Kore Simplification

The K Framework Team
Runtime Verification, Inc.

Chapter 1

Notation

In addition to the syntax of matching logic and Kore (described elsewhere), we adopt the convention in this document that italic variables (e.g., t, p, s) are meta-variables.

Simplification operates on *program configurations*, which are the conjunction (with $\wedge$) of a *term* representing the user-defined state, a *predicate* representing the conditions accumulated during execution, and a *substitution* produced during rule unification. The predicate is a matching logic predicate, i.e., it reduces to $\top$ or $\bot$. The substitution is a conjunction of predicates of the form $x = \phi$ where x is a matching logic variable and ϕ is any pattern. Configurations are represented by the type `ExpandedPattern`, and in this document are denoted by a triple (t, p, s) where t is the term, p is the predicate, and s is the substitution. A configuration triple is logically equivalent to its conjunction $(t \wedge p \wedge s)$ but we find it instructive to give a distinct syntax.

Chapter 2

`\or` simplification

2.1 Driver

`simplify` is a driver responsible for breaking down an `\or` pattern and simplifying its children.

```
{-|'simplify' simplifies an 'Or' pattern with 'OrOfExpandedPattern'
children by merging the two children.
-}
simplify
    ::  ( MetaOrObject level
        , SortedVariable variable
        , Ord (variable level)
        , Show (variable level)
        , Unparse (variable level)
        )
    => Or level (OrOfExpandedPattern level variable)
    ->  ( OrOfExpandedPattern level variable
        , SimplificationProof level
        )
simplify
    Or
        { orFirst = first
        , orSecond = second
        }
  =
```

```
    simplifyEvaluated first second

{-| simplifies an 'Or' given its two 'OrOfExpandedPattern' children.

See 'simplify' for detailed documentation.
-}
{- TODO (virgil): Preserve pattern sorts under simplification.

One way to preserve the required sort annotations is to make 'simplifyEvaluated'
take an argument of type

> CofreeF (Or level) (Valid level) (OrOfExpandedPattern level variable)

instead of two 'OrOfExpandedPattern' arguments. The type of 'makeEvaluate' may
be changed analogously. The 'Valid' annotation will eventually cache information
besides the pattern sort, which will make it even more useful to carry around.

-}
-- TODO(virgil): This should do all possible mergings, not just the first
-- term with the second.
simplifyEvaluated
    ::  ( MetaOrObject level
        , SortedVariable variable
        , Ord (variable level)
        , Show (variable level)
        , Unparse (variable level)
        )
    => OrOfExpandedPattern level variable
    -> OrOfExpandedPattern level variable
    ->  ( OrOfExpandedPattern level variable
        , SimplificationProof level
        )
simplifyEvaluated first second

  | (head1 : tail1) <- OrOfExpandedPattern.extractPatterns first
  , (head2 : tail2) <- OrOfExpandedPattern.extractPatterns second
  , Just (result, proof) <- simplifySinglePatterns head1 head2
  = (OrOfExpandedPattern.make $ result : (tail1 ++ tail2), proof)
```

```
  | otherwise =
    ( OrOfExpandedPattern.merge first second
    , SimplificationProof
    )

  where
    simplifySinglePatterns first' second' =
        disjoinPredicates first' second' <|> topAnnihilates first' second'
```

2.2 Disjoin predicates

When two configurations have the same substitution, it may be possible to simplify the pair by disjunction of their predicates.

$$\begin{aligned}(t_1, p_1, s) \vee (t_2, p_2, s) &= ([t_2 \vee \neg t_2] \wedge t_1, p_1, s) \vee ([t_1 \vee \neg t_1] \wedge t_2, p_2, s) \\ &= (t_1 \wedge t_2, p_1 \vee p_2, s) \vee (\neg t_2 \wedge t_1, p_1, s) \vee (\neg t_1 \wedge t_2, p_2, s)\end{aligned}$$

It is useful to apply the above equality when $\neg t_2 \wedge t_1 = \neg t_1 \wedge t_2 = \bot$, so that

$$(t_1, p_1, s) \vee (t_2, p_2, s) = (t_1 \wedge t_2, p_1 \vee p_2, s), \quad \neg t_2 \wedge t_1 = \neg t_1 \wedge t_2 = \bot \ . \tag{2.1}$$

In general, it may be expensive to check the side condition of Eq. (2.1), so in practice we limit ourselves to applying this simplification when $t_1 = t_2$.

