Classified Advertisements: Official Specification Version α*

Rajesh Raman, Miron Livny, James Pruyne and Weiru Cai

University of Wisconsin-Madison
Computer Science Department
1210 West Dayton Street
Madison, WI 53706

February 17, 1997

"God send you joy, Petruchio! 'tis a match."
–Baptista, The Taming of The Shrew, II(i)

This document specifies the logical structure and semantics of the "classified advertisement" (i.e., classad) mechanism. Every classad implementation is required to comply with the described structure and semantics.

This document also includes lexical and syntactic conventions currently employed. While implementors are strongly encouraged to comply with these conventions, they are not required to do so if these conventions make classad expression unnatural or contrived in the paradigm of the implementation environment. Such conventions include the use of infix notation for expressions, the use of certain reserved lexical tokens for operators, etc.

Finally, details of our pilot C++ implementation are also occasionally included to provide intuition and implementation hints.

Unless explicitly specified, given descriptions are to be treated as required features rather than desirable conventions or implementation hints.

1 Introduction

The classified advertisement mechanism is a flexible and expressive framework for match-making between entities. The classad mechanism does not constrain which entities take part in the match-making process, how they wish to describe themselves, or what it takes to satisfy their requirements. Instead, the classad framework rigorously defines a mechanism which arbitrary entities can use to describe themselves and their requirements. This document is the official specification of the classad mechanism.

The document is structured as follows: section 2 is an overview of the classad mechanism, section 3 defines classad types, section 4 defines the structure and semantics of expressions used in the classad system and section 5 defines the structure of classads and the operations defined on them.

2 Overview

This section gives a high-level overview of the classad mechanism to provide intuition for developers. An entity that requires match-making services describes itself in a classad, which is an aggregate of the following information:

*This document is the first draft of the specification. As such, it is likely to change and must be treated as work in progress. Kindly restrict distribution till Version 1 is released.

Type information This is a tuple of two types, self type and match type, which characterizes the kinds of entities involved in the match.

Attribute List This is a list of attributes which describes the properties of the advertising entity.

Requirement This is an expression which represents the constraints that the advertising entity places on the match ad.

Rank This is an expression which represents the advertising entity's assessment of the quality of a match.

Advertisement ID If an entity publishes several ads and a match is found, the match can be qualified with an advertisement ID of the matched ad so that the advertising entity can identify which ad was matched.

The following table shows two example classads.

	Classad Example 1	Classad Example 2
Self type	Machine	Job
Match type	Job	Machine
Attribute List	OpSys = "OSF/1" Arch = "Alpha" Memory = 32 Disk = 782 KbdIdle = 17 LoadAvg = 0.1	Owner = "raman" Executable = "a.out" ImageSize = 1000 State = "Idle" RemoteCPU = 0
Requirement	(LoadAvg < 0.3) && (KbdIdle > 15)	(OpSys == "OSF/1") && (Arch == "Alpha")
Rank	((TARGET.Owner == "raman") * 50)	(TARGET.Memory * 100) + TARGET.Disk
Ad ID	0	0

Since requirement and rank expressions need to reference variables which may not be in the same classad, facilities are provided whereby a classad can explicitly name a variable not in its scope. In particular, a variable reference may be decorated with a scope resolution prefix which forces variable lookup in the hosting ad's attribute list, the candidate ad's attribute list or a common global scope. These are the MY, TARGET and ENVIRONMENT scopes respectively.

Two classads A and B are said to *match* if and only if:

1. A's self type is equal to B's match type and B's self type is equal to A's match type.

2. A's requirement evaluates to `TRUE` with respect to B, and B's requirement evaluates to `TRUE` with respect to A.

In particular, note that the two classads in the example match. One may also conceive of a one-way match in which A matches B's requirement, but the converse is not true.

The rest of this document formalizes the notion of a classad, and defines the concepts that constitute the classad mechanism.

3 Classad types

Every classad is primarily characterized by exactly two type values: the self type and the match type. The specific values which compose the entire type space of classads is unspecified. However, the following constraints must be observed.

1. All classads must share the same type space.

2. The type space must contain a distinguished value indicating the `NULL` type value.

3. Values in the type space must be comparable for equality and inequality.

4. Any value in the type space must be equal to itself (and only itself).

Implementation hint:

The current implementation of classads uses character strings as the typespace. Thus, "Machine" and "Job" are type values, as is `NULL`, which serves as the `NULL` type value.

