Reduction

Example

LE can be used to perform logic by constructing an expression that represents the axioms of a system and then reducing the expression to it's canonical form.
If an expression reduces to F then the expression contains a contradiction.
If an expression reduces to T then the system is consistent, that is, always true.

For instance, the expression...
`A and (if A then B)`
...is written as an LE like so...
`(T (A (A (T B))))`
...and, using deiteration and double-cut elimination, this expressions reduces to...
`(T (A B))`
...which is equivalent to...
`A and B`

Note that the reduction process has performed the equivalent of reduction and produced a simpler expression that entails all the same information as the starting expression.

Overview

Expressions are reduced by looking for opportunities to rewind/reverse an application of one of the LE rewrite rules.
Completely reversing all the inferences embedded in an expression produces an expression that is canonical.
Expressions that minimize to T/F are tautologies/contradictions.

Just an observation, might not mean anything, but there are three fundamental forms of inference, and Lucid reduction reverses all three...

- Induction : LE reduction reduces the simplest non-canonical term in an expression first, thus working from simpler expressions to more complex expressions.
- Deduction : Grounding Cofactors(represented as implications) and deduction are used to identify *critical terms* in *mostly-canonical* expressions.
- Abduction : Rules are applied using critical terms and only those results are simpler are used.

Grounding Cofactors and Critical Terms

The reduction method computes, and remembers, all grounding cofactors, or *groundings* of all subterms in *mostly-canonical* expressions.

A *term cofactor* of a boolean function F is derived by replacing all instances of a given term S in F with a constant value (T of F).

A *grounding cofactor* is a term cofactor that is equivalent to T of F. A grounding cofactor's terms is called a *critical term*.

The reduction method uses grounding cofactors to discover opportunities to apply the rules in a way that reduces an expression.

A grounding cofactor can be represented by a tuple *Grounding(E,S,PN,G)*,
where Grounding(E,S,PN,G) represents a cofactor E(S->PN) of an expression,
where PN is a constant and E(S->PN)'s canonical form is a constant G.
Meaning that when sub-expression S of E has the value PN then E reduces to G.
Put another way, a grounding represents one of these implications; S->E, ~S->E, S->~E, ~S->~E.

Grounding are used to compute erasures and de-iterations.
When one side of an expression contains a negative grounding then it can be erased from the other side (deiteration).
If the other side of the formula is empty (T) then the term is iterated to one side (replacing the T) and then erased/deiterated.

Note that a grounding can be written in CNF form with clauses of size <= 2.
Meaning that new groundings can be derived from existing grounding in polytime (aka krom logic).
Also meaning that it's also easy to keep the set of groundings transitively, or maximally, complete.
Think of all the cofactors discovered in an expression stored as a set of clauses. Then, whenever this set is expanded, resolution is performed until complete (in polytime) in order to derive any newly implied grounding and equivalences.

Groundings are conditional equivalences.
Or, equivalences are 2 groundings where PN == G and PN != G.

If an expression has no grounding cofactors then an expression is canonical.

Mostly Canonical Expressions

An expression E = (x y) is a *mostly-canonical* expression if x and y are canonical but E is not.
There's a big difference between mostly-canonical and all-canonical.
With all-canonical, well, with all-canonical there's usually only one thing you can do 😃.

Reduction

This section contains a pseudo-code description of the REDUCE function.
The Reduce function accepts an expression and returns the canonical form of the expression.

```
Let CANONICAL = a global list of expressions known to be canonical.
Let GROUNDINGS = a global, indexed, table of tuples that represent all known
grounding cofactors and all derivable cofactors.

Function Reduce
{
    Let START = the expression to be reduced.
    Let NEXT =  the current reduction of START, initialized to START.
    While NEXT has terms not in CANONICAL
    {
        Let SMALLEST = the simplest term in NEXT that's not known to be canonical.

        Compute and add all groundings of SMALLEST to GRONUDINGS.
            if SMALLEST is
```

```
        if (SMALLEST == NEXT) // true when NEXT is mostly-canonical
        {
            For terms S that are common to both sides of an expression
            {
                If S is a negative cofactor of one side of NEXT then
                {
                    S can be erased from the other side of NEXT.
                    That is...
                    Let L and R be terms such that NEXT = (L R)
                    If S is a negative cofactor of L then let NEXT = (L R(S->T))
                    If S is a negative cofactor of R then let NEXT = (L(S->T) R)
                }
            }

            If one side of NEXT is empty (T) then
                for any positive cofactor of the other side
                    the cofactor is iterated to the empty side (replacing the T)
and   then erased/deiterated.


        }
        else
            Let REDUCED = Reduce(SMALLEST);
            Replace all instances of SMALLEST in NEXT with REDUCED.
    }

    Add NEXT to CANONICAL, if not already there.

    return NEXT
}
```

Equivalence

After computing groundings UKS can use them to discover erasures and de-iterations that can simplify an expression.
When a simplification is discovered the fact is record in the ABOUT table as an equivalence between two expressions.

A clause that records an equivalence between two formulas AND proof that its valid. The 'proof' also provides the necessary data to reverse the operation. Is equivalent to a line of identity in an existential graph.
UKS computes several types; constant reductions, erasures, de-iterations.

Complexity

- LEs can be reduced in polynomial time.

  > The satisfiability of LEs can also be determined in polynomial time.