Finding Dependencies on the `NUMBER` of *Modula-3*'s `CHAR` Type

Mike Spreitzer

September 25, 1992

1 Introduction

The *Modula-3* programming language is defined in chapter 2 of the book "Systems Programming With Modula-3", edited by Greg Nelson. On page 13, this book specifies the "predeclared" ordinal type `CHAR` to be

> an enumeration containing at least 256 elements,

and notes that

> the first 256 elements of the type `CHAR` represent characters in the ISO-Latin-1 code, which is an extension of ASCII.

While this specification does not assert that there are *only* 256 elements in type `CHAR`, that is the case in the only extant implementation of *Modula-3*. If one considers making an implementation in which there are many more characters, say 2^{16} or 2^{32}, the question naturally arises of how much code will have to be changed, and how can that code be reliably found? This paper describes programs that can be used to examine a corpus of code and estimate where changes would have to be made.

The *Modula-3* grammar is organized into five areas: types, statements, declarations, the module system, and expressions. The analysis roughly follows this division. We start by identifying ways that expressions and types can be affected by changing the `NUMBER` of `CHAR`. We then identify statements, declarations, and expressions that may be affected, and also some uses of the items declared in potentially-affected declarations. Code fragments concerning the module system are not affected.

2 Kinds of expressions and types, and closure operations

In describing the kinds of code fragment that are identified, we make repeated reference to a few kinds of expressions and types. This section makes those

$\mathcal{E}_s$	constant expression that introduces dependency
$\mathcal{E}_{cs}$	constant expression that depends
$\mathcal{T}_n$	type whose `FIRST`, `LAST`, or `NUMBER` depends
$\mathcal{T}_i$	type that will become impractical
$\mathcal{T}_r$	type that will be replaced
$\mathcal{T}_s$	type whose size introduces dependency
$\mathcal{TC}(\mathcal{T}_x)$	maps kind $\mathcal{T}_x$ to kind of types that contain a $\mathcal{T}_x$
$\mathcal{TCO}(\mathcal{T}_x)$	maps kind $\mathcal{T}_x$ to kind of object types that contain a $\mathcal{T}_x$
$\mathcal{T}_m$	type that depends in any way
$\mathcal{T}_d$	type distantly related to `CHAR`

Table 1: Auxiliary code fragment kinds and closure operators

preliminary definitions; the next section then explains the identification of code fragments in these terms.

We identify 2 kinds of dependent [on the number of `CHAR`][1] expression, 6 kinds of dependent type, and two closure operations on kinds of types. Each of these is denoted by a short symbol, and is defined in terms of others. These definitions are summarized in Table 1, and appear in the following subsections.

Note that while the classification of fragments of code depends, in some places and in some ways, on the meanings of certain subfragments in their contexts, we are discussing the identification of fragments of code, not the semantic objects that are manipulated by the [meaning of] the program.

Program semantics intrude, among other ways, in the necessity of finding the definitions of identifiers. The rules below omit this detail, assuming implicitly that identifiers are replaced with their definitions as necessary.[2] We also take the liberty of imagining that partial revelations `TYPE T` $<:$ `U` have been skolemized to `TYPE T =` Φ`(U)`.

2.1 $\mathcal{E}_s$, constant expression that introduces dependency

There are two kinds of expression that introduce a dependency:

- `BITSIZE`, `BYTESIZE`, or `ADRSIZE` applied to a type that "contains" a type whose size introduces a dependency.

[1] Henceforth we write simply "dependent" or "depends".

[2] Here's a more precise formulation. Wherever a pattern has a (proper) descendant type or constant expression node when parsed as the appropriate non-terminal of the *Modula-3* grammar, we understand the descendant node to match not only a type or constant expression of the form parsed into that node, but also an identifier ultimately (as far as the reference can see according to *Modula-3* semantics — i.e., not through un-revealed (partially) opaque types) equated to a type or constant expression of the form parsed into the node. For example, the pattern `NEW(REF` $\mathcal{TC}(\mathcal{T}_r)$`)` matches not only `NEW(REF SET OF CHAR)`, but also `NEW(T3)` if elsewhere the program defines `TYPE T3 = REF T2; T2 = SET OF T1; T1 = CHAR`.

- FIRST, LAST, or NUMBER applied to a type, or a value of a type, whose number depends on the size of CHAR.

