MIA 99-41: BETA Language Modifications - Reference Manual

Table of Contents

Copyright Notice

Mjølner Informatics Report
MIA 99-41
August 1999

BETA Language Modifications

BETA Language Modifications - Reference Manual

Mjølner Informatics Report, MIA 99-41, August 1999

1. Introduction

The BETA language is described in [MMN 93] which is the main reference to BETA. The BETA book is a tutorial description of BETA. As of today there is no language definition manual, but one in preparation. Short introductions to BETA may be found in [Madsen 99], and [MIA 94-26]. In addition the following documents are available:

- A structured context-free grammar for BETA .
- A summary of BETA Terminology.
- A Quick Reference Card for BETA.

This document describes modifications of the BETA language that have been made since the publication of the BETA book. These include modifications, clarifications as well as extensions:

- A number of new basic patterns have been introduced.
- The operations on basic patterns such as integer, char, boolean and real, and references are further specified.
- Repetition constructors have been introduced.
- Text literals are further specified and extended and support for UniCode text has been included.
- For imperatives, the type of a general imperative must be a basic pattern or a reference, a simple if-imperative has been introduced, the labeled compound imperative has been eliminated. There are various restrictions with respect to leave/restart of patterns. Inserted objects are implemented as dynamic objects.
- Virtual superpatterns are not implemented, but a final pattern may be used as a superpattern.
- The use of `this(p)[]` for component objects is specified.
- Low level primitives for bit manipulation.
- Finally there are some changes regarding dynamic denotations, concurrency, exception handling, structure objects and the fragment system.

In the following, the above mentioned changes will be described in detail. **The reader is assumed to be familiar with [MMN 93]**

2. Basic Patterns

The BETA book defines the following basic patterns: `boolean`, `char`, `integer`, and `real`. To support different sizes of integers and characters, a number of new basic patterns have been introduced. The following table shows the current basic patterns of BETA and their representation.

`int8`	signed 8-bit integer

`int8u`	unsigned 8-bit integer
`int16`	signed 16-bit integer
`int16u`	unsigned 8-bit integers (replaces pattern `shortInt`)
`int32`	signed 32-bit integer (identical to `integer`)
`int32u`	unsigned 32-bit integer
`integer`	signed 32-bit integer (identical to `int32`
`char`	8-bit ASCII character
`wchar`	16-bit UniCode character
`boolean`	boolean value `true` or `false`
`real`	64-bit floating point number

Notes:

- Pattern `shortint` will be eliminated and should no longer be used.
- Pattern `wchar` will eventually replace pattern `char`. I.e. characters will in the future be represented as 16-bit UniCode characters.

3. Operations on Basic Patterns and References

In this section the rules for basic patterns- and references operations such as assignment, arithmetic operations, relational operations, boolean operations, etc. are further specified.

All expressions are evaluated as 32-bit values, either signed or unsigned. If an expression is assigned to a a variable representing 8- or 16-bit values, only the least significant 8- or 16-bits are assigned.

In the following a number of tables showing legal combinations of operands and result type for the operations on basic patterns and references will be given. Entries not shown are illegal. Entries marked with ***** are illegal. Entries marked with **!** will give a compiler warning, and may become illegal in a future version of BETA. The following abbreviations will be used:

Abbreviations:

int represents an evaluation of type **integer**

bool represents an evaluation of type **boolean**

iref represents an evaluation of type **item reference**

cref represents an evaluation of type **component reference**

sref represents an evaluation of type **structure reference**

NONE is both an **iref**, a **cref** and an **sref**.

3.1 Assignment

The following table shows the legal combinations of the left and right side of an assignment for basic patterns and references:

```
E -> V</PRE>
```

The rows of the table shows possible types of of E and the columns of the tables show possible types of of V. The elements of the table shows the result type, which for assignment are the same as the type of V.

Note that E and V stand for arbitrary evaluations including values as in:

```
a+b*10 -> E</PRE>
```

and references as in:

```
a[] -> S[]</PRE>
```

int
char
real
bool
iref
cref
sref
int
int
char
real
!
*****
*****
*****
char
int
char
*****
*****
*****
*****
*****
real
int
*****
real
*****
*****
*****
*****
bool
!
*****
*****

bool
*
*
*
iref
*
*
*
*
iref
*
*
cref
*
*
*
*
*
cref
*
sref
*
*
*
*
*
*
sref

Notes:

- Assignment between integer, and real is allowed.
- In assignments of a real value to an integer value, the real value is truncated.
- If an integer value is assigned to a variable of type int8, int8u, int16, int16u, the value may be truncated.
- Assignment between integer and char/wchar is allowed. Character constants have their ASCII or UniCode value. Assignment of an arbitrary integer value to char/wchar may result in truncation of the integer value.
- Assignment between instances of integer and boolean is currently allowed, but a warning is given. In a future version of BETA these assignments may not be allowed and may give an error. The patterns true and false have the values 1 and 0 respectively. Assignment of an arbitrary integer value to a boolean instances may result in truncation of the integer value.

