Memo: Implementing a small imperative language with safe dynamic allocation

Darko Stefanovic
Object Systems Laboratory and *Fox Project*

April 1, 1993

Abstract. In this memo, we discuss an implementation of Modula-X, a small imperative language with dynamic allocation and garbage collection. We first describe the language itself and its origins. We then describe our compilation model, and illustrate the front-end and the back-end. We analyse the implications of incorporating safe dynamic storage allocation. Finally we give some performance measurements.

1 The language

Modula-X is the name given to a small, Pascal-like subset of Modula-3, which has been used in the basic compiler construction course at UMass under Eliot Moss and also in the advanced language implementation classes at UMass and CMU. (See [4] for a description of a testbed for optimisation techniques based on the same language.) The language inherits procedural style and block structure and a solid repertoire of control flow constructs from Modula-3. (It lacks modules and exeptions, however.) It provides records and fixed-bound arrays as type constructors, and integer subranges and booleans as base data types.

In this implementation, we decided to limit procedure parameter modes to pass-by-value (VALUE) and pass-by-reference (VAR), and eliminate the READONLY mode because of its obscure semantics. For practical reasons (see below) we disallowed nested comments.

We added reference types and dynamic storage allocation for values of such types. To make reference types useful, we added a new kind of declaration for recursive types, much in the style of ML. Declaration TYPEREC can be used to declare several types simultaneously, provided that mutual recursion is limited to reference types. We believe this to be a good compromise between the approach of Modula-3, which allows unrestricted declaration ordering, and the Pascal style which requires cumbersome forward declarations. Our reference types are strictly monomorphic, in keeping with the rest of the language. In implementing reference types, we decided to use garbage collection rather than explicit deallocation, because of safety concerns, and because we could draw on our experience in garbage collector implementation.

See Appendix for the abstract syntax of the language and some concrete program examples.

2 Compilation model

The compiler is multi-pass; the front end comprises the parser and the annotator. The back end (currently) is a single pass code generator. The compiler itself is written in Standard ML, and each pass is implemented as a functor, which should simplify maintenance.

For the parser we decided to use ML-Yacc and ML-Lex, and we took and augmented the existing Yacc source for Modula-X and an ML-Lex source for Pascal (here we had to sacrifice nested comments). The parser builds an abstract syntax tree. The annotator analyses the abstract syntax tree, and performs a thorough semantic analysis. Static environments are computed, references resolved, all expressions type-checked. A fat annotated syntax tree results.

On our agenda we have several alternative proposals for implementing the back end, in particular, closure conversion within (or without) the context of continuation-passing style. However, we decided to produce a working compiler first, to serve as a sort of baseline in comparisons. We chose C as a target language over any particular machine code because of its availability, portability, and ease of use. After all, C has been described as a good high-level assembly language. We chose to rely on GNU C in particular to simplify the translation task further, because we can map nested procedures directly.

Nevertheless, the translation is not entirely straightforward. For example, an obstacle lies in structural equivalence of Modula-X vs. name equivalence of C. Essentially, it is not possible to declare variables by writing out their types each time. Instead, we collect all types effectively used in the entire program, place them in a graph, give them names, and then produce the equivalent C `typedefs` in a correct order. Since C doesn't allow assignment of arrays, the equivalent of an array is a `struct` with one field, thus, all `typedefs` are structures or pointers. Another restriction of C forced comparisons for non-scalars to be converted into `memcpy` calls.

3 Dynamic allocation and garbage collection

We use the UMass Garbage Collection Toolkit, a generational copying collector (see [3]) for details). This is the fourth language it has been applied to, after Smalltalk, ML, and GNU Modula-3.

For allocation, we use a contiguous region of memory called the nursery. Garbage collection is triggered when this region overflows. We use an explicit check for the purpose. (The alternative is to use memory protection mechanisms.) Large objects are immediately allocated in a separate area and are not copied at collection time. (We could statically determine the size of objects and inline either of the allocation calls, but we forgot to do so in the present compiler.)

To start a collection, we must be able to find all the references in the stack. (See [1] for a solution of this problem in a real implementation.) These references have been mapped to C pointers, and they are certainly at known offsets within the C stack frames. However, since we do not have a way of syntactically expressing the frames, in the presence of recursion we must incur a run-time cost, and keep a dedicated stack for collection root pointers. Therefore, we emit code at the beginning of each block to push the addresses of any locally declared pointers, and we emit code at the end of each block (and also before procedure returns and loop exits) to discard them. Note also that taking the addresses of these pointers and storing them keeps the C compiler from performing optimisations, such as pointer arithmetic or register allocation, that could compromise code correctness under garbage collection.