Note: It is not clear that we should *ever* apply this simplification. We attempt to refute the conditions on configurations using an external solver to reduce the configuration space for execution. The solver operates best when it has the most information, and the predicate $p_1 \vee p_2$ is strictly weaker than either p_1 or p_2.

```
{- | Merge two configurations by the disjunction of their predicates.

This simplification case is only applied if the configurations have the same
'term'.

 -}
disjoinPredicates
    ::  ( MetaOrObject level
        , SortedVariable variable
```

```
        , Ord (variable level)
        , Show (variable level)
        , Unparse (variable level)
        )
    => ExpandedPattern level variable
    -- ^ Configuration
    -> ExpandedPattern level variable
    -- ^ Disjunction
    -> Maybe (ExpandedPattern level variable, SimplificationProof level)
disjoinPredicates
    predicated1@Predicated
        { term = term1
        , predicate = predicate1
        , substitution = substitution1
        }
    Predicated
        { term = term2
        , predicate = predicate2
        , substitution = substitution2
        }

  | term1         == term2
  , substitution1 == substitution2
  = Just (result, SimplificationProof)

  | otherwise =
    Nothing

  where
    result = predicated1 { predicate = makeOrPredicate predicate1 predicate2 }
```

2.3 `\top` annihilates `\or`

$\top$ is the annihilator of $\vee$; when two configurations have the same substitution, it may be possible to use this property to simplify the pair by annihilating the lesser term.

$$
\begin{aligned}
(\top, p_1, s) \vee (t_2, p_2, s) &= (\top, [p_2 \vee \neg p_2] \wedge p_1, s) \vee (t_2, [p_1 \vee \neg p_1] \wedge p_2, s) \\
&= (\top, p_1 \wedge p_2, s) \vee (\top, p_1 \wedge \neg p_2, s) \vee (t_2, \neg p_1 \wedge p_2, s)
\end{aligned}
$$

It is useful to apply the above equality when $\neg p_2 \wedge p_1 = \neg p_1 \wedge p_2 = \bot$, so that

$$
(\top, p_1, s) \vee (t_2, p_2, s) = (\top, p_1 \wedge p_2, s), \quad \neg p_2 \wedge p_1 = \neg p_1 \wedge p_2 = \bot \ . \tag{2.2}
$$

In general, it may be expensive to check the side condition of Eq. (2.2), so in practice we limit ourselves to applying this simplification when $p_1 = p_2$.

```
{- | 'Top' patterns are the annihilator of 'Or'.
 -}
topAnnihilates
    ::  ( MetaOrObject level
        , SortedVariable variable
        , Ord (variable level)
        , Show (variable level)
        , Unparse (variable level)
        )
    => ExpandedPattern level variable
    -- ^ Configuration
    -> ExpandedPattern level variable
    -- ^ Disjunction
    -> Maybe (ExpandedPattern level variable, SimplificationProof level)
topAnnihilates
    predicated1@Predicated
        { term = term1
        , predicate = predicate1
        , substitution = substitution1
        }
    predicated2@Predicated
        { term = term2
        , predicate = predicate2
        , substitution = substitution2
        }

  -- The 'term's are checked first because checking the equality of predicates
  -- and substitutions could be expensive.
```

```
| _ :< TopPattern _ <- Recursive.project term1
, predicate1    == predicate2
, substitution1 == substitution2
= Just (predicated1, SimplificationProof)

| _ :< TopPattern _ <- Recursive.project term2
, predicate1    == predicate2
, substitution1 == substitution2
= Just (predicated2, SimplificationProof)

| otherwise =
  Nothing
```