“System Verilog Tagged Unions and Pattern Matching”

(An extension to System Verilog 3.1 proposed to Accellera)

Bluespec, Inc.

Contact:
Rishiyur S. Nikhil, CTO, Bluespec, Inc.
200 West Street, 4th Flr., Waltham, MA 02541, USA
Email: nikhilbluespec.com Phone: +1 (781) 250 2203

November 19, 2003
(see rationale in previous docs Sep 9 and Oct 3)

© 2003 Bluespec, Inc.
(should Accellera adopt this proposal, copyright will transfer to Accellera)

Introduction

We submitted a proposal for Tagged Unions and Pattern Matching on Sep 9 (revised Oct 3). It was discussed for the first time at the SV-BC meeting Nov 10. This is a revised proposal based on the extensive feedback of Nov 10. It focuses on the proposal per se, omitting the extensive rationale in the previous document.

We have structured the proposal into 3 parts:

I. Tagged unions for type-safety and brevity.
Benefits: Type-safety and brevity. Type-safety improves correctness. Tagged unions also benefit formal verification because it improves the ability to reason about programs.

II. Pattern matching in **case** statements, **if** statements and conditional expressions.
Benefits: Dramatic improvement in brevity and readability. Makes it easier to implement tagged unions without runtime checks. Also improves formal reasoning.

III. User-control over representations of tagged unions, unions and structures.
Benefits: Precise control over bits.

Part I stands alone. Parts II, III depend on Part I, but are independent of each other.

These constructs significantly raise the level of programming with structures and unions, eliminate a number of common errors, and improve readability. The underlying ideas have had an enthusiastic following in many high-level languages for many years, in no small part because they have clean formal semantics. They are fully synthesizable (> 3 years experience with synthesis), and so are of interest both to designers and to verification engineers.

Part I: Proposal for Tagged Unions

Change the syntax box at the top of "Section 3.11 Structures and Unions" by prefixing both the **union** *keywords with the optional keyword* **tagged**

```
data_type ::=                                                    // from Annex A.2.2.1
    ...
  | [ tagged ] union packed [ signing ] { { struct_union_member } } { packed_dimension }
    ...
  | [ tagged ] union [ signing ] { { struct_union_member } }
```

At the end of Section 3.11, add the following text.

The keyword **tagged** preceding a union declares it as a tagged union, thereby making it type-safe. An ordinary union can be updated using a value of one member type and read as a value of another member type, which is a type loophole. A tagged union stores both the member value and a *tag*, i.e., additional bits representing the current member name. The tag and value can only be updated together consistently, using a statically type-checked tagged union expression (Section 7.14+). The member value can only be read with a type that is consistent with the current tag value (i.e., member name). Thus, it is impossible to store a value of one type and (mis)interpret the bits as another type.

In addition to type safety, the use of member names as tags also makes code simpler and smaller than code that has to manage tags explicitly. Tagged unions can also be used with pattern matching (Section 8.4), which further dramatically improve readability.

Example: an integer together with a valid bit:

```
typedef tagged union {
    void  Invalid;
    int   Valid;
} VInt;
```

A value of `VInt` type is either Invalid and contains nothing, or is Valid and contains an int. Section 7.14+ describes how to construct values of this type, and also describes how it is impossible to read an integer out of a `VInt` value that currently has the `Invalid` tag.

Example:

```
typedef tagged union {
    struct {
        bit [4:0] reg1, reg2, regd;
    } Add;
    tagged union {
        bit [9:0] JmpU;
        struct {
             bit [1:0] cc,
```

```
              bit [9:0] addr;
          } JmpC;
      } Jmp;
  } Instr;
```

A value of `Instr` type is either an Add instruction, in which case it contains three 5-bit register fields, or it is a Jump instruction. In the latter case, it is either an unconditional jump, in which case it contains a 10-bit destination address, or it is a conditional jump, in which case it contains a 2-bit condition-code register field and a 10-bit destination address. Section 7.14+ describes how to construct values of `Instr` type, and describes how, in order to read the `cc` field, for example, the instruction must have opcode `Jmp` and sub-opcode `JmpC`.