We must also keep track of intragenerational references. We use remembered sets to implement the write barrier. (See [2] for a discussion and comparison of this and other approaches). There is one remembered set for old pointers into each generation and one, the process set, for newly recorded pointers. For each assignment to an l-value of a reference type, the compiler tries to establish whether it lies on the stack or in the heap, and in the latter case, emits code to update the process set. (Note that if the l-value is a parameter passed by reference, the compiler cannot in general know where it is; if it is on the stack, the collector interface then discards this pointer at run-time because its source address is seen not to be in object space.)

4 Performance

We didn't pay too much attention to perfomance issues in this work, but some measured results are in order.

The two test programs were as follows.

- Ramanujan, originally written in Pascal, a large integer arithmetic implementation, which uses records and arrays, but no dynamic allocation. The running time is 43.9 seconds if the resulting C code is compiled using gcc with no optimisations, and 10.4 seconds using gcc -O3.

  A good part of time is spent doing range checks. If we were allowed to omit all run-time range checks in Modula-X, the way Pascal does, (or if we proved they weren't necessary), our time would go down to 5.8 seconds.

For comparison, the running time of the original Pascal program, compiled using Unix pc with no optimisations is 15.9 seconds, and 8.8 using pc -O.

- Destroy-Config1 (see Appendix), originally written in Smalltalk and Modula-3, which repeatedly allocates a very large tree and replaces some subtrees
  Under the first implementation of gc: 1401.6 sec unoptimised, 682.9 sec optimised.
  Unnecessary store checks removed: 1379.6 sec, 674.8 sec
  Removed 'asserts' and gc printouts: 1057.8 sec, 573.0 sec
  The above are for a nursery size of 256K; considerably better performance is obtained with larger nurseries (for this extreme benchmark):
  2.5M — 364.6 sec
  5M — 242.9 sec
  7.5M — 147.2 sec
  25M — 70.2 sec

All measurements on a DecStation 5000/200.

References

[1] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, CA, June 1992. *ACM SIGPLAN Not. 27*, 7 (July 1992).

[2] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992.

[3] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, MA 01003, September 1991. Submitted for publication.

[4] Norm Walsh. Constructing a framework for compiler optimization. University of Massachusetts, Amherst, 1992.

A Abstract syntax

```
(* The abstract syntax of Modula-X, expressed
```

```
  as several recursive data types in SML
*)

signature MODULAX_SYNTAX =

sig

datatype Program =
             PROGRAM of Module

and      Module =
             MODULE       of (* id: *) string
                           * (* block: *) Block

and      Block =
             BLOCK        of (* OptDecls: *) Declaration list
                           * (* Stmts: *) Stmt list

and      Declaration =
             DECLCONST    of (* id: *) string
                           * (* type: *) Typ option
                           * (* Expr: *) Expr
           | DECLTYPE     of (* id: *) string
                           * (* type: *) Typ
           | DECLTYPEREC  of ((* id: *) string
                           * (* type: *) Typ) list
           | DECLVAR      of (* idlist: *) string list
                           * (* type: *) Typ option
                           * (* Expr: *) Expr option
           | DECLPROC     of (* id: *) string
                           * (* signature: *) Signature
                           * (* Blocknoscope: *) Block

and      Signature =
             SIGNATURE    of (* Optformals: *) Formal list
                           * (* OptDeclType: *) Typ option

and      Formal =
             FORMAL       of (* idlist: *) string list
                           * (* optMode: *) Mode option
                           * (* type: *) Typ

and      Mode =
             VALUE
```

```
           | VAR
           | READONLY

and      Stmt =
             STMTBLOCK     of Block
           | STMTASSIGN    of Expr * Expr
           | STMTCALL      of ProcCall
           | STMTEXIT
           | STMTEVAL      of Expr
           | STMTFOR       of (* id: *) string
                            * (* from: *) Expr
                            * (* to: *) Expr
                            * (* optby: *) Expr option
                            * (* stmts: *) Stmt list
           | STMTIF        of (Expr * (* then part stmts: *) Stmt list)
                            (* some ELSIF clauses: *)
                            * (Expr * Stmt list) list
                            (* and an ELSE clause: *)
                            * (Stmt list) option
           | STMTLOOP      of Stmt list
           | STMTREAD      of (* format: *) string
                            * (* read list: *) Expr list
           | STMTREPEAT    of Stmt list
                            * (* until: *) Expr
           | STMTRETURN    of Expr option
           | STMTWHILE     of Expr
                            * Stmt list
           | STMTWRITE     of (* format: *) string
                            * (* write list: *) Expr list

and      Typ =
             TYPARRAY      of Typ list * Typ
           | TYPRECORD     of Fields
           | TYPSUBRANGE   of Expr * Expr
           | TYPID         of string
           | TYPREF        of Typ

and      Fields =
             FIELDS        of ((string list) * Typ) list

and      Expr =
             EXPROR        of Expr * Expr
           | EXPRAND       of Expr * Expr
           | EXPRNOT       of Expr
```

```
            | EXPREQUAL     of Expr * Expr
            | EXPRNEQUAL    of Expr * Expr
            | EXPRLESS      of Expr * Expr
            | EXPRGREATER   of Expr * Expr
            | EXPRLE        of Expr * Expr
            | EXPRGE        of Expr * Expr
            | EXPRADD       of Expr * Expr
            | EXPRSUB       of Expr * Expr
            | EXPRMUL       of Expr * Expr
            | EXPRDIV       of Expr * Expr
            | EXPRMOD       of Expr * Expr
            | EXPRUPLUS     of Expr
            | EXPRUMINUS    of Expr
            | EXPRID        of string
            | EXPRINT       of int
            | EXPRTEXT      of string
            | EXPRRECSEL    of Expr
                              * (* id: *) string
            | EXPRARRAYREF  of Expr
                              * Expr list
            | EXPRPROCCALL  of ProcCall
            | EXPRDEREF     of Expr
            | EXPRNEW       of Typ
            | EXPRNIL       of Typ

and      ProcCall =
              PROCCALL      of Expr
                              * Expr list

end
```

