stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 95/78 MAART

H.J. BOOM

TASK REDUCTION SYSTEMS

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS/MOS subject classification scheme (1978): 68A05, 02E05
AMS/MC: 68J05.

Computing Reviews 4.0

Task reduction systems *

by

H.J. Boom

ABSTRACT

The essential computational dependencies in a problem to be programmed can be expressed as a "task replacement system" which indicates how tasks can be accomplished if other tasks are first accomplished. Associating a well-ordering with a suitable task replacement system can provide a constructive proof that an algorithm exists for solving a problem, without actually requiring the effort of writing a program. The method is demonstrated on very small examples.

KEYWORDS & PHRASES: *task replacement, task reduction, programming methodology, termination, well-ordering.*

---

* This report will be submitted for publication elsewhere

0. INTRODUCTION

When writing a large program, it is necessary to have some plan of attack before beginning to use a methodology such as top-down programming. In particular, one must have some measure of simplicity to ensure that a refinement step indeed does lead in the proper direction. An operation can be replaced by other operations only when there exist algortihms to implement the new operations.

Using oversimplified examples, this paper presents one method for organizing the essential computational dependencies in a problem into a proof that an algorithm exists. First, a set of "tasks" is constructed, which includes the tasks that the program is to perform. Each of these tasks can be viewed as a challenge to the computer to solve some problem. Statements about the problem domain then lead to methods of replacing tasks by other simpler tasks. The topology of the resulting graph of interreducible tasks is then analyzed by using a well-ordering to ensure the existence of one or more algorithms. Afterwards, it is possible to schedule the indicated computations in order to achieve efficient execution.

1. TASK REPLACEMENT SYSTEMS

A task replacement system consists of:

"T", a set of "tasks", possibly infinite,
" ", the "empty" task,
"M", a set of "methods",
">>", a relation of "replacability on $T \times T \cup \{\cdot\} \times M$.

"t >> s(m)" means that "t" can be solved by solving "s" an then applying method "m". We write "t >>•(m)" or even "t >>(m)" to indicate that method "m" solves "t" directly. If the method m is clear from context or not relevant, we omit it and simply write "t >>s" or "t >>". Such a relation "t >>s" is called a "task replacement". We are primarily interested in finite sequences of the form

$$t_n >> t_{n-1} >> t_{n-2} >> \ldots >> t_2 >> t_1 >> .$$

The set T of tasks will often consist of several parameterized sets. In this case we can speak of "task replacement schemata". A task replacement schema is simply a general rule for summarizing many task replacements. Task replacement schemata will usually be written using free variables, and occasionally with validity conditions as constraints.

2. TASK REPLACEMENT SYSTEMS

Let us suppose a computer is occasionally faced with the task of evaluating a factorial. Then

$$T = \{ \triangleright n! = ? \mid n \text{ is an integer} \}.$$

The notation "$\triangleright n! = ?$" means the task of finding some value v for "?" and proving $n! = v$. We have the following task replacement schemata:

$\triangleright 1! = ? \gg$	(method: take $1! = 1$)
$\triangleright n! = ? \gg \triangleright (n-1)! = ?$	if $n > 0$.
	(method: if $(n-1)! = v$, multiply v by n to get w, and let $n! = w$).
$\triangleright (n-1)! = ? \gg \triangleright n! = ?$	if $n > 0$.
	(method: if $n! = v$, divide v by n to get w, and let $(n-1)! = w$).

The integers and the symbols "$\triangleright$", "=", "?", and "!" above are symbols used in the notations for tasks. "(",")", "n", "+1", and "−1" are used to summarize many task replacements in one line. These schemata lead to the following set of replacement rules: (the methods have not been written)

$$\triangleright 1! = ? \gg .$$

$\triangleright 0! = ? \gg \triangleright 1! = ?$	$\triangleright 1! = ? \gg \triangleright 0! = ?$
$\triangleright 1! = ? \gg \triangleright 2! = ?$	$\triangleright 2! = ? \gg \triangleright 1! = ?$
$\triangleright 2! = ? \gg \triangleright 3! = ?$	$\triangleright 3! = ? \gg \triangleright 2! = ?$
$\triangleright 3! = ? \gg \triangleright 4! = ?$	$\triangleright 4! = ? \gg \triangleright 3! = ?$
.	.
.	.
.	.