4 Expressions

Expressions are constructs that yield values upon evaluation. These constructs are composed of variable references, literals and other expressions combined with unary and binary operators.

4.1 Types

Types in the classad expression system may be thought of as sets of values. (These expression types should not be confused with classad types as discussed in section 3.) The set of types $\mathcal{T}$ supported by the classad expression mechanism is

$$\mathcal{T} = \{S, I, R, B, U, E\}$$

where

- $S = \{$ set of strings $\}$
- $I = \{$ set of integers $\}$
- $R = \{$ set of real numbers $\}$
- $B = \{$ `FALSE`, `TRUE` $\}$
- $U = \{$ `UNDEFINED` $\}$
- $E = \{$ `ERROR` $\}$

The sets U and E are composed of exactly one element each, and B is composed of the boolean constants `FALSE` and `TRUE`.

The exact characterization of the sets S, I and R in terms of values, size and range is implementation dependent. However, there must exist the following injective "type cast" functions

- $C_{BI} : \{\texttt{FALSE} \rightarrow 0, \texttt{TRUE} \rightarrow 1\}$ to map boolean values to integers,
- $C_{BR} : \{\texttt{FALSE} \rightarrow 0.0, \texttt{TRUE} \rightarrow 1.0\}$ to map boolean values to reals,
- $C_{IB} : I \rightarrow B; C_{IB}(v) = \begin{cases} \texttt{FALSE} & \text{if } v = 0 \\ \texttt{TRUE} & \text{otherwise} \end{cases}$ to convert integers to boolean values, and
- $C_{IR} : I \rightarrow R$ to promote integers to reals in the natural way.

We also define the relation $\lhd$ on $\{B, I, R\}$ as

$$\lhd = \{(B, I), (B, R), (I, R)\}$$

This relation defines an order on the types B, I, and R, and is useful when defining the semantics of operators. Intuitively, given $t_1, t_2 \in \{B, I, R\}$, then $t_1 \lhd t_2$ implies that t_1 is a "lesser type" than t_2, and that values in t_1 can be promoted to values in t_2.

Implementation hint:

The sets S, I and R naturally correspond to the character string, signed integer and floating point types expressible in the implementation environment. In our implementation, these are the `char *`, `int` and `float` types respectively.

4.2 Syntax

This document does not define the explicit syntax of expressions so that the classad mechanism may be implemented in various languages and paradigms without arbitrarily constraining expressibility. Instead, this document defines the structure of expressions (section 4.3), and places certain constraints on the implementation's syntax of the classad expression language. The single most important constraint on syntax is that there be no ambiguity in identifying variable references, operators, or the types of expressed literal values in the expression.

4.2.1 Literals

All values of *all* types are required to be syntactically expressible as literals. This includes the values `UNDEFINED` and `ERROR`.

Convention:

Additional syntax is usually necessary to distinguish string literals from variable references. These "string delimiter" tokens are usually quote characters. In addition, implementations normally include special escape sequences to denote characters that may not otherwise be expressed in the string, such as the delimiter itself. Both practices are encouraged.

In our implementation, the delimiter is the double quote (`"`) character, and the escape character is the backslash (\) character.

4.2.2 Operator expressibility

All operators are required to be syntactically expressible.

Convention:

The convention followed is to express all operators in the syntax of the C language. Therefore, the expressed operators are:

```
!  (logical not)    -  (unary minus)   && (logical and)    || (logical or)
                 +  -  *  /  %      (arithmetic operators)
                 <  <= >  >= == !=  (comparison operators)
```

4.2.3 Variables

The syntax of identifiers used in naming variables is left undefined. However, the syntax of variable references must allow the following scope resolution prefixes to be prepended to identifiers:

1. MY scope prefix,
2. TARGET scope prefix, and
3. ENVIRONMENT scope prefix.

The semantics of decorating a variable reference with a scope resolution prefix is explained in section 4.5.3.

Convention:

The convention followed is that identifiers must match the regular expression: `[A-Za-z_][0-9A-Za-z_]*`

The scope resolution prefixes are `MY.`, `TARGET.` and `ENV.`, so a variable reference to the `LocalTime` variable in the environment scope would be: `ENV.LocalTime`.