When the **packed** qualifier is used on a tagged union, all the members must have packed types, but they do not have to be of the same size. The (standard) representation for a packed tagged union is the following.

- The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the members.
- The size of the tag is the minimum number of bits needed to code for all the member names (e.g., 5 to 8 members would need 3 tag bits).
- The tag bits are always left-justified (*i.e.,* towards the most-significant bits).
- For each member, the member bits are always right-justified (*i.e.,* towards the least-significant bits).
- The bits between the tag bits and the member bits are undefined (shall contain '`x`'). In the extreme case of a member of `void` type, only the tag is significant and all the remaining bits are undefined.

The representation scheme is applied recursively to any nested tagged unions.

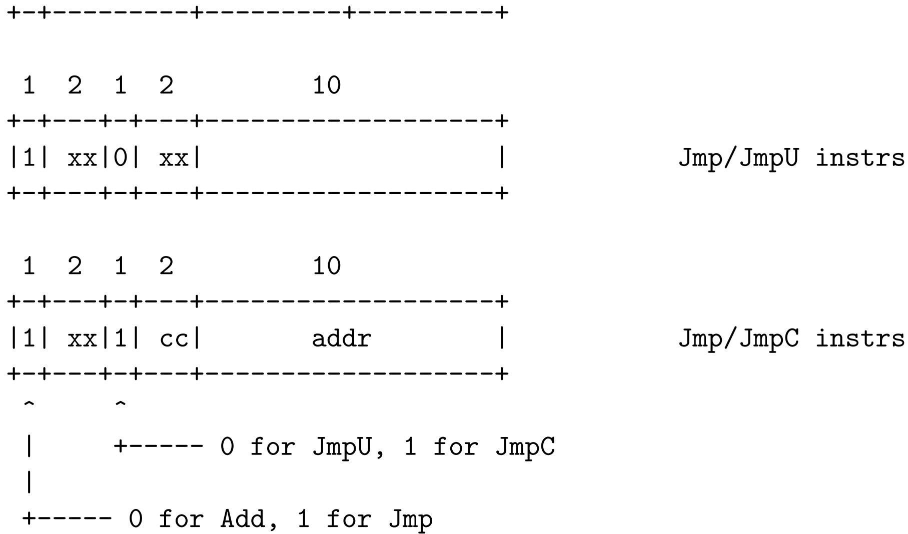
Example: If the `VInt` type definition had the **packed** qualifier, Invalid and Valid values will have the following layouts, respectively:

```
 1                              32
+-+-----------------------------------------------------------------+
|0| x x x x x x x ...          ...         ...   x x x x x x x x x |
+-+-----------------------------------------------------------------+

 1                              32
+-+-----------------------------------------------------------------+
|1|                    ... an int value ...                         |
+-+-----------------------------------------------------------------+
 ^
 +----- 0 for Invalid, 1 for Valid
```

Example: If the `Instr` type had the **packed** qualifier, its values will have the following layouts:

```
 1     5         5         5
+-+---------+---------+---------+
|0|  reg1   |  reg2   |  regd   |           Add instrs
```



```
+-+---------+---------+---------+

 1  2  1  2        10
+-+---+-+---+------------------+
|1| xx|0| xx|                  |          Jmp/JmpU instrs
+-+---+-+---+------------------+

 1  2  1  2        10
+-+---+-+---+------------------+
|1| xx|1| cc|       addr       |          Jmp/JmpC instrs
+-+---+-+---+------------------+
 ^     ^
 |     +----- 0 for JmpU, 1 for JmpC
 |
 +----- 0 for Add, 1 for Jmp
```

Add the following new sub-section just after "Section 7.14 Structure expressions":

Section 7.14+ Tagged union expressions and member access

expression ::= *from Annex A.8.3*
    ...
  | tagged_union_expression

tagged_union_expression ::=
    tagged identifier expression
  | **tagged** identifier ()
  | **tagged** identifier

A tagged union expression (packed or unpacked) is expressed using the keyword **tagged** followed by a tag identifier, followed by an expression representing the corresponding member value. If the member type is `void`, the member value expression can be `()` or omitted altogether.