To compute the value of 6!, we might take the following heuristic:

```
solve (t):   if
             t >>.(m) for some m → apply method m
             □ t >>s(m) for some m →
                          solve (s); apply method m
             fi
solve (⊳ 6! = ?)
```

This method might lead to

$$6! >> 5! >> 4! >> 3! >> 2! >> 1! >>.$$

It might equally well lead to

$$6! >> 5! >> 6! >> 7! >> 8! >> 7! >> 8! >> 9! >> 10! \ldots .$$

Although a Universal Problem Solver could solve $\triangleright 6! = ?$ by the above task replacement schemata, in practice it is necessary to impose extra structure on the system. Sections 3 and 4 describe ways of obtaining task replacements; section 6 shows how a task replacement system can be transformed into an algorithm.

3. TRANSFORMING IDENTITIES INTO TASK REPLACEMENTS

Given an identity

$$F(x_1,x_2,\ldots,x_n) = G(x_1,x_2,\ldots,x_n)$$

where F and G are expressions in the x's, we can obtain the following task replacements. For each valid set of values of $x_0,x_1,\ldots,x_n$ we get

$$
\begin{array}{lll}
\triangleright? = F(x_1,x_2,\ldots,x_n) & >> & \triangleright? = G(x_1,x_2,\ldots,x_n) \\
\triangleright? = G(x_1,x_2,\ldots,x_n) & >> & \triangleright? = F(x_1,x_2,\ldots,x_n) \\
\triangleright x_0 = F(\ ?,x_2,\ldots,x_n) & >> & \triangleright x_0 = G(\ ?,x_2,\ldots,x_n) \\
\triangleright x_0 = F(\ ?,x_2,\ldots,x_n) & >> & \triangleright x_0 = F(\ ?,x_2,\ldots,x_n) \\
\triangleright? = F(\ ?,x_2,\ldots,x_n) & >> & \triangleright? = G(\ ?,x_2,\ldots,x_n) \\
 & \ldots & \\
 & \ldots & \\
\triangleright? = F(,?,\ldots,?) & >> & \triangleright? = G(?,?,\ldots?)
\end{array}
$$

Each rule like the above, which produce a task replacement when valid values are substituted for its "schema variables" is called a "task replacement schema". For example, the fifth schema above can lead to the task replacement

$$\triangleright? = F(?,137,\ldots,252) >> \triangleright? = G(?,137,\ldots,252).$$

Other task replacement schemata can also be obtained from identities, such as those generated by substituting one side of an identity for another in some other formula.

4. TASK REPLACEMENTS FROM (CONSTRUCTIVE) THEOREMS

Every (constructive) theorem

$$\forall x_1 \ldots x_\ell \ (\exists y_1 \ldots y_m \ P(x_1 \ldots x_\ell \ y_1 \ldots y_m))$$

$$\Rightarrow (\exists z_1 \ldots z_n \ Q(x_1 \ldots x_\ell \ z_1 \ldots z_n))$$

gives us a task replacement schema

$$\triangleright P(x_1,\ldots,x_\ell,?,\ldots,?) >> \triangleright Q(x_1,\ldots,x_\ell,?,\ldots,?) \ .$$

This result does not hold in non-constructive mathematics, since it may there be impossible to derive a method from the proof of the theorem. This property of constructive mathematics suggests that it is reasonable to use

it for computer programming, even in the (unreasonable) case that one is not a constructive mathematician. Constructive logic is used throughout this paper. To remind the reader of this, the word "constructive" will occasionally appear in parentheses.

