Classified Advertisements

Presented by
Rajesh Raman
(raman@cs.wisc.edu)
Condor Team, University of Wisconsin–Madison

http://www.cs.wisc.edu/condor/

September 16, 1998

ClassAds: Historical Motivation

The Condor system required a mechanism to

- Represent resources and customers of the system
  - Expressive enough to describe entities and their policies
  - Simple enough to be general and efficient
- Identify compatible matches between resources and customers
- Rank compatible matches

Resource allocation in Condor is performed by matching classads.

ClassAds: Definition

The classad mechanism is a language for expressing and evaluating attributes.

- A classad is a set of named expressions
- Each named expression is called an attribute
- Expressions similar to those found in C/C++
  - Literals (boolean, integer, real, string, undefined, error)
  - Aggregates (classad, list)
  - Operators (arithmetic, logical, comparison, bitwise)
  - Attribute references
  - Calls to in-built functions
- Some extra features

ClassAd: Example

```
[
 MyType = "Apartment";
 SquareArea = 3500;
 RentOffer = 1000;
 HeatIncluded = False;
 OnBusLine = True;
 Rank = other.RentOffer+
        other.UnderGrad?0:10;
 Constraint = self.RentOffer
    - 150 < other.RentOffer
]
```

```
[
 MyType="ApartmentRenter";
 UnderGrad = False;
 RentOffer = 900;
 Rank = 1/(other.RentOffer
   + 100.0)
   +other.HeatIncluded?50:0;
 Constraint=other.OnBusLine
   && other.SquareArea>2700
   SquareArea > 2700
]
```

ClassAd: Another example

```
[
  Type       = "Machine";
  Activity   = "Idle";
  DayTime    = 36107;    // current time (secs since midnight)
  KbdIdle    = 1432;     // seconds
  Disk       = 323496;   // kbytes
  Memory     = 64;       // megabytes
  State      = "Unclaimed";
  LoadAvg    = 0.042969;
  Arch       = "INTEL";
  OpSys      = "SOLARIS251";
  Name       = "leonardo.cs.wisc.edu";
  RsrchGrp   = { "raman", "miron", "solomon", "jbasney" };
  Friends    = { "tannenba", "wright" };
  Untrusted  = { "rival", "riffraff" };
  Rank       = member(other.Owner, RsrchGroup) ? 10 : 0
                + member(other.Owner, Friends) ? 1 : 0;
  Constraint = !member(other.Owner, Untrusted)
                && (Rank == 10 ? true
                 : Rank > 0 ? (LoadAvg < 0.3 && KbdIdle > 15*60)
                 : (DayTime < 8*60*60 || DayTime > 18*60*60)))
]
```


Features of the ClassAd Language

- Lexical scoping
  - Supports hierarchical name spaces
  - Similar to (but more powerful than) C/C++'s lexical scoping
  - All expressions are evaluated in the "context" of a unique *top-level classad*
- Dynamic typing
  - Unlike C/C++ *variables* do not have types; only *values* do
  - Similar to LISP/Scheme
- Concrete types, representations and semantics for exceptional values (i.e., `undefined`, `error`)
- Aggregations (i.e., lists and classads) are orthogonal

Literals

Literals are one of the atoms of the classad language.

- Integer (e.g., `314`, `-17`, `0`, etc.)
  - 32-bit, signed, 2's complement
  - May be expressed in hex or octal (e.g., `0xff`, `0777`, etc.)
- Real (e.g., `3.14159`, `2.781`, etc.)
  - IEEE-754 64-bit double precision floating point numbers
- String (e.g., `"foo"`, `"bar"`, etc.)
  - May also include escaped sequences (see K&R, p.38)
- Boolean (`true`, `false`)
- Undefined (`undefined`)
- Error (`error`)

Operators

Almost all operators from C/C++ and Java are supported

- Arithmetic: `+`, `-`, `*`, `/`, `%`, (unary)`+`, (unary)`-`
- Logical: `&&`, `||`, `!`
- Bitwise: `&`, `|`, `~`, `^`, `<<`, `>>`, `>>>`
- Comparison: `<`, `<=`, `==`, `!=`, `>=`, `>`
- Others: `.`, `?:`, `[]`, and then some ...

