-- --

Efficient Translation of External Input in a Dynamically Typed Language

Robert Paige [*]

Computer Science Department, New York University, 251 Mercer St., New York, NY 10012

New algorithms are given to compile external data in string form into data structures for high level datatypes. Let *I* be a language of external constants formed from atomic constants and from set, multiset, and tuple constructors. We show how to read an input string *C*, decide whether it belongs to *I*, convert it to internal form, and build initial data structures storing the internal value of *C* in linear worst case time with respect to the number of symbols in *C*. The algorithm does not require hashing or address arithmetic, but relies only on list processing.

A principal subproblem is to detect and remove duplicate elements from set-valued input. To solve this subproblem we extend the technique of multiset discrimination [2, 5] to detect all duplicate elements of a multiset, where these elements may themselves be tuples, multisets, or sets with arbitrary degree of nesting. To handle the case where the elements are multisets, we introduce a new technique called weak sorting, which sorts all of these multisets uniformly according to an arbitrary total order computed by the algorithm. The cost of computing this total order and of sorting all of the multisets is linear in the sum of the number of elements in each of the multisets. Our algorithms are based on a sequential pointer RAM model of computation [4, 7], which accesses and stores data using pointers but disallows address arithmetic (which precludes direct access into arrays).

This improves on previous algorithms used to solve the related reading problem in SETL [3, 6]. Those algorithms used hashing even for deeply nested data to detect duplicate values. If we assume that hashing unit-space data takes unit expected time and linear worst case time, then for arbitrary data their algorithm would require linear expected time and quadratic worst case time in the number of symbols in *C*.

Keyword Codes: F.3.2, I.1.1, I.1.2
Keywords: Semantics of Programming Languages; Expressions and Their Representation; Algorithms;

1. The Reading Problem

Consider an external read operation **read** *v* that inputs external data into program variable *v*. Our framework for understanding what this operation means depends in part on ascribing to *v* a *type* τ, which represents a set of abstract values $val(\tau)$. This framework also includes two maps. The first map $front\_end_\tau : ext(\tau) \rightarrow val(\tau)$ is a total, onto, many-to-one map from external representations $ext(\tau)$ to their abstract values. The second map $back\_end_\tau : val(\tau) \rightarrow imp(\tau)$ is a total one-to-many map from abstract values to their default implementations $imp(\tau)$. The composition $back\_end_\tau \circ front\_end_\tau$ defines the semantics of reading external data into a variable of type τ.

We will only consider a limited type system with a single elementary type *identifier* that represents the set of finite nonempty strings over a fixed finite alphabet Σ. For any type τ there are nonelementary types for finite sets $set(\tau)$ (i.e., unordered collections of values of type τ with no repeated values), finite multisets $mset(\tau)$ (i.e., unordered collections of values

* This research was partially supported by Office of Naval Research Grant N00014-93-1-1036 and Air Force Office of Scientific Research Grant No. AFOSR-91-0308. Part of this research was done while visiting DIKU at the University of Copenhagen.

of type τ with repeated values allowed), and finite tuples *tuple*(τ) (i.e., ordered collections of values of type τ indexed by numerals 1,2,...).

A multiset m with underlying set s is represented abstractly by a mapping f defined on s by the rule $f(x) = n$ iff x occurs n times in m. A tuple t with $k \geq 0$ components is represented abstractly by a mapping f defined by the rule $f(i) = x$ iff the ith component of t has the value x for $i=1,...,k$. A formal description of the core type system is found below, where **INT>0** stands for the set {1,2,...} of positive integers greater than 0, and *functions*(s,t) represents the set of total functions with domain s and range t. We support dynamic typing using tagged domains with strings *identifier*, *set*, *mset*, and *tuple* for tags. Tags also serve to make domains of different types disjoint. Thus, domain values for the empty set [*set*,{}], the empty multiset [*mset*,{}], and the empty tuple [*tuple*,{}] are distinguished by their tags.

