-category is another such category.

$\mathrm{Kar}(\mathcal{C})$ can clearly be identified with a subcategory of $\mathcal{C}^{\rightarrow}$ and is therefore effectively given. We therefore have an effective presentation of the Karoubi envelope of **BC-ISys**, **SFP-ISys**, **DI-ISys**, (defined in the introduction) which are equivalent to the categories of the corresponding continuous domains.

Any endofunctor on $\mathcal{C}$ induces in the obvious way an endofunctor on $\mathcal{C}^{\rightarrow}$. If $r : A \to A$ is an object of $\mathrm{Kar}(\mathcal{C})$, then the constant endofunctor on $\mathrm{Kar}(\mathcal{C})$ with value $r : A \to A$ can be extended to $\mathcal{C}^{\rightarrow}$. All this means that if F is an endofunctor on $\mathrm{Kar}(\mathcal{C})$ obtained by composition of endofunctors induced from $\mathcal{C}$ and constant endofunctors, then we can solve the domain equation $F(X) \cong X$ in $\mathrm{Kar}(\mathcal{C})$ effectively by solving it effectively in $\mathcal{C}^{\rightarrow}$, i.e. we have an effective initial algebra theorem for such functors. In particular we can solve domain equations over continuous domains involving disjoint sum, product, function space and constant endofuctors.

Acknowledgement

I would like to thank my colleagues in the theory and formal methods section of the department of computing at Imperial college, in particular Samson Abramsky for discussions on domain theory, Mike Smyth for discussions on continuous domains and Steve Vickers for his suggestion to present the continuous I-categories internally. Above all, however, I am grateful to Achim Jung who during my two week visit to Darmstadt in August 1991 carefully examined the results of this paper and made a number of valuable suggestions concerning the results in the last three sections. Thanks are also due to Mark Ryan for his patient assistance in LaTeX. Paul Taylor's diagram macros have been used in this paper.

References

[ES91a] A. Edalat and M. B. Smyth. Categories of information systems. In D. H. Pitt, P. L. Curien, S. Abramsky, A. M. Pitts, A. Poigne, and D. E. Rydeheard, editors, *Category theory in computer science*, pages 37–52. Springer-Verlag, 1991.

[ES91b] A. Edalat and M. B. Smyth. Categories of Information Systems. *Theoretical Computer Science*, 1991. to appear.

[GHK+80] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices.* Springer Verlag, Berlin, 1980.

[JJ82] P. T. Johnstone and A. Joyal. Continuous categories and exponentiable toposes. *J. Pure Appl. Algebra*, 25:255–296, 1982.

[Joh81] P. T. Johnstone. Injective toposes. In *Continuous lattices*, pages 284–297. Springer, Berlin, 1981. Lecture Notes in Math. No. 871.

[Joh82] P. T. Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, Cambridge, 1982.

[Plo81] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[Smy77] M. B. Smyth. Effectively given domains. *Theoretical Computer Science*, 5:257–274, 1977.

[Tan74] A. Tang. *Recursion theory and descriptive set theory in effectively given T_0 spaces.* PhD thesis, University of Princeton, 1974.