The following examples are based on the tagged union type definitions in Section 3.11. The expressions in braces are structure expressions (Section 7.14).

```
tagged Valid   (23+34)                    // Create Valid int
tagged Invalid ()                         // Create an Invalid value
tagged Invalid                            // Create an Invalid value

// Create an Add instruction with its 3 register fields
tagged Add { e1, 4, ed }                  // struct members by position
tagged Add { reg2:e2, regd:3, reg1:19 }   // by name (order does not matter)

// Create a Jump instruction, with ``unconditional'' sub-opcode
tagged Jmp (tagged JmpU 239)

// Create a Jump instruction, with ``conditional'' sub-opcode
```

```
tagged Jmp (tagged JmpC { 2, 83 })          // inner struct by position
tagged Jmp (tagged JmpC { cc:2, addr:83 })  // by name
```

The type of a tagged union expression must be known from its context (e.g., it is used in the RHS of an assignment to a variable whose type is known, or it is has a cast, or it is used inside another expression from which its type is known). The expression evaluates to a tagged union value of that type. The tagged union expression can be completely type-checked statically: the only member names allowed after the **tagged** keyword are the member names for the expression type, and the member expression must have the corresponding member type.

An uninitialized variable of tagged union type shall contain `x`'s in all the bits, including the tag bits. A variable of tagged union type can be initialized with a tagged union expression provided the member value expression is a legal initializer for the member type.

Members of tagged unions can be read or assigned using the usual dot-notation. Such accesses are completely type-checked, i.e., the value read or assigned must be consistent with the current tag. In general, this is a runtime check. An attempt to read or assign a value whose type is inconsistent with the tag results in a runtime error.

All the following examples are legal only if the instruction variable `instr` currently has tag A:

```
x              = instr.Add.reg1;
instr.Add      = {19, 4, 3};
instr.Add.reg2 = 4;
```

In "Annex A Formal Syntax" make all the BNF changes described in this part of the proposal.

In "Annex B Keywords", add the keyword **tagged**.

Part II: Proposal for Pattern Matching in Case Statements, If Statements and Conditional Expressions

Change the syntax box at the start of "Section 7.16 Conditional operator":

```
conditional_expression ::=                                                from Annex A.8.3
    expressions_and_cond_patterns ? { attribute_instance } expression2 : expression3

expressions_and_cond_patterns ::=                                         from Annex A.6.6
    expression_or_cond_pattern { , expression_or_cond_pattern }

expression_or_cond_pattern ::=
    expression | cond_pattern

cond_pattern ::= expression matches pattern
```

Insert the following after the syntax box and before the first sentence in "Section 7.16 Conditional operator":

> This section describes the traditional notation where expression_and_cond_patterns is just a single expression. SystemVerilog also allows expression_and_cond_patterns to perform pattern matching, and this is described in Section 8.4.