3.2 Relational Operators

3.2.1 Restrictions on Relational Expressions

The relational operators

```
<, <=, >, >=
```

can only be used for the basic patterns integer, real, boolean, and char.

The relational operators:

```
<relate>
```

```
can in addition be used for references.
```

It is **not** possible to compare a list of values as in:

```
(# P: (# ... exit(e1,e2,e3) #);
   A: (# ... exit(f1,f2,f3) #);
   B: @boolean
do P = A -> B;
   P = (g1,g2,g3) -> b
#)
```

3.2.2 Type Rules for Relational Expressions

The following table shows the legal combinations of types for expressions of the form:

```
E1 = E2/PRE>
```

and

```
E1 <> E2/PRE>
```

int

char

real

bool

iref

cref

sref

int

bool

bool

bool

!

*

*

*

char

bool

bool

bool

*

*

*

*

real

bool

bool

bool

*

*

*

*

bool

!

*

*

bool

*

**
**
iref
**
**
**
**
bool
**
**
cref
**
**
**
**
bool
**
sref
**
**
**
**
**
**
bool

The following table shows the legal combinations of types for expressions of the form:

E1 -> E2

int
char
real
bool
sref
int
bool
bool
bool
**
**
char
bool
bool
bool
**
**
real
bool
bool
bool
**
**

bool
*****
*****
*****
bool
*****
sref
*****
*****
*****
*****
bool

3.3 Arithmetic Operators.

The following table shows the legal combinations of types for expressions of the form:

int
char
real
int
int
int
real
char
int
int
*****
real
real
*****
real

The following table shows the legal combinations of types for expressions on the form :

Note that the `div` operator is integer division.

int
char
int
int
int
char
int
int

The following table shows the legal combinations of types for expressions of the form: E1 / E2

Note that the `/` operator is a a real operator. The result is always a real, even if the operands are integers. If integer division is wanted, use the `div` operator.

int
char
real
int
real
real
real
char
real
real
*****
real
real
*****
real

char is likely to be eliminated as a legal operand for **/** in a future version.

3.4 Boolean Operators

3.4.1 Xor Primitive

An xor primitive is supported as a basic operation on booleans. If `E1` and `E2` are boolean expressions, then the expression

```
E1 xor E2 is an exclusive or between tt>E1/tt> and tt>E2/tt>.
```

3.4.2 Short-circuit Boolean Expressions

Boolean expressions are implemented as short-circuit.

That is, in
```
B1 and B2 PRE>
B2 is I>not/I> evaluated if B1 is true /P>
```

Similarly, in
```
B1 or B2 PRE>
B2 is I>not/I> evaluated if B1 is false /P>
```

3.4.3 Type rules for boolean expressions

The following table shows the legal combinations of types for expressions of the form: E1 and E2

bool
bool
bool

Unary Operators:

For any expressions of the form:

The legal types for E are: **int**, **char**, or **real**. The result type is the same as operand type

For any expression of the form:

The type of E must be **bool** and the result type is **bool**

4. Repetition Constructors

4.1 Value Repetitions

Consider a value repetition:
```
R: [exp] @T
```
where T is a basic pattern.

A repetition object may be constructed and assigned to R in the following way:
```
(E1,E2,...,En) -> R
```

where `E1, E2, ... En` are evaluations of type `T`.

A repetition object with `range = n` is constructed and `R[i] = Ei`, `i` in `[1,n]`.