5. WELL-ORDERED SETS

A well-founded set is a partially-ordered set in which every descending sequence terminates.

A well-ordered set is a totally-ordered set in which every descending sequence terminates. The literature on well-ordered sets is extensive [1,2,3] and will not be repeated here. However, it is worthwhile to mention a few of the properties of well-orderings for those not yet familiar with them.

Every (constructive) well-ordered set is isomorphic to one that can be built up using the following methods:

(1) The empty set is well-ordered (no sequences!)

(2) Every totally-ordered finite set is well-ordered

(3) The non-negative integers are well-ordered. (this well-ordered set is usually called "ω")

(4) Given any set $\{A_i \mid i \in I\}$ of well-ordered sets A_i indexed by a well-ordered index set I, their concatenation

$$\sum_{i \in I} A_i = \{\langle i,a\rangle \mid a \in A_i,\ i \in I\}$$

is well ordered under the lexicographical order

$$\langle i,a\rangle < \langle j,b\rangle \text{ iff } i < j \text{ or } (i=j \text{ and } a < b).$$

There are standard well-ordered sets, called "ordinal numbers". Every well-ordered set is order-isomorphic to a subset of an ordinal number. A well-founded ordering relation is defined on the ordinals. $\alpha \leq \beta$ iff α is order-isomorphic to an initial segment of β. Every ordinal is order-isomorphic to the set of ordinals less than it.

The class of all ordinal numbers is denoted On.

The first ordinal numbers are identified with the nonnegative integers

$$\begin{array}{ll} 0 & \{\} \\ 1 & \{0\} \\ 2 & \{0,1\} \\ 3 & \{0,1,2\} \\ \ldots & \end{array}$$

The first ordinal after the integers is 'w'. Then we get

$$\begin{array}{ll} w & \{0,1,2,\ldots\} \\ w+1 & \{0,1,2,\ldots;w\} \\ w+2 & \{0,1,2,\ldots;w,w+1\} \\ \vdots & \end{array}$$

After this sequence we get

$$\begin{array}{ll} w2 = w + w & \{0,1,2,\ldots;w,w+1,\ldots\} \\ w2+1 & \{0,1,2,\ldots;w,w+1,\ldots,w2\} \\ w2+2 & \{0,1,2,\ldots;w,w+1,\ldots,w2,w2+1\} \\ \ldots & \\ w3 & \{0,1,2,\ldots;w,w+1,\ldots;w2,w2+1,\ldots\} \\ w3+1 & \{0,1,2,\ldots;w,w+1,\ldots;w2,w2+1,\ldots;w3\}. \\ \ldots & \end{array}$$

After $0,w,w2,w3,\ldots$ we get $ww = w^2$, and so forth:

$0,1,2,3,\ldots;w,w+1,w+2,\ldots;w2,w2+1,\ldots;w3,\ldots;w4\ldots;\ldots;$
$w^2,w^2+1,w^2+2,\ldots;w^2+w,w^2+w+1,\ldots;w^2+w2,\ldots;\ldots;w^2+w^2=w^22,$
$w^22+1,\ldots;\ldots;w^23\ldots w^24\ldots;w^2w=w^3,\ldots w^4\ldots w^w\ldots\ \ldots\ .$

Warning:

Addition, multiplication, etc., are in general non-commutative; in particular

$$1+\omega = \omega \neq \omega+1,$$

since given a one-element set $\{a\}$ and an ω-element set $\omega = \{0,1,2,\ldots\}$ we have the order isomorphism

$$\begin{array}{ll} 1+\omega: & a,0,1,2,\ldots \\ & \updownarrow\ \updownarrow\ \updownarrow\ \downarrow \\ \omega: & 0,1,2,3;\ldots\ . \end{array}$$

But $\omega+1 = \{0,1,\ldots;\omega\}$ has ω as a limit point, whereas $\omega = \{0,1,\ldots\}$ has no limit point.