We use a stylized language to describe such kinds of fragment. $\mathcal{E}_s$ is described thusly:

- BITSIZE|BYTESIZE|ADRSIZE ($\mathcal{TC}$($\mathcal{T}_s$))
- FIRST|LAST|NUMBER ($\mathcal{T}_n$)
- FIRST|LAST|NUMBER ($\langle$a $\mathcal{T}_n\rangle$)

Each of these patterns describes a code fragment that would be parsed into an "Expr" node according to the *Modula-3* grammar in section 2.8 of "Systems Programming With Modula-3". The pattern language is very similar to the metalanguage used in that section to define the *Modula-3* grammar. In our language, the vertical bar ('|', used to signify alternation) has the greatest binding power, and terminals are not quoted. Our names for kinds of fragments, and closure operations on them, are used like non-terminals in the grammar. The notation '$\langle$a $\mathcal{T}_x\rangle$' is used to stand for any fragment whose meaning is a value whose type specification is of kind $\mathcal{T}_x$; this is another way that program semantics intrude. In later patterns, the notation '$\langle$a $\mathcal{T}_x$ var$\rangle$' will be used; it matches any designator whose type specification is of kind $\mathcal{T}_x$. Ellipses ('...') will also be used; an ellipsis matches any fragment appropriate for the context of the ellipsis.

The first pattern for $\mathcal{E}_s$ is exact; the other two are conservative, due to the vagueness of $\mathcal{T}_n$.

2.2 $\mathcal{E}_{cs}$, constant expression that depends

Any constant expression constructed according to the rules of section 2.6 of "Systems Programming with Modula-3" from an $\mathcal{E}_s$ expression and other fragments is $\mathcal{E}_{cs}$. Note that this wording does not admit a constant expression constructed (using one of the rules of section 2.6.13) from a type that is in turn constructed from an $\mathcal{E}_s$,[3] because the construction of types is not in section 2.6; dependencies carried through types are caught by the rules for $\mathcal{E}_s$ when they re-appear in the expression part of the language.

2.3 $\mathcal{T}_n$, type whose FIRST, LAST, or NUMBER depends

- CHAR
- [... .. $\mathcal{E}_{cs}$]

[3] Such an expression is not admitted because of the dependency carried by the type; such an expression could be admitted because of a dependency carried by a sub-expression — for example, VAL(ORD(LAST(CHAR)), [FIRST(CHAR)..LAST(CHAR)]).

- `[`$\mathcal{E}_{cs}$ `..  ...]`
- `ARRAY` $\mathcal{T}_n$ `OF ...`

That is, the `CHAR` type itself, or any subrange whose upper or lower bound depends, or an array indexed by another $\mathcal{T}_n$. The collapsing into one kind of types whose `FIRST`, `LAST`, and `NUMBER` depend leads to conservatism in patterns that use this kind. For example, because `FIRST(CHAR)` is $\mathcal{T}_n$, so is `[0 .. FIRST(CHAR)]`, and thus `SET OF [0 ..  FIRST(CHAR)]` is considered (by the rules of the next subsection) a type that will become impractical.

2.4 $\mathcal{T}_i$, type that will become impractical

- `ARRAY` $\mathcal{T}_n$ `OF ...`
- `SET OF` $\mathcal{T}_n$

This is conservative not only because the kind $\mathcal{T}_n$ confuses `NUMBER`-dependency with `FIRST`-dependency and `LAST`-dependency, but also because $\mathcal{T}_n$ does not quantify how much its `NUMBER` will change. For example,

```
SET OF [0 ..  BITSIZE(['\000' ..  LAST(CHAR)])],
```

while increasing in size, will not actually become impractical.

2.5 $\mathcal{T}_r$, type that will be replaced

- $\mathcal{T}_i$
- `BITS` $\mathcal{E}_{cs}$ `FOR ...`
- `BITS ... FOR` $\mathcal{TC}(\mathcal{T}_s)$

The types that will have to be changed are those that will become impractical or statically erroneous; the last two patterns make a conservative identification of the latter case. While $\mathcal{T}_r$ is the kind of types that an enlargment of the `CHAR` type forces to change, the consequent re-design of code may lead to changes in other types; these programs do not attempt to do enough of that re-design to identify those other types.

2.6 $\mathcal{T}_s$, type whose size introduces dependency

- $\mathcal{T}_n$
- $\mathcal{T}_r$

When the `NUMBER` of a type changes, its size in memory generally also changes. Types that are edited will, in general, also change their size in memory. Types that "contain" a $\mathcal{T}_s$ will also change their size, as a consequence of that containment; such consequently-changing types are identified by $\mathcal{TC}(\mathcal{T}_s)$.