$val(identifier) = \{[identifier,\ x]: x \in \Sigma^+\}$
$val(set(\tau')) = \{[set,\ s]: s \subseteq val(\tau') \textbf{ and } |s| < \infty\}$
$val(mset(\tau')) = \{[mset,\ f]: \exists s \subseteq val(\tau') \mid (|s| < infinity \textbf{ and } f \in functions(s, \textbf{INT>0}))\}$
$val(tuple(\tau')) = \{[tuple,\ f]: k=0,1,...,\ f \in\ functions(\{1,...,k\},\ val(\tau'))\}$

The preceding core type system is extended with a single recursive type *external*, where *val*(*external*) is the value of the following least fixed point:

$lfp\ t.\ (val(identifier) \cup val(set(t)) \cup val(mset(t)) \cup val(tuple(t)))$

In order to let the type of program variable v in the **read** statement be completely determined by the input, we assume that v is of type *external*. Hence, the semantics of reading is restricted to a definition for $back\_end_{external} \circ front\_end_{external}$.

Let the external data language *ext*(*external*) be defined by the following context free grammar with lexical category *identifier* and nonterminal *external* serving as the start symbol:

```
external = identifier          |
        '{' external * '}' |   -- sets
        '<' external * '>' |   -- multisets
        '[' external * ']'     -- tuples
```

Then $front\_end_{external}$ is defined recursively on the terms of language *ext*(*external*) by the following rules:

$front\_end_{external}(x) = [identifier,\ x]$, where x is an *identifier*
$front\_end_{external}(\{x_1,...,x_k\}) = [set, \{front\_end_{external}(x_i): i=1,...,k\}]$
$front\_end_{external}(<x_1,...,x_k>) = [mset, \{\ [front\_end_{external}(x_i),\ |\{j: j=1,...,k \textbf{ and }$
$front\_end_{external}(x_j)=front\_end_{external}(x_i)\}|\ ]: i=1,...,k\ \}]$
$front\_end_{external}([x_1,...,x_k]) = [tuple, \{[i, front\_end_{external}(x_i)]: i=1,...,k\}]$

A simple list-processing implementation is defined in terms of two auxiliary functions. The first function *record*$(x_1,...,x_k)$ takes an arbitrary number of arguments, and returns a pointer to a one-way list that stores x_i in the ith list cell for $i=1,...,k$. The second function *list*(x,t) takes an argument x and a tuple-valued argument t and returns the pointer *record*$(x, first, last)$, where *first* and *last* are pointers to the first and last cells of a doubly linked list storing the ith component $t(i)$ of tuple t in the ith list cell for $i=1,..., |t|$.

Map $backend_{external}$ is defined recursively and nondeterministically by the following rules:

$back\_end_{external}([identifier,\ x]) = record(identifier, x)$
$back\_end_{external}([set,\ s\,]) = list(set,\ [back\_end_{external}(x): x \in\ s\,])$
$back\_end_{external}([mset,\ f\,]) = list(mset, [back\_end_{external}(x): x \in\ domain\ f,\ i=1,...,f(x)])$
$back\_end_{external}([tuple,\ f\,]) = list(tuple, [back\_end_{external}(f(i)): i=1,..., |f\,|])$

The nondeterminism stems from having to arrange the elements of abstract sets in an arbitrary order before mapping them into list implementations.

The same default physical structure is used for sets, multisets, and tuples with element type τ. Data entry is through a pointer to two fingers pointing to both ends of the body, which is a two-way linked list that stores the element values of type τ. In assigning or incorporating records and lists (as in the preceding recursive rules defining $back\_end_{external}$), only the entry (a pointer) gets copied.

2. Solution To The Reading Problem

Semantic maps $front\_end_{external}$ and $back\_end_{external}$ are implemented in three stages. The first stage does input recognition, and generates the abstract syntax tree (abbr. AST). Since the grammar generating the external data language $ext(external)$ can be easily made to be LL(1), we will implement $front\_end_{external}$ using an LL(1) parser [1].