6. ALGORITHMIC CONTENT

Even though, in principle, a task replacement system may contain enough information to enable a Universal Problem Solver to accomplish a task, it does not yet provide an algorithm. Repeated task replacement may proceed in useless direction, encounter blind alleys, or fail to terminate. Some intelligence is necessary to act as guide. The knowledge obtained from this intelligence can be (represented) as a "complexity mapping" from tasks to ordinal numbers.

Given a set T of tasks and a task replacement relation >>, a mapping $c : T \to On$ is called a complexity mapping. It is said to provide "algorithmic content" iff

$$\forall t \in T: \{(\exists s \in T: t>>s \wedge c(t) > c(s)) \vee (t>>\cdot)\}$$

holds (constructively). If there is algorithmic content, there is an algorithm for performing an arbitrary task $t \in T$:

```
proc perform (t):
    if
       t >>· (method M) → use M
    □ ∃s ∈ T: t>>s (method M) ∧ c(t) > c(s)
            → perform (s); use M
    fi
```

When we have algorithmic content, we obtain a "task reduction system" from a task replacement system, by selecting those task replacements, which reduce complexity. A task replacement system with algorithmic content is called a "task reduction system". The general programming heuristic suggested by this paper now reads:

Determine properties of relevant (and possibly irrelevant) concepts.
Obtain replacement system, possibly expressed as a collection of schemata.
Attempt to find a complexity mapping which provides algorithmic content, possibly adding more task replacements to accomplish this.
Perform proper scheduling of the computations.
Determine data representations and write the program.

It is only in the last steps that one can use anything like top-down coding, bottom-up refinement, middle-out implementation or any other conventional programming "methodology". By this time, all the important decisions about the program have been made, and therefore nearly any systematic methodology will work. Furthermore, the task replacement schemata clearly indicate the necessary data dependencies involved in the global design.

7. BACK TO FACTORIAL

Let

$$c(n!) = |n-1|.$$

This proves algorithmic content for the task replacement system for factorial. The resulting task reduction system is:

$$\triangleright 1! = ? \gg .$$

$$\triangleright 0! = ? \gg \triangleright 1! = ?$$

$$\triangleright 2! = ? \gg 1! = ?$$
$$\triangleright 3! = ? \gg 2! = ?$$
$$\triangleright 4! = ? \gg 3! = ?$$
$$\cdots$$

Here those task replacements which are solutions or decrease complexity have been selected. It is a poper task reduction system because every task is either solution or reducible to a simpler task.

(Note: I first tried c(n!) = n, but this failed to provide a reduction for ▷0! = ?. Adding the fact that 0! = 1 as a new task replacement ▷0! = ? >> . would also have patched things up, but the above discussion is more in line with my intuitive understanding of factorial.)

8. THE ACKERMAN FUNCTION

Suppose a computer (a human or mechanical being) is given the task of determining Ack(3,4), a value of the Ackerman function. He opens the Chemical Rubber Company Handbook of Peculiar Recursions Used in Informatics, and finds several identities:

$$\text{Ack}(0,n) = n+1 \tag{1}$$

$$\text{Ack}(m,\text{Ack}(m+1,n)) = \text{Ack}(m+1,n+1) \tag{2}$$

$$\text{Ack}(2,n) = 3+2n \tag{3}$$

$$\text{Ack}(m+1,0) = \text{Ack}(m,1) \tag{4}$$

$$x+y = y+x \tag{5}$$

Each of those identities shows how he can replace some computations by others. For example, we get:

▷Ack(3,4) = ? >> ▷ find an n such that Ack(4,n) = 4;
then evaluate Ack(4,n+1).
▷Ack(3,4) = ? >> ▷Ack(2,Ack(3,3)) = ?
▷2n = ? >> ▷Ack(2,n) (method: subtract 3).
▷6+7 = ? >> ▷7+6 = ?
▷7+6 = ? >> ▷6+7 = ?
etc., etc.