In the syntax box at the top of "Section 8.4 Selection Statements", change predicates in if-statements to allow cond_patterns, add productions for cond_patterns, and add a new family of clauses to the case_statement production for pattern matching case statements:

```
conditional_statement ::=                                        from Annex A.6.6
     if ( expressions_and_cond_patterns ) [ statement_or_null ]
   | unique_priority_if_statement

unique_priority_if_statement ::=
     [ unique_priority ] if ( expressions_and_cond_patterns ) statement_or_null
     { else if ( expressions_and_cond_patterns ) statement_or_null }
     [ else statement_or_null ]

unique_priority ::= unique | priority

expressions_and_cond_patterns ::=
     expression_or_cond_pattern { , expression_or_cond_pattern }

expression_or_cond_pattern ::=
     expression | cond_pattern

cond_pattern ::= expression matches pattern

case_statement ::=                                            // from Annex A.6.7
     [ unique_priority ] case ( expression ) case_item { case_item } endcase
   | [ unique_priority ] casez ( expression ) case_item { case_item } endcase
   | [ unique_priority ] casex ( expression ) case_item { case_item } endcase
   | [ unique_priority ] casep ( expression ) casep_item { casep_item } endcase
   | [ unique_priority ] casepx ( expression ) casep_item { casep_item } endcase
   | [ unique_priority ] casepz ( expression ) casep_item { casep_item } endcase

case_item ::=
     expression { , expression } : statement_or_null
   | default [ : ] statement_or_null

casep_item ::=
     pattern : statement_or_null
   | default [ : ] statement_or_null
```

At the end of "Section 8.4 Selection Statements", add the following sub-section:

Section 8.4.1 Pattern matching

Pattern matching provides a visual and succinct notation to compare a value against structures, tagged unions and constants, and to access their members. SystemVerilog adds pattern matching capability to **case** and **if** statements, and to conditional expressions. Before describing pattern matching in those contexts, we first describe the general concepts.

A pattern is a nesting of tagged union and structure expressions with identifiers, constant identifiers, integer constants and string literals at the leaves. For tagged union patterns, the identifier represents the member name. If the member type is `void`, the nested member pattern must be `()` or omitted altogether.

```
pattern ::=                                                   // from Annex A.6.7.1
      identifier
    | number
    | string_literal
    | tagged identifier pattern
    | tagged identifier ()
    | tagged identifier
    | { pattern , ... , pattern }
    | { identifier : pattern , ... , identifier : pattern }
```

A pattern always occurs in a context of known type. Hence its nested patterns also have known type.

An identifier in a pattern may be a named constant that is currently in scope (such as a parameter or an identifier defined in an enum). Otherwise, the identifier is implicitly declared as a fresh variable with the unique type that it must have based on its position in the pattern. These are known as *pattern identifiers*, and they must be unique in the pattern, i.e., the same identifier cannot be used in two different positions in a single pattern.

We always match (for equality) the value V of an expression against a pattern. Note that static type-checking ensures that V and the pattern have the same type. The result of a pattern match is:

- A 1-bit determined value: 0 (false, or fail) or 1 (true, or succeed). The result cannot be `x` or `z` even if the value and pattern contain such bits.
- If the match succeeds, the pattern identifiers are bound to the corresponding members from V, using ordinary procedural assignment. The scope of these identifiers is described below separately for **casep** and **if** statements and conditional expressions. Since pattern identifiers are fresh variables, subsequent assignments to these identifiers have no effect on any other variable. In particular, they cannot be used to update any structure or tagged union.

Each pattern is matched using the following simple recursive rule:

- An identifier pattern always succeeds (matches any value), and the identifier is bound to that value (using ordinary procedural assignment).
- An integer constant, string literal, or constant identifier pattern succeeds if V is equal to that value.
- A tagged union pattern succeeds if the value has the same tag, and if the nested pattern matches the member value of the tagged union.
- A structure pattern succeeds if each of the nested member patterns matches the corresponding member values in V. In structure patterns with named members, the textual order of members does not matter, and members may be omitted. Omitted members are ignored.

Conceptually, if the value V is seen as a flattened vector of bits, the pattern specifies which bits to match, with what values they should be matched and, if the match is successful, which bits to extract and bind to the pattern identifiers. Matching can be insenstitive to `x` and `z` values, as described in the individual constructs below.

Section 8.4.1.1 Pattern matching in case statements

A **casep** statement consists of a series of "casep_items" whose left-hand sides are patterns and right-hand sides are statements. All the patterns are completely statically type-checked. The expression being tested in the **casep** statement must have the same type as the pattern in each case item. If the expression and pattern are tagged unions or structures, the corresponding members are recursively type-checked. The types of all pattern identifiers are therefore unique and self-evident..