We consider an AST that reflects the type and nested structure of the input. Such an AST can be formed from the parse tree by rewriting to remove brackets and to label internal nodes by appropriate tags - either *set*, *mset*, or *tuple*. Consequently, every AST corresponds to a unique value in $val(external)$, and every value in $val(external)$ has at least one AST that would be formed from some input string by parsing and rewriting. It is straightforward for the parser to validate the input string C, and to transform correct input directly into an AST that represents the unique abstract value $v \in val(external)$ associated with C in linear time in the number of symbols in C.

In order to support the algorithm implementing the next stage, the AST needs to be described in somewhat greater detail. If x is a record with field f, then we use the notation $x.f$ to retrieve the value of field f in record x. If x is a pointer to a variable y, then the notation $deref(x)$ is used to represent the value stored at y. Each node n in the AST is implemented as a record with four fields: *n.succ* is a pointer to the right sibling of n, *n.child* is a pointer to the leftmost child, *n.parent* is a pointer to the parent, and *n.type* is the *type* represented by the node. To simplify the implementation, we will also place a unique identifier **0** called the sentinel as the rightmost sibling of every internal node.

All the identifiers appearing in the input and the sentinel are stored in memory within an initial universe U of data items; each node in the AST representing an identifier or sentinel stores a pointer in its *child* field to its corresponding symbolic form in U. This can be achieved by the list processing implementation of lexical scanning and parsing described in [2]. It will be convenient to refer to a node of the AST and its record implementation interchangably.

The goals of the second stage of input processing are (i) to recognize subtrees of the AST that represent the same value, (ii) to prune subtrees of the AST that represent duplicate elements from sets, and (iii) to compress the AST into an abstract syntax dag (ASD) in which every two subtrees in the AST that represent the same abstract value are identified by a single subdag of the ASD. That is, every two nodes x and y of the ASD that represent equal values also share the same children; i.e., $x.child = y.child$. After stage two is performed, the set of all *child* pointers in the ASD represents all distinct data items appearing in the input.

Our tree-to-dag transformation is based on two algorithmic techniques. The first technique is a generalization of multiset discrimination (i.e. finding duplicate elements in a multiset) described in [2, 5]. The extension rests on a surprisingly simple idea. If U is a set of arbitrary values and x and y are pointers to two such values, then equality of $deref(x)$ and $deref(y)$ can be decided in unit time by comparing x and y. Here, U can be stored in any data structure that supports pointer access to its elements. If U is an *exogenous* set [8] of pointers to arbitrary values, where each distinct pointer references a distinct value, then comparing pointers x and y decides equality of pointers $deref(x)$ and $deref(y)$, which in turn decides equality of values $deref(deref(x))$ and $deref(deref(y))$.

We give new multiset discrimination algorithms tailored to three different element types - tuples, multisets, and sets. These algorithms exploit the following notion of equality. Two tuples are equal if they have the same number of components, and their elements are component-wise equal. Two multisets are equal if the two tuples, formed from sorting these multisets relative to an arbitrary linear ordering (formally defined later) of their elements, are equal. Set equality is the same as multiset equality after each set is purged of duplicates.

The second new technique, called 'weak sort', is used by the multiset discrimination algorithm for multisets with multiset elements. Weak Sort is a lexicographic sort of a multiset of strings over an unordered alphabet that is linearly ordered arbitrarily by the algorithm.

Our discrimination algorithms all make use of the following more basic procedure for multiset discrimination of pointers.

(*multiset discrimination of pointers*)

Let Q be a set of records x, where $x.child$ is a pointer to an element in a universe U. The goal is to compute the set $child\_pnters = \{x.child : x \in Q\}$, and for each element y in set $\{deref(z) : z \in child\_pnters\}$ compute the set $y.back$ of back pointers to all elements x of Q in which $x.child$ points to y. The solution involves a single linear time scan through the elements of Q.

Multiset discrimination of tuples can be solved in much the same way that multiset discrimination of strings was solved in [2] (based on the earlier array-based method found in [5]) - by repeated application of multiset discrimination of pointers. Let R be a set of tuple-valued nodes whose children collectively satisfy the following property: if x_1 and x_2 are two child nodes (not necessarily siblings) and $x_1.child \neq x_2.child$, then the values represented by x_1 and x_2 cannot be equal. In this case we say that the children of nodes in R are *dagified*. Let the set U of data items be represented by all the children of all the children of elements of R. The algorithm is sketched just below.