4.2.4 Parentheses

Parentheses are not required to be a part of the syntax if the syntax of expressions is powerful enough to express any expression without ambiguity. For example, parentheses are not required in prefix notation. However, if infix notation is adopted, parentheses *must* be provided to unambiguously specify grouping of subexpressions.

Convention:

Current implementations of classads use infix notation and parentheses.

4.3 Structure

The definition of the structure of an expression is as follows:

1. A literal is an expression.
2. A variable reference is an expression.
3. A unary operator applied to an expression is an expression.
4. A binary operator applied to two expressions is an expression.
5. An expression wrapped in parentheses is an expression. (If required.)

4.4 Operators

The set of operators $\mathcal{O}$ supported in classad expressions is

$$\mathcal{O} = \mathcal{O}_{\mathcal{U}} \cup \mathcal{O}_{\mathcal{B}}$$

where $\mathcal{O}_{\mathcal{U}}$, the set of unary operators is

$$\mathcal{O}_{\mathcal{U}} = \{\neg, \prime\}$$

which are the logical negation and unary minus operators, and $\mathcal{O}_{\mathcal{B}}$, the set of binary operators is

$$\mathcal{O}_{\mathcal{B}} = \{+, -, \div, \times, \%, \wedge, \vee, =, \neq, <, \leq, >, \geq\}$$

which are the addition, subtraction, division, multiplication, modulus, logical and, logical or, equal, not equal, less than, less than or equal, greater than and greater than or equal operators respectively.

It is important to note that there is no assignment operator in the set $\mathcal{O}$. Therefore, expressions may not contain assignment operators. Expressions may however be bound to variables to form *attribute.* Only attributes may be inserted into a classad's attribute list.[1] This concept will be more rigorously defined in section 5.

4.4.1 Precedence and order of evaluation

The order of evaluation rules for the operators are as follows:

1. Unary operators have higher precedence than binary operators, i.e., operators of the set $\mathcal{O}_{\mathcal{U}}$ have higher precedence than operators of the set $\mathcal{O}_{\mathcal{B}}$.
2. The relative precedence of the binary operators is described by the following table. Operators on the same row have the same precedence, and the rows are given in decreasing order of precedence.

[1] Though a mild abuse of terminology, one may speak of the expressions in the attribute list instead of the attributes in the attribute list. The intended meaning is usually clear from context.

Operators	Associativity
$\times$, $\div$, %	left to right
$+$, $-$	left to right
$<$, $\leq$, $>$, $\geq$	left to right
$=$, $\neq$	left to right
$\wedge$	left to right
$\vee$	left to right

3. Operators of the same precedence are evaluated in left to right order.

4.4.2 Operator semantics

Operators primarily operate on values of types B, I, R, and S. However, they also operate on the `UNDEFINED` and `ERROR` values of types U and E respectively. Thus, operators may be considered to be polymorphic functions.

The semantics of operators is defined to be total, so that the action of operators in any given situation is defined. Thus, there is the notion that any given operator has primary and secondary domains. The primary domain represents the set of values on which the operator naturally operates. Additionally, the secondary domain respresents the values on which the usual actions of the operator are meaningless, e.g., `''Joe'' && ''Alice''`. These situations result in the generation of the `ERROR` value as the result of the operation.

In general, the following semantics are used:

- Comparison primarily operates on booleans, integers, strings, and reals. Comparing `UNDEFINED` values results in the `UNDEFINED` value.
- Arithmetic operators operate primarily on booleans, integers and reals. If necessary, type conversions are performed to convert both operands to the same type. Arithmetic on the `UNDEFINED` value results in the `UNDEFINED` value.
- Logical operators primarily operate on booleans and the `UNDEFINED` value. If `UNDEFINED` values in the expression have no contribution to the truth value of the expression, they are ignored.
- In general (but not always), the action of an operator on values from its secondary domain results in the generation of the `ERROR` value.
- The `ERROR` value always propogates through any evaluation.

The detailed semantics of operators is defined below.

- Group 1: Comparison operators.

  $$\mathcal{O}_c = \{<, \leq, =, \neq, >, \geq\}$$

  For any operator $\asymp \in \mathcal{O}_c$ and values v_1 and v_2 of types t_1 and t_2 respectively, the semantics of comparison are described by the following rules, which are to be applied in order.