The value V of the expression being tested in a **casep** statement is matched against the patterns, one at a time, in the order they are given. If a pattern is successfully matched, its case item is selected. Any pattern identifiers are bound to the corresponding member values in V. Finally, the statement on the right-hand side of the case item is executed. The scope of the pattern identifiers in each case item is just the statement in the right-hand side of that case item. Thus different case items can reuse pattern identifiers.

If none of the patterns match the value, no statement is executed. The **default** pattern always matches any value, and can be used no more than once in a catch-all case item.

The following examples are based on the tagged union declaration examples in Section 3.11.

Example:

```
casep (v)
  tagged Invalid  : $display ("v is invalid");
  tagged Valid  n : $display ("v is Valid with value %d", n);
endcase
```

If `v` currently has the Invalid tag, the first pattern is matched. Otherwise, it must have the Valid tag, and the second pattern is matched. The identifier `n` is bound to the value of the Valid member, and this value is displayed. It is impossible to access the integer member value (`n`) when the tag is Invalid.

Example:

```
Instr instr;
...
casep (instr)
  tagged Add {r1,r2,rd} : rf[rd] = rf[r1] + rf[r2];
  tagged Jmp j          : casep (j)
                            tagged JmpU  a   : pc = pc + a;
                            tagged JmpC {c,a}: if (rf[c]) pc = a;
                          endcase
endcase
```

If `instr` holds an Add instruction, the first pattern is matched, and the identifiers `r1`, `r2` and `rd` are bound to the three register fields in the nested structure value. The right-hand side statement executes the instruction on the register file `rf`. It is impossible to access these register fields if the tag is Jmp.

If `instr` holds a Jmp instruction, the second pattern is matched, and the identifier `j` is bound to the nested tagged union value. The inner **casep** statement, in turn, matches this value against JmpU and JmpC patterns, and so on.

Example (same as previous example, but using nested patterns):

```
casep (instr)
  tagged Add {r1,r2,rd}         : rf[rd] = rf[r1] + rf[r2];
  tagged Jmp (tagged JmpU a)    : pc = pc + a;
  tagged Jmp (tagged JmpC {c,a}): if (rf[c]) pc = a;
endcase
```

Example (same as previous example, with named structure components):

```
casep (instr)
  tagged Add {reg2:r2,regd:rd,reg1:r1}   : rf[rd] = rf[r1] + rf[r2];
  tagged Jmp (tagged JmpU a)             : pc = pc + a;
  tagged Jmp (tagged JmpC {addr:a,cc:c}) : if (rf[c]) pc = a;
endcase
```

casep, **casepz** and **casepx** are analogous to **case**, **casez** and **casex**, respectively. In other words, during pattern matching, wherever two bits are compared (whether they are tag bits or members), the **casez** form ignores `z` bits, and the **casex** form ignores both `z` and `x` bits.

The **priority** and **unique** qualifiers play their usual role. **priority** implies that some pattern must match. **unique** also implies that exactly one pattern will match, so that the patterns may be matched in parallel.

Section 8.4.1.2 Pattern matching in if statements

The predicate in an **if** statement can be a comma-separated series of clauses, where each clause is either an expression (used as a boolean), or has the form *expression* **matches** *pattern*. The clauses represent a sequential conjunction from left to right, i.e., if any clause fails the remaining clauses are not evaluated, and all of them must succeed for the predicate to be true. Boolean expression clauses are evaluated as usual.

In each e **matches** p clause, the value e is matched against the pattern p as described above. The pattern identifiers in each clause are implicitly declared with the unique type that they must have based on their position in the pattern. The scope of a pattern identifier consists of the remaining clauses in the predicate, and the "true" arm of the **if** statement.

Even though pattern matching clauses always return a defined 1-bit result, they may be conjoined with ordinary expressions in the predicate, and hence the overall overall result may be ambiguous. The standard semantics of **if** statements remain unchanged, i.e., the first statement is executed if and only if the result is a determined value other than 0.