Notably, no pointer or assignment operators

ClassAd expressions cannot *side-affect*; the language is *functional*.

Operators (contd.)

- Operators in the language are *total functions*
  - i.e., they are defined over all values in the language
- Almost all operators are *strict* on `undefined` and `error`
  - If an operand is `undefined` (`error`), the result is `undefined` (`error`)
  - `10 + undefined` ⇒ `undefined`
  - `error / 3.14` ⇒ `error`
- In addition, an expression evaluates to `error` if the operands are not in the primary domain of the operator
  - `10 * "foo"` ⇒ `error`
  - `17 / 0` ⇒ `error`

Operators (contd.): Non-strict operators

- Logical-AND (&&) and Logical-OR exhibit “don’t care” semantics

AND	F	T	U	E
F	F	F	F	F
T	F	T	U	E
U	F	U	U	E
E	E	E	E	E

OR	F	T	U	E
F	F	T	U	E
T	T	T	T	T
U	U	T	U	E
E	E	E	E	E

- The conditional operator is not strict for the two consequents `true?10:undefined` evaluates to 10, and `false?error:"foo"` evaluates to ”foo”
- The special comparison operators `is` and `isnt` are not strict `undefined is 10` evaluates to false, while `error is error` evaluates to true.

Aggregates (ClassAds)

The two aggregate structures in the language are classads and lists. ClassAds are created by the classad construction operator [ ]

- A sequence of zero or more attributes separated by semi-colons
  [*name0* = *expr0* ; *name1* = *expr1* ; . . . ; *namei* = *expri*]
- Each classad defines a scope from which attributes may be looked up
- May be arbitrarily nested
  - e.g., `[ foo=10; bar=[baz=20; foo=30] ]`
- Leads to a natural lexical scoping structure

More on the lexical scoping properties of ClassAds shortly.

Aggregates (Lists)

Lists are created by the list construction operator { }

- A sequence of zero or more expressions separated by commas
  { *expr0*, *expr1*, . . . , *expri* }
- *May be arbitrarily nested*
  - *e.g.,* `{ 10, [foo={10}], {17, [bar=3]} }`
- *May be treated as an array through use of the* subscript operator
  - *Elements numbered from 0 (like C/C++)*
  - *e.g.,* `{10, 17*2, 30}[1]` $\Rightarrow$ `34`

Attribute References

Attribute references are “similar” to variable references in C

1. If the reference is *attr*, find the expression bound to *attr* in the closest enclosing scope; the value of the attribute reference is the value of this expression
2. If the reference is *.attr*, find the expression bound to *attr* in the *top-level* ad; the value of the attribute reference is the value of this expression
3. If the reference is *expr.attr*, evaluate *expr* and proceed to lookup *attr* as in case 1, using the value of *expr* as the initial scope to search from. (If *expr* does not evaluate to a classad (or `undefined`), the value of the attribute reference is `error`.)

In all cases, if no attribute named *attr* can be found, the value of the attribute reference is `undefined`.