  1. If $t_1, t_2 \in \{B, I, R\}$ and $t_1 \triangleleft t_2$, the result of the comparison is $C_{t_1 t_2}(v_1) \asymp v_2$. Namely, the value of the "lesser" type is to be promoted to the corresponding value of the "greater" type after which comparison proceeds as defined in rule 3.
  2. Similar to rule 1, if $t_2 \triangleleft t_1$, the result of the comparison is $v_1 \asymp C_{t_2 t_1}(v_2)$.
  3. If $t_1 = t_2$ and $t_1 \in \{B, I, R, S\}$, $v_1 \asymp v_2$ is defined by the ordering of values inherent in the set t_1. The boolean value `FALSE` is to be considered less than the value `TRUE`.

     Convention:

     Ideally, the string comparison functionality should be equivalent to `strcmp` of the standard C library with respect to ordering strings.
  4. If one of v_1 and v_2 (or both) is `UNDEFINED`, the result of the comparison is `UNDEFINED`.

5. If none of the above rules are applicable, the result of the comparison is **ERROR**, the single value of type E.

- Group 2: Arithmetic operators.

$$\mathcal{O}_a = \{\times, \div, \%, +, -\}$$

The semantics of arithmetic operators is similar to that of the comparison, with the following notable changes:

1. Arithmetic cannot be performed on strings.
2. Arithmetic involving boolean values is explicitly defined.
3. The result of division by zero is defined.
4. The behavior of the unary minus operator $\prime$ (not included in the set $\mathcal{O}_a$) is also defined. Namely, the result of the expression $\prime v$ where v is of type $t \in \{B, I, R\}$ is $(-1 \times v)$ as defined below.

For any operator $\odot \in \mathcal{O}_a$ and values v_1 and v_2 of types t_1 and t_2 respectively, the semantics of arithmetic are described by the following rules in order.

1. If $t_1, t_2 \in \{B, I, R\}$ and $t_1 \lhd t_2$, the result of the computation is $C_{t_1 t_2}(v_1) \odot v_2$. Namely, the value of the "lesser" type is to be promoted to the corresponding value in "greater" type after which arithmetic proceeds as defined in rule 3.
2. Similar to rule 1, if $t_2 \lhd t_1$, the result of the computation is $v_1 \odot C_{t_2 t_1}(v_2)$.
3. The case of arithmetic on values of the same type (i.e., $t_1 = t_2$), is decomposed into two cases.
   (a) (Booleans) If $t_1 = t_2 = B$, the result of the computation is $C_{BI}(v_1) \odot C_{BI}(v_2)$. Namely, the boolean values should be converted to integers before the arithmetic is performed.
   (b) (Non-booleans) If $t_1 = t_2$ and $t_1 \in \{I, R\}$, $v_1 \odot v_2$ is computed in the arithmetic of that particular type. However, the result of the computation is **UNDEFINED** if one of the following statements is true:
      i. $\odot = \div$ and v_2 is zero.
      ii. $\odot = \%$ and $t_1 = R$.
      iii. $\odot = \%$ and v_2 is zero.
4. If either v_1 or v_2 (or both) is equal to **UNDEFINED**, the result of the computation is **UNDEFINED**.
5. If none of the above rules are applicable, the result of the comparison is **ERROR**, the single value of type E.

- Group 3: Logical operators.

$$\mathcal{O}_l = \{\wedge, \vee, \neg\}$$

The significant difference between the logical operators and the comparison and arithmetic operators is that logical operators, as a result of their "don't care" properties, can result in **TRUE** and **FALSE** outcomes even if some of their operands are **UNDEFINED**. The semantics of the logical operators is defined as follows:

1. If one or more operands of a logical operator are of type I, they are to be converted to boolean values using the function C_{IB}, after which the expression is evaluated with the converted values.
2. The truth tables of the logical operators are given below. The values **TRUE**, **FALSE** and **UNDEFINED** are abbreviated as T, F, and U respectively.

$\wedge$	T	F	U	$\vee$	T	F	U	$\neg$	
T	T	F	U	T	T	T	T	T	F
F	F	F	F	F	T	F	U	F	T
U	U	F	U	U	T	U	U	U	U

3. If none of the above rules are applicable, the result of the operation is **ERROR**.

4.5 Expression evaluation

This section defines the semantics of expression evaluation. An expression in a classad is normally evaluated only when the classad is being tested for a match with another classad. Thus, the evaluation of a given expression is usually triggered only if a requirement (or rank) expression directly (or indirectly) names a variable that the expression is bound to.

4.5.1 Expression evaluation semantics

The semantics of evaluating an expression E is as follows.