(*multiset discrimination of tuples*)

(i) Let Q be the set of records $\{deref(u.child) : u \in R\}$. Let P be a workset initialized to $\{Q\}$.

(ii) While P is nonempty, remove a set Q from P and perform multiset discrimination of pointers on Q (according to the method described above) to obtain $child\_pnters = \{x.child : x \in Q\}$ and sets $y.back$ of back pointers for each child node $y \in \{deref(z) : z \in child\_pnters\}$. For each such child node y that represents the sentinel symbol, then the parents of each node $deref(w)$ for $w \in y.back$ have child pointers to distinct copies of the same tuple. Choose one of these copies, say from the first element w_1 in bucket $y.back$, and reassign the child pointer of the parent of each node $deref(w)$ for $w \in y.back$ to the location of the leftmost sibling of $deref(w_1)$; i.e., to precisely $deref(deref(w_1).parent).child$. Otherwise, if y does not represent the sentinel symbol, and if $y.back$ contains more than one element, then add the new set $\{deref(deref(w).succ) : w \in y.back\}$ to P. Both $child\_pnters$ and sets $y.back$ for each $y \in \{deref(z) : z \in child\_pnters\}$ need to be re-initialized after each of the preceding passes of multiset pointer discrimination.

Based on proofs found in [2, 5], we know that the preceding algorithm will dagify the nodes in R if the children of these nodes are already dagified. Analysis also follows from these sources. Let m be the total number of all children of nodes in R. Let m' be the total number of pointers in the shortest prefixes needed to distinguish each of these tuples. The multiset tuple discrimination algorithm just above only scans the child pointers of these shortest prefixes. It runs in worst case time $O(m')$ and worst case auxiliary space $O(|R|)$.

Recall that equality of multisets is defined in terms of equality of tuples formed by sorting these multisets relative to an arbitrary total ordering of their elements. By this definition we can solve multiset discrimination of multisets by reduction to multiset tuple discrimination. Let R be a set of multiset-valued nodes whose children are dagified. The following procedure solves this discrimination problem:

(*multiset discrimination of multisets*)

(i) Perform multiset pointer discrimination on the set Q of ALL (not just leftmost) children of nodes in R. The arbitrary order in which the elements of *child_pnters* = $\{x.child : x \in Q\}$ are encountered and stored creates an arbitrary linear ordering on the children of R. We can then use this ordering to rearrange all of the multisets by the following procedure, called *weak−sort*.

(ii) (weak-sort) Assign *nil* to the child pointers of each node in R. Next, search through the ORDERED set *child_pnters*, which represents the arbitrary linear ordering. For each pointer z in *child_pnters* encountered in this search, pass through each back pointer w in *deref* (z).*back* and perform assignments *deref* (w).*succ* := *deref* (*deref* (w).*parent*).*child* and then *deref* (*deref* (w).*parent*).*child* := w.

(iii) Perform multiset discrimination of tuples on the multisets of R ordered in the preceding step.

The performance of the preceding algorithm is the same as for multiset discrimination of tuples, except that parameter m' must be replaced by m, because weak-sort requires that every child of every node in R must be scanned.

Multiset discrimination of sets follows the same logic as multiset discrimination of multisets with one exception. Weak-sort must be modified so that duplicate set elements are eliminated. This is achieved by performing the two assignments only when they don't introduce a duplicate value into the set; i.e., only when *deref*(*deref*(*deref* (w).*parent*).*child*).*child* ≠ *deref* (w).*child*. The performance is the same as before.

We can now describe the general algorithm for solving stage two; i.e., converting the AST T produced by stage one into the compressed ASD D. We exploit the fact that data represented by nodes at different heights of the tree T cannot be equal. Let d be the depth of T.

(*tree−to−dag algorithm*)

(i) For each height $h = 1,2,...,d$ perform Step (ii).

(ii) Partition all nodes in T at height h into set-valued nodes R_{set}, tuple-valued nodes R_{tuple}, and multiset-valued nodes $R_{multiset}$. Then perform the datatype-specific form of multiset discrimination on each of these sets.