A repetition object consisting of one element may be constructed using:

4.2 Example

The following is an example of using repetitions constructors:
```
(# a,b: @integer;
   R:[1] @integer
do 10->a; 20 -> b
   (a,b,a+b) -> R;
   (for i: R.range repeat R[i] -> putint; ' '-> put for)
#)
```
The above example will print the sequence

4.3 Variable number of enter parameters

Repetitions may be used to define patterns with a limited form of a variable number of parameters. Consider the following example:

```
foo:
   (# S: ^stream;
      V: [1] @integer;
      sep: @char
   enter(S[],V,x)
   do (for i: V.range repeat
              V[i] -> S.putint; sep -> S.put
      for)
   #)
```

Pattern foo may be used in the following way:

(screen[],(1,2,3,4,5),'-') -> foo</PRE>

which will print the following sequence to the screen:

1-2-3-4-5-</PRE>

5. Pattern `text` and `wtext`

Pattern `text` represents 8-bit ASCII texts.

Pattern `wtext` represents 16-bit UniCode texts. For details see the library `basiclib/wtext.bet`. Operations supporting conversion between `text` and `wtext` are available. In a future version of BETA, `wtext` may replace `text`.

5.1 String Literals as References

The pattern `Text` enters and exits a char-repetition. This means, that a text may be initialized using constant strings as follows:

```
   t: @text;
do 'hello' -> t;
```

Many operations involving texts, however, takes *references* to texts as enter/exit parameters. This is mainly for efficiency reasons.

To allow easy invocation of such operations on string literals, the following is *also* allowed:

```
   t: ^text;
do 'hello' -> t[];
```

The semantics of this is, that a text object is instantiated, initialized by the constant string, and finally assigned to the text reference. It thus corresponds to the following code:

```
do ] -> t[];
   'hello' -> t
```

5.2 Special Characters in String Literals

The following special characters are allowed in BETA string literals.

\a alert (bell) character

\b backspace

\f formfeed

\n newline

\r carriage return

\t horizontal tab
\v vertical tab

\\ backslash

\? question mark

\' single quote

\" double quote

\ooo octal number

`\ooo` can also be `\o` or `\oo`, if the character immediately following `\o` or `\oo` respectively, is not a digit.

Previous versions of BETA has allowed `''` to represent a quote `'` in strings as in `'Tom''s Cottage'`. **This is no longer allowed**. Quote must be represented using `\'` as in:
`'Tom\'s Cottage'</PRE>`

5.3 Text Literal Concatenation

A text literal cannot contain newlines. Alternatively a text literal may be written as a sequence of strings separated by white space as in:

```
'Lisa Nelson, '
'2454 West Street, '
'Palo Alto, CA 94304' -> T[]
```

This corresponds to:

```
'Lisa Nelson, 2454 West Street, Palo Alto, CA 94304' -> T[]
```

5.4 Text literals

Text literals like `'Hello'` may be considered as abbreviations of char repetition constructors like `('H','e','l','l','o')`. Consider:
`Q:H[1]@char</PRE>`

The evaluation:
`'Hello'->Q</PRE>`
may considered as an abbreviation for:
`('H','e','l','l','o')->Q</PRE>`

6. Imperatives

6.1 General If-Imperative

A general if-imperative has the form:

```
(if E0
 // E1 then ...
 // E2 then ...
```

```
 ...
 // En then ...
if)
```

with a possible else-part.

The value of `E0`, `E1`, `E2`, ..., `En` must be of type int, bool, char, real, iref or cref.

It is thus not possible to compare a list of values as in:

```
(# P: (# ... exit(e1,e2,e3) #);
   A: (# ... exit(f1,f2,f3) #);
do (if P
    // A then ...
    // (g1,g2,g3) then ...
   if)
#)
```

6.2 Simple If-Imperative

Often the following If imperative is used:

```
(if boolExp // true
 then ...
 else ...
if);
```

The current version of the compiler supports an extension to the BETA language called Simple If Imperative. This extension means, that the case-selector // may be omitted, if the evaluation on the left hand side exits a boolean. That is, the above may be written

```
(if boolExp
 then ...
 else ...
if);
```

Like in the general if-statement, the else part is optional.

6.3 The labeled compound imperative

The labeled compound imperative:

```
(L: Imp1; Imp2; ... ; ImpN :L)</PRE>
```

has been eliminated from the language. Instead the following construct may be used:

```
L: (# do Imp1; Imp2; ... ; ImpN #)</PRE>
```

Inserted items with no declarations and no superpattern will be inlined in the enclosing code. There will thus be no execution overhead compared to the old (never implemented) labeled compound imperative statement.