1. If E is a literal, the value of E is the value represented by the literal.
2. If E is a variable reference, the value of E is the value of the expression bound to the variable referred to. (Section 4.5.3 describes how this expression may be found.) If no such variable exists, the value of the expression is `UNDEFINED`.
3. If E is a unary operator applied to an expression, the value of E is defined by the operator semantics as defined in section 4.4.2.
4. If E is a binary operator applied to two expressions, the value of E is defined by the operator semantics as defined in section 4.4.2.
5. If E is an expression F wrapped in parentheses, the value of E is the value of F.

4.5.2 Scopes

Since there are always two classads involved in the process of expression evaluation, there are two attribute lists either of which can serve as the variable lookup environment for a given variable reference. Additionally, it is also meaningful to consider a global environment for resolving non-specific variables such as `Random`, `UniversalCoordinatedTime`, etc. which do not belong to either ad, but are useful in constructing expressions. Therefore, we define three scopes which can serve as environments for a variable reference:

1. The attribute list which hosts the expression which contains the variable reference (i.e., the MY scope).
2. The attribute list of the candidate classad which is being tested for a match (i.e., the TARGET scope).
3. The global environment (i.e., the ENVIRONMENT scope).

The contents of the MY and TARGET scopes during evaluation is determined by the entities which published the respective ads.

The variables available in the ENVIRONMENT scope is implementation dependent. These variables are different from variables in an attribute list in that expressions are not bound to them. Instead, references to variables in the environment scope result in the computation of a value as a result of some "action routine" that is called when the variable is referenced. For example, successive references to `ENV.Random` can result in the return of random values, references to `ENV.LocalTime` and `ENV.UniversalTime` can return the local and universal coordinated time respectively.

4.5.3 Variable reference

A variable reference is a special case of expression evaluation. Given two classads A and B, the semantics of a variable reference in an expression E of classad A is as follows:

1. If the reference is prefixed with a MY scope resolution, the variable is looked up in A's attribute list.
   - If the variable exists, the value of the reference is the value of the expression bound to the variable.
   - Otherwise, the value of the reference is `UNDEFINED`.
2. If the reference is prefixed with a TARGET scope resolution, the variable is looked up in B's attribute list.

- If the variable exists, the value of the reference is the value of the expression bound to the variable.
- Otherwise, the value of the reference is `UNDEFINED`.

3. If the reference is prefixed with a ENVIRONMENT scope resolution, a check is made to ensure that the variable is supported by the implementation.
   - If the reference is valid, the value computed as a result of the reference is returned.
   - Otherwise, the value of the reference is `UNDEFINED`.
4. If no scope resolution prefix is defined, the following steps are to be performed in order:
   - The variable is looked up in A's scope (i.e., in the MY scope). If an expression is bound to the variable, the expression is evaluated and returned.
   - Otherwise, the variable is looked up in B's scope (i.e., in the TARGET scope). If an expression is bound to the variable, the expression is evaluated and returned.
   - Otherwise, the variable is looked up in the ENVIRONMENT scope. If the variable is valid in the ENVIRONMENT scope, a value is computed and returned.
   - Otherwise, the value `UNDEFINED` is returned.

4.5.4 Detection of cyclic dependencies

Due to the semantics of variable lookup and expression evaluation, it is possible for an expression evaluation to have cyclic dependencies on variables which can result in infinite recursion. Note that this dependency can span across both classads involved in the match-making process. Every classad implementation is required to test for this condition when evaluating expressions.

Specifically, if a variable reference to a particular variable is triggered during the evaluation of the expression bound to that same variable, a cyclic dependency exists. (This condition is both necessary and sufficient.) In this situation, the value of the expression is `ERROR`.

4.6 Construction of expressions

In order to construct expressions, an implementation must provide a *Parse* function that produces an expression from an implementation dependent data source, such as a stream of characters. This parser must fail if the character stream does not represent a syntactically valid expression form.

5 Classified advertisement

This section defines the structure and operations defined on classads.

5.1 Structure

A classad is an aggregate of the following elements:

1. Type information. This is a pair of classad types as defined in section 3. These types are:
   - Self type (or My type)
   - Match type (or Target type)
2. Attribute list. An attribute list is a list of attributes. An attribute is a binding of an expression to a variable in the context of a given scope. In a given attribute list, at most one expression may be bound to a given variable. Therefore, an attribute list cannot contain duplicate variables.
3. Requirement expression. This is an expression that represents the advertising entity's constraints on the match.