Attribute References (contd.): Examples of Cases 1 and 2

Top-level ClassAd (Case 1)	Value
[a=1;b=$\boxed{\texttt{a}}$]	1
[a=2;b=[c=1;d=$\boxed{\texttt{a}}$]]	2
[a=2;b=[c=1;d=$\boxed{\texttt{a+f}}$];e=[f=10]]	`undefined`
[a=3;b=[c=1;d=[e=5;f=$\boxed{\texttt{a+c+e}}$]]]	9
[a=3;b=[a=2;c=1;d=[e=5;f=$\boxed{\texttt{a+c+e}}$]]]	8
[a=3;b=[a=2;c=1;d=[e=5;f=$\boxed{\texttt{a+b+c}}$]]]	`error`
Top-level ClassAd (Case 2)	**Value**
[a=2;b=[a=1;d=$\boxed{\texttt{.a}}$]]	2
[a=3;b=[a=1;d=[a=5;f=$\boxed{\texttt{a+.a}}$]]]	8
[a=2;b=[c=1;d=$\boxed{\texttt{.c}}$]]	`undefined`

Attribute References (contd.): Examples of Cases 3

Top-level ClassAd (Case 3)	Value
`[a=1;b=`$\boxed{\texttt{[c=5].c}}$`]`	5
`[a=1;b=`$\boxed{\texttt{[c=5].a}}$`]`	1
`[a=1;b=[a=2;c=[b=.a]];d=`$\boxed{\texttt{.b.c.a}}$`]`	2
`[a=1;b=[a=2;c=[b=.a]];d=`$\boxed{\texttt{.b.c.b}}$`]`	1
`[a=1;b=[a=2;c=[b=a]];d=`$\boxed{\texttt{.b.c.b}}$`]`	2
`[a=1;b=[a=2;c=[b=.a]];d=`$\boxed{\texttt{.a.b.a.b}}$`]`	`error`

Note: Remember to use the lexical scoping of the evaluated expression and not the lexical scoping of the attribute reference (i.e., the box)!!

Attribute References (contd.): Example Top-level ClassAd

```
[
  adl=[ other = .adr.self;
        self = [ Owner = "Ms. Foo";
                 Arch = "INTEL";
                 MemorySize = 32M;
                 Constraint = other.Owner != "foo"
               ];
      ];
  adr=[ other = .adl.self;
        self = [ Owner = "Mr. Bar";
                 MemorySize = 16M;
                 Constraint = (other.Arch=="INTEL) &&
                   (other.MemorySize > self.MemorySize)
               ]
      ]
]
```

Attribute References (contd.): The "super" attribute

Strict lexical scoping can be changed with the `super` attribute

- A classad may have an attribute named `super`, which will (upon evaluation to a classad) be treated as the parent scope of the ad. For example, `[a=1;b=[c=2];d=[super=.b]].d.c` evaluates to 2
- The `super` attribute may also be used as attribute references in expressions
  - If an explicit `super` attribute is not supplied, the reference evaluates to the natural parent scope of the expression's scope
  - For example, `[a=1;b=[a=7;c=super.a]]` evaluates to 1

Attribute References (contd.): Circularities

The ClassAd language requires that circularities be detected during evaluation

- Circularities in expression evaluation
  `[b = a; a = b].a` evaluates to `undefined`
- Circularities in scoping
  `[a=[super=.b]; b=[super=.a]].x` evaluates to `undefined`

Function Calls

Expressions may also consist of calls to inbuilt functions.

- The arity and semantics of a call depends on the function
- The following functions are currently supported

`isString(e)`	True iff `e` evaluates to a string
`isInteger(e)`	True iff `e` evaluates to a integer
`isReal(e)`	True iff `e` evaluates to a real
`isClassAd(e)`	True iff `e` evaluates to a classad
`isList(e)`	True iff `e` evaluates to a list
`isMember(l,e)`	True iff `e` equals a member of the list `l`
`isMember(l,e)`	True iff `e` `is` a member of the list `l`

- Will add more (type conversions, mathematical functions, string functions, pattern matching, time, etc.)

Status of implementations

- The Java implementation is complete
- The C++ implementation is undergoing final touches
  - Will be complete coming monday (Sep 21)
  - Some aspects of the library (such as memory management) require testing — you are the guinea pigs! Send bug reports to me (raman@cs.wisc.edu)
- Watch http://www.cs.wisc.edu/condor/classad/classad.html for new developments