B Example program

```
MODULE Main;

TYPE CARDINAL = INTEGER;
TYPE CHAR = INTEGER;
TYPE REFANY = INTEGER;


CONST height = 9;
```

```
CONST branching_factor = 4;
CONST bulk = 0;
CONST destroy_depth = 3;
CONST niters = 100;


TYPEREC HeapNode = RECORD
  height: INTEGER;
  bulk: REF ARRAY [1..bulk] OF REFANY;
  children: ARRAY [1..branching_factor] OF REF HeapNode;
END;

TYPE Node = REF HeapNode;



(*****************************************************************)
VAR last: INTEGER := 0;
PROCEDURE random_int (max: INTEGER): INTEGER =
BEGIN
  last := (last * 17 + 5) MOD (max);
  RETURN last + 1;
END random_int;
(*****************************************************************)

PROCEDURE construct (height: CARDINAL): Node =
VAR anode: Node;
    i: INTEGER := 1;
BEGIN
  anode := NEW (Node);
(*  anode^.bulk := NEW (REF ARRAY [1..bulk] OF CHAR); *)
  anode^.height := height;
  IF (height = 1) THEN RETURN anode; END;
  WHILE (i <= branching_factor) DO
    anode^.children[i] := construct (height - 1);
    i := i + 1;
  END;
  RETURN anode;
END construct;

(* Walk down STEPS random_int steps and destroy the tree at that height and
   replace it by a new one *)

PROCEDURE destroy (root: Node; steps: INTEGER;
```

```
                 branching_factor, bulk: CARDINAL) =
VAR current: Node;
    temp: INTEGER;
BEGIN
  current := root;
  WHILE (steps > 1) DO
    (* generate a random number between 0 and branching factor to decide
       which child to go to *)
    WRITE("**DESCEND ONE STEP\n");
    temp := 1; (*random_int (branching_factor); *)
    current := current^.children[temp];
    steps := steps - 1;
  END;
  temp := 1; (* random_int (branching_factor);*)
  WRITE("**REPLACING...\n");
  current^.children[temp] :=
    construct (current^.height - 1);
END destroy;

VAR root: Node;
    i: INTEGER := niters;
BEGIN
  root := NEW (Node);
  WHILE (i > 0) DO
    WRITE("**CONSTRUCT:%d\n",i);
    root :=  construct (height);
    WRITE("**DESTROY:%d\n", i);
    destroy (root, destroy_depth, branching_factor, bulk);
    i := i - 1;
  END;
END Main.
```