The following examples are based on the tagged union declaration examples in Section 3.11.

Example:

```
if (e matches (tagged Jmp (tagged JmpC {cc:c,addr:a})))
     ...     // c and a can be used here
else
     ...
```

Example (same as previous example, illustrating a sequence of two pattern-matches with identifiers from the first pattern used in the second pattern).

```
if (e matches (tagged Jmp j),
    j matches (tagged JmpC {cc:c,addr:a}))
     ...     // c and a can be used here
else
     ...
```

Example (same as first example, but adding a boolean expression to the sequence of clauses). The idea expressed is: "if e is a conditional jump instruction and the condition register is not equal to zero ...".

```
if (e matches (tagged Jmp (tagged JmpC {cc:c,addr:a})),
    rf[c] != 0)
     ...     // c and a can be used here
else
     ...
```

The **priority** and **unique** qualifiers play their usual role for **if** statements even if they use pattern matching.

Section 8.4.1.3 Pattern matching in conditional expressions

A conditional expression *(e1 ? e2 : e3)* can also use pattern matching, i.e., the predicate $e1$ can be a sequence of expressions and "expression **matches** pattern" clauses, just like the predicate in an **if** statement. As described in the prevous section, $e1$ may evaluate to true, false or an ambiguous value. The semantics of the overall conditional expression are unchanged from the description in Section 7.16, based on these three possible outcomes for $e1$. The scope of the pattern identifiers introduced in each "expression **matches** pattern" clause in $e1$ consists of the rest of the clauses in $e1$ and the consequent expression $e2$.

In "Annex A Formal Syntax" make all the BNF changes described in this part of the proposal.

In "Annex B Keywords" add the keywords **casep**, **casepz**, **casepx** *and* **matches**.

Part III: User-defined Representations for Packed Structures, Unions and Tagged Unions

Add the following section at the end of "Section 3.11 Structures and Unions"

SystemVerilog offers the designer precise control over the binary representation of a packed structure, union or tagged union type that has been typedef'd to a type identifier T. The representation can be customized simply by defining the following methods in the same scope as T's declaration:

```
function bit [n:0] pack   (T x);
function T         unpack (bit [n:0] bits);
```

These functions can perform arbitrary packing and unpacking of values of type T to and from bit vectors of size n (the designer chooses n). Such non-standard representations may be necessary to meet external representation requirements. They can also produce more efficient representations than the standard one (e.g., Huffman encoding).

Example: the example `VInt` type in this section is normally represented with 33 bits: 1 bit at the MSB end for the Invalid (0) and Valid (1) tags, followed by 32 bits for the `int` value in Valid members. By defining `pack` and `unpack` methods in the same scope as the `VInt` declaration, the designer can choose an alternate representation, such as this one which has the valid bit at the LSB and uses 1 for Invalid and 0 for Valid (this may be a requirement of some external interface or external IP):

```
function bit [32:0] pack (VInt x);
    case (x)
        Invalid : pack = 1;
        Valid x : pack = {x,1'b0};
    endcase
endfunction

function VInt unpack (bit [32:0] xb);
    if (xb == 1) unpack = tagged Invalid;
    else         unpack = tagged Valid xb[32:1];
endfunction
```

No other code needs to be changed (type declarations, member access and assignment, pattern matching). This mechanism is simply a hook into the SystemVerilog implementation to allow customized representations.

Checklist of issues raised in the Nov 10 meeting in San Jose

The following is a checklist of issues raised the first time the "Tagged Unions and Pattern Matching" proposal was presented before the SV-BC committee, on Nov 10, 2003 at San Jose (for which, many thanks!). For each issue, I have stated the issue, and described briefly how it has been addressed.

Implicitly declared identifiers and their scope

The issue: The implicit declaration of pattern identifiers does not follow Verilog tradition, which does not implicitly declare identifiers.