6.4 Leave- and Restart Imperative

It is in general not possible to use leave P or restart P where P is a pattern. P must in general be a label. *However*, the following has been implemented:

```
P: (#
   do ...
      leave P
      ...
```

```
      restart P;
      ...
   #)
```

Leave/restart from an inserted item, however, is *not* supported by the current version of the compiler:

```
P: (#
   do ...
      (#
      do ...
         leave P; (* ILLEGAL *)
         ...
         restart P; (* ILLEGAL *)
         ...
      #)
      ...
   #)
```

6.5 Inserted items

If P is a pattern, then an inserted item ([MMN93], section 5.10.2) may be specified as:

```
(# ... do ... inserted ... #)
```

Inserted items are implemented as dynamic items (&P).

Inserted components ([MMN93, Madsen93], section 5.10.3):

```
inner; (# ... #)
```

have not been implemented.

7. Virtual Patterns and Final Patterns as Superpatterns

A virtual pattern **cannot** be used as a super pattern as shown in the following example:

```
A::< (# ... #);
B: A(# ... #)
```

Previous version of BETA has supported the use of virtual patterns as super patterns, but due to efficiency considerations virtual super patterns are no longer supported.

A virtual pattern that has been *final bound* may be used as a superpattern as shown in the following example:

```
A:: (# ... #);
B: A(# ... #)
```

The situation may also occur in a more indirect way:

```
graph:
   (# node:< (# ... #);
      nodeList: @list(# element::< node #);
      ...
   #);
```

Here the virtual further binding of element in list is **not** allowed, since node is itself virtual.

The current version of the compiler will allow final binding using a pattern that is itself virtual. That is, you can do this:

```
graph:
  (# node:< (# ... #);
     nodeList: @list(# element:: node #);
```

```
    ...
  #);
```

8. The Use of `this(P)` for Component Objects

Consider the following example:

```
P: (#  A: (# X: ^P; (* reference to item qualified by P *)
       B: ^|P (* reference to component qualified by P *)
   do this(P)[] -> X[];   (* legal use of this(P)[] *)
      this(P)[] -> R[];   (* illegal use of this(P)[] *)
   #)
```

The compiler assumes that `this(P)[]` is a reference to an item object. Since an item reference cannot be assigned to a component reference the evaluation `this(P)[] -> R[]` is illegal.

It is, however, possible to use a run-time routine to convert an item reference to a component reference, provided that the item is part of a component. Consider:

```
(# P: (# B: |^ P
      do this(P)[] -> objectToComponent -> B[]
      #);
   X: |@ P;
   Y: @ P;
do X;  (* OK *)
   Y; (* a run-time error will happen *)
#)
```

When `X` is executed, the P-object is part of the component X and `objectToComponent` will return a reference to X.

When `Y` is executed, the P-object is not part of a component and `objectToComponent` will fail.

It is also possible to get a reference to the item-part of a component by using the pattern `componentToObject`:

```
x[] -> componentToObject -> R;
```

The patterns `objectToComponent` and `componentToObject` are placed in the library:

9. Dynamic denotations

In declarations like:

```
P: <AD>(# ... #);
X: @<AD>;
Y: ^<AD>;
```

it is checked that <AD> is a *static* denotation, where *static* is defined as follows:

· A name A is always static

· In a remote-name R.A, R must be a static object

· Use of THIS(A).T is static

· Only in Y: ^P.T, can P be a pattern

· Denotations using R[e], and (foo).bar are *not* static

This means that e.g. descriptors like:
```
R[e].A(# ... #)
(foo).bar(# ... #)
R.P(# ... #) where 'R' is a dynamic ref.
```
are only allowed in *imperatives*.

For Y: ^R.P where R is a dynamic reference, the compiler will currently report a warning and suggest to use

Y: ^A.P where A is the qualification of R.

Note: that when using --noWarnQua, this warning will *not* be printed. A future release may change the warning to an error.

10. Concurrency

There are some deviations with respect to the implementation of concurrency. Please consult [MIA90-8] before using the concurrency.

11. Exception Handling

The `Program` pattern as described in the chapter on exception handling in [MMN93] has not been implemented.

12. Pattern Variables/Structure Objects

If `P` is a pattern then `P##` is a structure object denoting the pattern `P`.

Similarly if `R` is an object, then according to the BETA book, a structure object corresponding to the descriptor/pattern used to instantiate `R` may be obtained using `P._struc`.

The expression `R##`, may also be used instead of `P._struc`.

13. Low Level Primitives

Low level primitives for bit manipulation are described in:

- Low Level Primitives

14. The Fragment System

A further specification including modifications of the fragment system is given in:

- The Fragment System: Further Specification