2.7 $\mathcal{TC}(\mathcal{T}_x)$, maps kind $\mathcal{T}_x$ to kind of types that contain a $\mathcal{T}_x$

- $\mathcal{T}_x$
- `RECORD ... f:` $\mathcal{TC}(\mathcal{T}_x)$ `... END`
- `ARRAY ... OF` $\mathcal{TC}(\mathcal{T}_x)$
- `BITS ... FOR` $\mathcal{TC}(\mathcal{T}_x)$

The closure operator $\mathcal{TC}$ maps a kind $\mathcal{T}_x$ of type fragment to the kind of type fragments whose memory layout linearly includes a $\mathcal{T}_x$.

2.8 $\mathcal{TCO}(\mathcal{T}_x)$, maps kind $\mathcal{T}_x$ to kind of object types that contain a $\mathcal{T}_x$

- `... OBJECT ... f:` $\mathcal{TC}(\mathcal{T}_x)$ `... END`
- $\mathcal{TCO}(\mathcal{T}_x)$ `OBJECT ... END`
- $\Phi(\mathcal{TCO}(\mathcal{T}_x))$

2.9 $\mathcal{T}_m$, type that depends in any way

$\mathcal{T}_m$ is almost, but not quite, the type fragments that mention `CHAR` in any way. The exceptions are due to "default values" for object and record fields and for procedure arguments: mentions of `CHAR` in those expressions do not qualify a type fragment for inclusion in $\mathcal{T}_m$. Every other way that a type can mention `CHAR` leads to a dependency of the represention of the constructed type, or of types "reachable" from it, on `NUMBER(CHAR)`. Thus, `PROCEDURE (c: CHAR)` is in $\mathcal{T}_m$, but `RECORD i: INTEGER := ORD(LAST(CHAR)) END` is not.

2.10 $\mathcal{T}_d$, type distantly related to `CHAR`

- $\mathcal{T}_m \setminus \mathcal{TC}(\mathcal{T}_s)$

The distantly-related types are those whose memory layout does not directly contain a field dependent on `NUMBER(CHAR)`, but does in some way depend.

3 Code fragments identified

3.1 Types to change

- $\mathcal{T}_r$

This kind of fragment has already been discussed. Such fragments are flagged with the message "`type needs changing`".

3.2 Statements to think about

We include declarations, of fields and of formal arguments as well as of variables, in this kind. These patterns are all conservative.

1. `(INC|DEC) (`$\langle$`a` $\mathcal{T}_n$ `var`$\rangle$`, ...)`
2. assignment to/from $\mathcal{TC}$`(ARRAY` $\mathcal{T}_n$ `OF ...)`
3. assignment to a $\mathcal{TC}(\mathcal{T}_i)$ variable
4. declaration of an `<*EXTERNAL*>` $\mathcal{T}_m$
5. declaration of an object field whose type is in $\mathcal{TC}(\mathcal{T}_r)$

These patterns are flagged with messages of the following forms, respectively (where N is replaced by the name of the flagged item):

1. `INC/DEC of a NUM(CHAR)-dependent type`
2. `assignment to/from NUM(CHAR)-dependent array [at formal` N`]`
3. `Copy of becoming-huge value [at formal` N`]`
4. `Decl of EXTERNAL Tm item` N
5. `Object field containing changing type`

Pattern 1 is needed because changing `LAST(CHAR)` changes the range of possible post-values for the variable, and may change whether such a statement gets a checked runtime error. Pattern 2 is needed because the rules for array assignability depend only on the `NUMBER` of the index type, not its full semantics; thus, an assignment between an `ARRAY [0..255] OF T2` and an `ARRAY CHAR OF T2` could change from valid to invalid. This pattern thus applies everywhere a value is tested for assignability to a type; the list of such places is:

- assignment statements,
- actual arguments passed to procedures (including `NEW`) or methods,
- the argument in a `RAISE` statement with an argument,
- the expression in a `RETURN` statement from a function,
- defaults for variables, fields, and formal arguments.

Pattern 3 identifies code fragments that create a mutable copy of a value of a hypothetically-impractical type; if such a type is changed to a reference to a complicated data structure, such code no longer makes a new mutable copy of the value (e.g., the set of characters) of interest to the program — it only copies the reference to the representation of that value. This pattern thus applies everywhere a value is stored into a mutable variable; the list of such places (and the place flagged, if non-obvious) is:

- assignment statements,
- the initialization of the variable to which a `VALUE` formal is bound during the execution of a procedure call (the formal is flagged at its declaration and at each call site),
- the initialization of fields of a `REF RECORD` or `OBJECT` created in a `NEW` call (the initial value expressions in field declarations are flagged, as are the bindings supplied in the `NEW` call),
- the initialization of v_i in an arm of a `TRY-EXCEPT` statement,
- the initialization of a variable given an initial value.