The correctness depends on an inductive argument with the following basis: the leaves of T are dagified to begin with. If T has n nodes, then the preceding algorithm produces D in $O(n)$ worst case time and space.

In the final stage of input processing, data represented in the ASD is converted into internal form. To support this last algorithm we now consider each node record n in the ASD as containing a new field *n.value*, which stores a pointer to the list-based implementation of the value represented by n. The algorithm also make use of functions *record* and *list* mentioned in the last section.

Let *root* denote the root node of the ASD associated with program variable v to be assigned input by the **read** statement. The algorithm below processes dag nodes in a depth-first-search starting from the root node.

```
v := imp(root);
procedure imp (node)     --obtain data structure
 if node.value = Ω then
  if deref (node.child).value = Ω then
   if node.type = identifier then
     node.value := record (node.type, deref (node.child))
   else                      --node.type is either set, mset, or tuple
     node.value := list (node.type, [imp (deref (n)):n=node.child ,deref (n).succ,...,until n =nil ])
   end if
  else                       --share the data structure
```

```
      node.value := deref (deref (node.chile).parent).value
    end if
  end if
  return node.value              --return data structure
end procedure
```

This final procedure is easily seen to take time proportional to the number of edges in the ASD. Hence, we have our main algorithmic result.

THEOREM . *For an input string of m symbols over a fixed alphabet* Σ *of k symbols, where* $k \geq 2$*, the algorithm above finds all redundant data in the input string without hashing or address arithmetic in worst case time and space O(m).*

REMARK: Avoiding address arithmetic depends on a primitive *getchar* operation that reads one character of external input, and returns a pointer to a unique record that stores the symbol. If the input character *c* was not previously input, then a new record uniquely associated with *c* is created. In the absence of such a primitive, we can limit address arithmetic to a single array of size $k+1$ associated with the alphabet.

3. Conclusion

A general principle of program improvement by eager evaluation is to push computation as far as possible towards the beginning of a program (e.g. code motion). The earliest point at which this is possible is often a **read** statement. To push computation beyond that point is to place it into the hands of the hapless user, which is what the standard algorithm texts seem to suggest in their ubiquitous requirements for preconditioned input. We have suggested that data entry and conversion may be useful points in a program for pipelining potentially expensive computations.

We have also shown that hashing is not needed (and may not even be desirable) for efficient recognition of redundant values occurring in external input. This surprising conclusion corrects the contrary view, which prevailed in the SETL and our own APTS projects. Although our type system, input language, and datatype implementations are greatly restricted, the overall treatment of reading extends to a far more general and practical context. Those results will be described in a subsequent paper.

References

1. Aho, A., Sethi, R. and Ullman, J., *Compilers,* Addison-Wesley, 1986.
2. Cai, J. and Paige, R., ‘‘Look Ma, No Hashing, And No Arrays Neither",’’ in *ACM POPL*, pp. 143 - 154, Jan, 1991.
3. Dewar, R., Grand, A., Liu S. C., Schwartz, J. T., and Schonberg, E., ‘‘Program by Refinement as Exemplified by the SETL Representation Sublanguage,’’ *TOPLAS*, vol. 1, no. 1, pp. 27-49, July, 1979.
4. Knuth, D. E., *The Art of Computer Programming, Vol 1: Fundamental Algorithms,* Addison-Wesley, 1973.
5. Paige, R and Tarjan, R., ‘‘Three Efficient Algorithms Based on Partition Refinement,’’ *SIAM Journal on Computing*, vol. 16, no. 6, Dec., 1987.
6. Schwartz, J., Dewar, R., Dubinsky, D., and Schonberg, E., *Programming with Sets: An introduction to SETL,* Springer-Verlag, 1986.
7. Tarjan, R., ‘‘A Class Of Algorithms Which Require Nonlinear Time To Maintain Disjoint Sets,’’ *J. Comput. Sys. Sci.*, vol. 18, pp. 110-127, 1979.
8. Tarjan, R., *Data Structures and Network Algorithms,* SIAM, 1984.