We decide that task replacements deriving from the first four schemata may be relevant. Now we attempt to find a complexity mapping. A standard heuristic for complexity is to see if it can be built up as a polynomial

in ω with the task parameters as coefficients. First we try

$$c(\triangleright Ack(m,m) = ?) = \omega n+m$$

and

$$c(\triangleright Ack(m,n) = ?) = \omega m+n.$$

The first one does not provide us with a reduction for $\triangleright$ Ack(1,0) = ?, since

$$c(\triangleright Ack(1,0) = ?) = 1,$$

and

$$c(\triangleright Ack(0,1) = ?) = \omega.$$

The second one works quite nicely, except that we do not know what to do with the task

$$\triangleright Ack(m,Ack(m+1,n)) = ?.$$

If we define its complexity as

$$\omega m + Ack(m+1,n)$$

we are begging the question, since we assume the existence of a terminating algorithm for Ackermann in the complexity function. We can break it up into two tasks, to be performed in order, however:

$$\triangleright Ack(m+1,n) = ?_1.$$
$$\triangleright Ack(m,\sim_1) = ?_2.$$

Completing the first task will cause the second one to be changed by replacing "$\sim_1$" by the proper value of "$?_1$" and then it too will be of a proper form.

So we try

$$c(\triangleright Ack(m,\sim) = ?) = \omega m+\omega = \omega(m+1).$$

To make the task reduction for Ack(m+1,0) = ? work, we redefine

$$c(\triangleright Ack(m,n) = ?) = \omega m+n+1$$
$$c(\triangleright Ack(m,\sim_1) = ?) = \omega(m+1).$$

Now we have rules resembling task replacement schemata, but occasionally our rules decompose a task into more than one simpler task.

$\triangleright Ack(0,n) = ?$ >> . (choose n+1)

$\triangleright Ack(m+1,0) = ?$ >> $\triangleright Ack(m,1) = ?$

$\triangleright Ack(m+1,n+1) = ?$ >>
$\triangleright Ack(m+1,n) = ?_1$;
$\triangleright Ack(m;\sim_1) = ?_2$.

In the next section, we shall show that a multiple task reduction system with multiple substitution tasks can be converted into a task replacement system, and so the problem can be solved.

9. MUTIPLE TASK REPLACEMENT

Let S be a "multiple task reduction system" i.e., one in which a task may be replaced by zero, one, or more simpler replacement tasks. For this formulation, instead of reducing a task to the empty task ".", we shall replace it by zero tasks.

Construct a task reduction system T as follows:

T = the set of all finite "subsets" of S.

We allow these "subsets" to have multiplicities; i.e., an element of S may appear more than once in one of these "subsets". This is realistic, since it requires effort to determine that one is given the same task to solve twice.

Let W > 0 be a strict upper bound on the complexity of tasks in S, i.e.,

for $s \in S$, $c(s) < W$.

If $t \in T$, and i is an ordinal, let $pop(i,t)$ be the number of tasks $s \in t$ such that $c(s) = i$. Let

$$d(t) = \sum_{i<W} w^i \, pop(i,t).$$

For $t_1, t_2 \in T$, let $t_1 >> t_2$ iff t_2 is obtained from t_1 by replacing one of the tasks s of t_1 by a reduction $s_1,\ldots,s_n$. If $c(s_i) \geq c(s)$ for each i, then $d(t_2) < d(t_1)$. Then T is a task reduction system, with $\{\}$ as empty task and d as complexity function.

REFERENCES:

1 HALMOS, P., *Naive Set Theory*.

2 KURATOWSKI, K. & A. MOSTOWSKI, *Set Theory*, (chapter VII: Well-ordered Sets), North Holland, 1968.

3 BROUWER, L.E.J., *Zur Begründing der intuitionistischen Mathematik III*, Math. Annalen 96, pp. 451-488, Reprinted in L.E.J. Brouwer, collected works, edited by A. Heyting, North Holland, 1975.