Pattern 4 is needed because declarations of external items with $\mathcal{T}_m$ types expose *Modula-3*'s representation of `CHAR` to code written in other languages. Pattern 5 is needed because object types can be hidden: rule 4 of the next subsection will not catch cases where the fact that a type "really is" $\mathcal{TCO}(\mathcal{T}_r)$ is due to a revelation not in scope.

3.3 Expressions with straightforward replacements

If `NUMBER(CHAR)` is greatly increased, a few standard data structures (some generic) will probably be provided as replacements for $\mathcal{T}_i$ types. This will require transforming all code that creates and manipulates values in these types. Such transformations will probably be rather regular and straightforward (one might even imagine writing a program to do them). The replacements for the other $\mathcal{T}_r$ types will probably also require rather regular and straightforward transformations of code that creates and manipulates their values. This subsection syntactically identifies most of the code fragments that create or manipulate values of $\mathcal{T}_r$ types.

1. $\langle$a $\mathcal{T}_r\rangle$[...]
2. $\mathcal{T}_r${ ...}
3. NEW(REF $\mathcal{TC}(\mathcal{T}_r)$, ...)
4. NEW($\mathcal{TCO}(\mathcal{T}_r)$, ...)
5. $\langle$a $\mathcal{T}_r\rangle$ binop $\langle$a $\mathcal{T}_r\rangle$
6. ... IN $\langle$a $\mathcal{T}_r\rangle$

Fragments matching these patterns are flagged with messages of the following forms, respectively:

1. `indexing into changing array`

2. `cons of val in changing type`
3. `NEW of container of changing type`
4. `NEW of container of changing type`
5. `binary operation on val in changing type`
6. `binary operation on val in changing type`

3.4 Expressions that require harder thought

1. $\mathcal{E}_s$ not in a type construction
2. use of an external $\mathcal{T}_m$
3. SUBARRAY($\langle$a $\mathcal{T}_r\rangle$, ...)
4. ORD($\langle$a $\mathcal{T}_n$ var$\rangle$)
5. VAL(..., $\mathcal{T}_n$)
6. ADR($\langle$a $\mathcal{TC}(\mathcal{T}_s)\rangle$)
7. LOOPHOLE($\langle$a $\mathcal{TC}(\mathcal{T}_s)\rangle$, ...)
8. LOOPHOLE(..., $\mathcal{TC}(\mathcal{T}_s)$)

Fragments matching these patterns are flagged with messages of the following forms, respectively:

1. `Expr depends on NUM(CHAR)`
2. `Use of EXTERNAL Tm item` N
3. `SUBARRAY of a changing array`
4. `ORD(var in NUM(CHAR)-dependent type)`
5. `VAL(..., NUM(CHAR)-dependent type)`
6. `ADR of a CHAR-size-dependent type`
7. `LOOPHOLE from a CHAR-size-dependent type`
8. `LOOPHOLE to a CHAR-size-dependent type`

The first pattern says that an $\mathcal{E}_s$ that contributes to making a type $\mathcal{T}_n$ or $\mathcal{T}_r$ does not need to be flagged because the consequences of the type being $\mathcal{T}_n$ or $\mathcal{T}_r$ are flagged, but other expressions that are $\mathcal{E}_s$ should be flagged because these programs know nothing about the consequences of that expression being $\mathcal{E}_s$. The second pattern is just an aid to tracking down the consequences of a declaration of an external $\mathcal{T}_m$ item, which is flagged according to an earlier subsection. The third pattern is in this subsection, rather than the previous one, because there are data structures that can represent a mapping from `CHAR` to `X` but do not readily support an operation analogous to `SUBARRAY`. Pattern 4 is flagged because the range of possible results depends — but only when applied to variables, because constants won't change their ordinal value. Pattern 5 is flagged because the range of valid values for the first argument depends (even when the first argument is constant). The last three patterns identify fragments that reveal *Modula-3*'s representation of `CHAR`.

3.5 Expressions distantly related to CHAR

These patterns identify code fragments that reveal *Modula-3*'s representation of a type related to, but not containing, a `CHAR`. These are definitely conservative.

1. `ADR(`$\langle$a $\mathcal{T}_d\rangle$`)`
2. `LOOPHOLE(`$\langle$a $\mathcal{T}_d\rangle$`, ...)`
3. `LOOPHOLE(..., `$\mathcal{T}_d$`)`

Fragments matching these patterns are flagged with messages of the following forms, respectively:

1. `ADR of a type related to CHAR`
2. `LOOPHOLE from a type related to CHAR`
3. `LOOPHOLE to a type related to CHAR`