Response: Yes, but it is worth it, for the following reasons:

- The trend in modern languages is to allow this, for simplicity, brevity and readability, provided the type and scope of the identifier is unambiguous and simply defined.
- In the pattern matching proposal, the type of the identifier is obvious and unique, based on its position in the pattern, and the scope is simple and very local: the right-hand side of a case item; the true-consequent in an if-statement; the true-consequent in a conditional expression.
- The **foreach** proposal (Section 12.4.7) also introduces implicitly declared identifiers.

Distinguishing Pattern Identifiers from Constants

The issue: for an identifier in a pattern, how do we know whether it represents a constant currently in scope (such an enum identifier or a parameter) to be checked for equality with the tested value, or whether it represents a new, implicitly declared pattern identifier to be bound to the value?

Response: We have defined it so that the former always takes precedence over the latter. If there is a constant identifier in scope, then that is what the identifier in the pattern represents. The identifier represents a new implicitly declared pattern identifier only if there is no such constant identifier in scope. These semantics have been implemented and used widely in several languages with pattern matching, for many years.

Can Pattern Identifiers be used for update?

The issue: What exactly are pattern identifiers bound to? A copy of the *value* of the corresponding member? Can these identifiers be assigned? If so, does the original member change? I.e., is the identifier bound to an l-value or an r-value representing the member?

Response The text has been clarified to explain that pattern matching is only used to access member values, not for assignment. Pattern identifiers represent fresh variables, and on a successful pattern-match they receive a copy of the corresponding member values using ordinary procedural assignment. Yes, these identifiers can be assigned, but that has no effect on any other structure or tagged union. The reason for these semantics is that pattern matching can test an arbitrary expression of the appropriate type, and so it does not make sense to think of the pattern identifiers as updatable l-values. To update an original structure or tagged union, one uses the standard update notations. Also, experience shows that one rarely updates individual members; if the structure or tagged union is ever updated, one usually assigns it as a whole item.

Ambiguity between pattern matching and normal case semantics

The issue: Since the left-hand side of a case item can be an arbitary runtime expression, and since patterns are a subset of expressions, how do we know if a case statement has normal semantics or pattern matching semantics?

Response: Correct: it would be impossible to disambiguate. We have introduced a new kind of case statement using the keyword **casep** which is used exclusively for pattern matching.

Behavior of x and z values in pattern matching

The issue: What are the semantics `x` and `z` values in **case**, **if** and conditional expressions, in the presence of pattern matching?

Response: These have been precisely defined in this proposal. There are no surprises: the behavior follows the normal behavior of these constructs without pattern matching.

Behavior of unique and priority keywords in pattern matching case and if statements

The issue: What are the semantics of the **unique** and **priority** qualifiers for **if** and **case**, in the presence of pattern matching?

Response: These have been precisely defined in this proposal. There are no surprises: the behavior follows the normal behavior of these constructs without pattern matching.

Semantics of pattern matching in if stmts (are pattern-matches really Boolean expressions)?

The issue: In the original proposal, "e1 **matches** pattern" was described as an ordinary Boolean expression. That raises tricky issues such as this: suppose we had:

if ((e1 matches p1) || (e2 matches p2)) ...

then what if p1 fails and p2 succeeds? What do the pattern variables in p1 (failed) represent? Can pattern variables in p1 (p2) be used in e2 (e1)? And similar questions arise in other Boolean contexts.

Response: "e1 **matches** pattern" is no longer an ordinary expression. The BNF has been fixed so that it can only be used at the top-level of the predicate in an if-statement predicate or conditional expression. The semantics of the phrase, and scope of pattern variables has been defined precisely and unambiguosly. The BNF is a proper extension, i.e., the existing BNF is a proper subset.

Can pattern matching be used in conditional expressions "(e1?e2:e3)"?

The issue: The original proposal only talked about pattern matching in **if** statements. What about conditional expressions?

Response: The proposal has been extended to allow pattern matching in conditional expressions as well. The new BNF is a proper extension, i.e., the existing BNF is a proper subset.