16.47   PM: A REDUCE pattern matcher

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica, and is based on the pattern matcher described in Kevin McIsaac, "Pattern Matching Algebraic Identities", SIGSAM Bulletin, 19 (1985), 4-13.

Author: Kevin McIsaac.

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica, and is based on the pattern matcher described in Kevin McIsaac, "Pattern Matching Algebraic Identities", SIGSAM Bulletin, 19 (1985), 4-13. The following is a description of its structure.

A *template* is any expression composed of literal elements, e.g. `5`, `a`, or `a+1`, and specially-denoted pattern variables, e.g. `?a` or `??b`. Atoms beginning with `?` are called *generic variables* and match any expression.

Atoms beginning with `??` are called *multi-generic variables* and match any expression or any sequence of expressions including the null or empty sequence. A sequence is an expression of the form `[a1,a2,...]`. When placed in a function argument list the brackets are removed, i.e. `f([a,1]) -> f(a,1)` and `f(a,[1,2],b) -> f(a,1,2,b)`.

A template is said to match an expression if the template is literally equal to the expression, or if by replacing any of the generic or multi-generic symbols occurring in the template, the template can be made to be literally equal to the expression. These replacements are called the *bindings* for the generic variables. A replacement is an expression of the form `exp1 -> exp2`, which means `exp1` is replaced by `exp2`, or `exp1 -> exp2`, which is the same except `exp2` is not simplified until after the substitution for `exp1` is made. If the expression has any of the properties associativity, commutativity, or an identity element, they are used to determine if the expressions match. If an attempt to match the template to the expression fails the matcher backtracks, unbinding generic variables, until it reaches a place where it can make a different choice. It then proceeds along the new branch.

The current matcher proceeds from left to right in a depth-first search of the template expression tree. Rearrangements of the expression are generated when the match fails and the matcher backtracks.

The matcher also supports *semantic* matching. Briefly, if a subtemplate does not match the corresponding subexpression because they have different structures, then the two are equated and the matcher continues matching the rest of the expression until all the generic variables in the subexpression are bound. The equality is then checked. This is controlled by the switch `semantic`. By default it is `on`.

717

16.47.1 `M(exp,temp)`

The template `temp` is matched against the expression `exp`. If the template is
literally equal to the expression `T` is returned. If the template is literally equal to
the expression after replacing the generic variables by their bindings then the set of
bindings is returned as a set of replacements. Otherwise `0` (`nil`) is returned.

Examples:

A "literal" template:

```
m(f(a), f(a));
t
```

Not literally equal:

```
m(f(a), f(b));
0
```

Nested operators:

```
m(f(a,h(b)), f(a,h(b)));
t
```

"Generic" templates:

```
m(f(a,b), f(a,?a));
{?a -> b}
m(f(a,b), f(?a,?b));
{?b -> b, ?a -> a}
```

The multi-generic symbol `??a` matches the "rest" of the arguments:

```
m(f(a,b), f(??a));
{??a -> {[a, b]}
```

but the generic symbol `?a` does not:

```
m(f(a,b), f(?a));
0
```

Flag *h* as "associative":

```
flag('(h), 'assoc);
```

Associativity is used to group terms together:

```
m(h(a,b,d,e), h(?a,d,?b));
{?b -> e, ?a -> h(a,b)}
```

"plus" is a symmetric function:

```
m(a+b+c, c+?a+?b);
{?b -> a, ?a -> b}
```

and it is also associative

```
m(a+b+c, b+?a);
{?a -> c + a}
```

Note that the effect of using a multi-generic symbol is different:

```
m(a+b+c,b+??c);
{??c -> [c,a]}
```

16.47.2   temp _= logical_exp

A template may be qualified by the use of the conditional operator _=, `such!-that`. When a `such-that` condition is encountered in a template, it is held until all generic variables appearing in `logical_exp` are bound.

On the binding of the last generic variable, `logical_exp` is simplified and if the result is not `T` the condition fails and the pattern matcher backtracks. When the template has been fully parsed any remaining held `such-that` conditions are evaluated and compared to `T`.

Examples:

```
m(f(a,b), f(?a,?b\_=(?a=?b)));
0
m(f(a,a), f(?a,?b\_=(?a=?b)));
{?b -> a, ?a -> a}
```

Note that `f(?a,?b_=(?a=?b))` is the same as `f(?a,?a)`.

16.47.3   S(exp,{temp1 → sub1, temp2 → sub2, . . . }, rept, depth)

Substitute the set of replacements into `exp`, re-substituting a maximum of `rept` times and to a maximum depth `depth`. `rept` and `depth` have the default values of 1 and ∞ respectively. Essentially, `S` is a breadth-first search-and-replace. (There is also a depth-first version, `Sd(...)`.) Each template is matched against `exp` until a successful match occurs.