4. Rank expression. This is an expression that quantifies the advertising entity's assessment of the quality of a match.

5. Advertisement ID. This is an integer that can be used by an advertising entity to keep track of several ads. There are no additional semantics defined on this field.

5.2 Operations

The list of operations that are defined on classads is enumerated below. The operations are described with the C++ function prototype syntax, i.e.

> *ReturnType* OperationName (*Arg1Type* Arg1Name, *Arg2Type* Arg2Name, ... , *ArgNType* ArgNName);

An argument is denoted call-be-reference if it is an "out" or "in-out" argument.

The notation used to denote types is as follows:

- *ClassAd* denotes the type of the classad structure itself.
- *AdType* denotes a classad type (section 3).
- *Id* is an identifier type which is used to identify variables (section 4.2.3).
- *Expr* denotes the classad expression type (section 4).
- *ValType* denotes a type that can hold any value supported by the classad expression system (section 4.1).
- *Success* is a flag type of the implementation environment that denotes success or failure of an operation.
- *Integer* is the integer type of the implementation environment.

The following operations are defined on classads:

1. *void* Construct(*ClassAd* &C);
   Every classad must be constructed with the following state:
   (a) MyType equal to `NULL`.
   (b) TargetType equal to `NULL`.
   (c) Empty attribute list.
   (d) Requirement expression set to the literal `UNDEFINED`.
   (e) Rank expression set to the literal `UNDEFINED`.
   (f) Advertisement ID set to zero.

2. *void* Destroy(*ClassAd* &C);
   The action and semantics of the destructor is implementation dependent.

3. *void* SetMyType(*ClassAd* &C, *AdType* T);
   Replace the ad's self type with the classad type value T.

4. *AdType* GetMyType(*ClassAd* C);
   Returns the current self type of the classad.

5. *void* SetTargetType(*ClassAd* &C, *AdType* T);
   Replace the ad's match type with the classad type value T.

6. *AdType* GetTargetType(*ClassAd* C);
   Returns the current match type of the classad.

7. *void* SetRequirementExpression(*ClassAd* &C, *Expr* E);
   Replace the ad's current requirement expression with the expression E.

8. *Expr* GetRequirementExpression(*ClassAd* C);
   Returns the requirement expression of the classad.

9. *void* SetRankExpression(*ClassAd* &C, *Expr* E);
   Replace the ad's current rank expression with the expression E.

10. *Expr* GetRankExpression(*ClassAd* C);
    Return the rank expression of the classad.

11. *void* SetAdID(*ClassAd* &C, *Integer* I);
    Replace the ad's current ID with the integer I.

12. *Integer* GetAdID(*ClassAd* C);
    Return the ID of the classad.

13. *void* InsertAttribute(*ClassAd* &C, *Id* V, *Expr* E);
    If V already exists in the attribute list of the ad, replace the current binding expression of V with E. Otherwise, insert the variable V with the binding E into the attribute list.

14. *Success* DeleteAttribute(*ClassAd* &C, *Id* V);
    If V exists in the attribute list, delete it and return success. Otherwise, return failure.

15. *Success* Lookup(*ClassAd* C, *Id* V, *Expr* &E);
    Return the expression bound to the variable V in classad C in E, and return success. If no such attribute exists, return failure.

16. *ValType* EvaluateExpression(*ClassAd* A, *ClassAd* B, *Expr* E);
    Evaluate the expression E using the attribute lists of A as the MY scope and B as the TARGET scope.

17. *Sucess* IsSymmetricMatch(*ClassAd* A, *ClassAd* B, *ValType* &RankA, *ValType* &RankB);
    Return success if and only if:

    (a) B's self type is equal to A's target type,
    (b) A's requirement expression evaluates to `TRUE`, and
    (c) A's self type is equal to B's target type,
    (d) B's requirement expression evaluates to `TRUE`.

    Additionally, if the match is successful, return A's rank of B in RankA, and B's rank of A in RankB.

18. *Success* IsOneWayMatch(*ClassAd* A, *ClassAd* B, *ValType* &RankA);
    Return success if and only if:

    (a) B's self type is equal to A's target type,
    (b) A's requirement expression evaluates to `TRUE`, and

    Additionally, if the match is successful, return A's rank of B in RankA.