Any replacements for generic variables are applied to the r.h.s. of that replacement and `exp` is replaced by the r.h.s. The substitution process is restarted on the new expression starting with the first replacement. If none of the templates match `exp` then the first replacement is tried against each sub-expression of `exp`. If a matching template is found then the sub-expression is replaced and process continues with the next sub-expression.

When all sub-expressions have been examined, if a match was found, the expression is evaluated and the process is restarted on the sub-expressions of the resulting expression, starting with the first replacement. When all sub-expressions have been examined and no match found the sub-expressions are reexamined using the next replacement. Finally when this has been done for all replacements and no match found then the process recures on each sub-expression. The process is terminated after `rept` replacements or when the expression no longer changes.

The command

```
Si(exp, {temp1 -> sub1, temp2 -> sub2, ...}, depth)
```

means "substitute infinitely many times until expression stops changing". It is short-hand for `S(exp,{temp1 -> sub1, temp2 -> sub2,...},Inf, depth)`.

Examples:

```
s(f(a,b), f(a,?b) -> ?b\^{}2);
2
b
s(a+b, a+b -> a{*}b);
b*a
```

"Associativity" is used to group $a + b + c$ to $(a + b) + c$:

```
s(a+b+c, a+b -> a*b);
b*a + c
```

The next three examples use a rule set that defines the factorial function. Substitute

once:

```
s(nfac(3), {nfac(0) -> 1, nfac(?x) -> ?x*nfac(?x-1)});
3*nfac(2)
```

Substitute twice:

```
s(nfac(3), {nfac(0) -> 1, nfac(?x) -> ?x*nfac(?x-1)}, 2);
6*nfac(1)
```

Substitute until expression stops changing:

```
si(nfac(3), {nfac(0) -> 1, nfac(?x) -> ?x{*}nfac(?x-1)});
6
```

Only substitute at the top level:

```
s(a+b+f(a+b), a+b -> a*b, inf, 0);
f(b+a) + b*a
```

16.47.4   temp :- exp and temp ::- exp

If during simplification of an expression, `temp` matches some sub-expression, then that sub-expression is replaced by `exp`. If there is a choice of templates to apply, the least general is used.

If an old rule exists with the same template, then the old rule is replaced by the new rule. If `exp` is `nil` the rule is retracted.

`temp ::- exp` is the same as `temp :- exp`, but the l.h.s. is not simplified until the replacement is made.

Examples:

Define the factorial function of a natural number as a recursive function and a termination condition. For all other values write it as a gamma function. Note that the order of definition is not important, as the rules are re-ordered so that the most specific rule is tried first. Note the use of `::-` instead of `:-` to stop simplification of the l.h.s. `hold` stops its arguments from being simplified.

```
fac(?x \_= Natp(?x)) ::- ?x*fac(?x-1);
hold(fac(?X-1)*?X)
fac(0) :- 1;
1
```

```
fac(?x) :- Gamma(?x+1);
gamma(?X + 1)
fac(3);
6
fac(3/2);
gamma(5/2)
```

16.47.5   Arep({rep1,rep2,...})

In future simplifications automatically apply replacements `rep1, rep2, ...`
until the rules are retracted. In effect, this replaces the operator `->` by `:-` in the set
of replacements `{rep1, rep2, ...}`.

16.47.6   Drep({rep1,rep2,..})

Delete the rules `rep1, rep2, ....`

As we said earlier, the matcher has been constructed along the lines of the pat-
tern matcher described in McIsaac with the addition of such-that conditions and
"semantic matching" as described in Grief. To make a template efficient, some
consideration should be given to the structure of the template and the position of
such-that statements. In general the template should be constructed so that fail-
ure to match is recognized as early as possible. The multi-generic symbol should
be used whenever appropriate, particularly with symmetric functions. For further
details see McIsaac.

Examples:

`f(?a,?a,?b)` is better than `f(?a,?b,?c_=(?a=?b))`. `?a+??b` is better
than `?a+?b+?c....`

The template `f(?a+?b,?a,?b)`, matched against `f(3,2,1)` is matched as
`f(?e_=(?e=?a+?b),?a,?b)` when semantic matching is allowed.

16.47.7   Switches

TRPM Produces a trace of the rules applied during a substitution. This is useful to
see how the pattern matcher works, or to understand an unexpected result.

In general usage the following switches need not be considered:

SEMANTIC Allow semantic matches, e.g. `f(?a+?b,?a,?b)` will match
`f(3,2,1)`, even though the matcher works from left to right.

SYM!-ASSOC Limits the search space of symmetric associative functions when the template contains multi-generic symbols so that generic symbols will not function. For example `(a+b+c,?a+??b)` will return

```
{?a -> a, ??b -> [b,c]} or
{?a -> b, ??b -> [a,c]} or
{?a -> c, ??b -> [a,b]}
```

but not `{?a -> a+b, ??b -> c}`, etc. No sane template should require these types of matches. However they can be made available by turning the switch off.