Algorithms for the GER Representation of Pos

Supervisors - Harald Søndergaard, Peter Schachte

March 8, 2019

1 Purpose

This project provides an imperative implementation of the GER representation described by R. Bagnara and P. Schachte in [1].The implementation, which we refer to as the FR representation, was developed as an extension of an existing ROBDD library written by Schachte, for use in a groundness analyser. The purpose of the F component (representing Ground and Equivalent variables) is to reduce the size of the ROBDDs by extracting redundant information. Therefore, the primary problems to solve in this project were (1) How best to represent F, and (2) how to implement the standard binary operations (and, or, implies), while preserving F information from the operands and extracting consequential F information.

2 Definitions

We represent a boolean function ϕ as a pair (F, R), where F is an equivalence relation on $Vars \cup \{\top, \bot\}$, and R is an ROBDD.

2.a Semantics

- $\mathcal{S}(F, R) = \mathcal{S}(F) \wedge \mathcal{S}(R)$
- $\mathcal{S}_F(F) = \bigwedge \{x \leftrightarrow y | (x, y) \in F\}$ (Note: $x \leftrightarrow \top \equiv x$, and $x \leftrightarrow \bot \equiv \neg x$)
- $\mathcal{S}_R(\mathbf{R}) = var_R \wedge \mathcal{S}(R_{then}) \ \vee \ \neg var_R \wedge \mathcal{S}R_{else}$
- $\mathcal{S}_R(\mathbf{0}) = \bot$
- $\mathcal{S}_R(\mathbf{1}) = \top$

2.b Invariants

To preserve the canonical form inherited from the ROBDD, we enforce the following invariants on any given (F, R) pair.

- $\forall x \in Vars : ((\exists y \in Vars : \ (y < x) \wedge \ \mathcal{S}(F, R) \vDash x \leftrightarrow y) \ \Rightarrow \ x \notin dep(R))$

  Essentially, for any set of equivalent variables entailed by $\mathcal{S}(F, R)$, either only the least variable in the set appears in the ROBDD, or none do.

- $\forall x \in Vars : (\mathcal{S}(F,R) \vDash x \vee \mathcal{S}(F,R) \vDash \neg x) \;\Rightarrow\; x \notin dep(R)$
- $\mathcal{S}_F(F) \equiv \bot \Leftrightarrow \mathcal{S}_R(R) \equiv \bot$

  The following invariants are redundant, given the previous properties, however they are included for clarity.

- $\forall x, y \in Vars : \mathcal{S}(F,R) \vDash x \leftrightarrow y \;\Rightarrow\; (\mathcal{S}_F(F) \vDash x \leftrightarrow y \wedge \mathcal{S}_R(R) \nvDash x \leftrightarrow y)$
- $\forall x \in Vars : \mathcal{S}(F,R) \vDash x \;\Rightarrow\; (\mathcal{S}_F(F) \vDash x \wedge \mathcal{S}_R(R) \nvDash x)$
- $\forall x \in Vars : \mathcal{S}(F,R) \vDash \neg x \;\Rightarrow\; (\mathcal{S}_F(F) \vDash \neg x \wedge \mathcal{S}_R(R) \nvDash \neg x)$

  Lastly we must canonicalise the F-representation of $\bot$ (The below implementation will explain how it is possible to represent a contradiction).

- $\mathcal{S}_F(F) \vDash \bot \;\Rightarrow\; \forall x, y \in Vars : (x, y) \in F$

  Therefore, the canonical representations of $\top$ and $\bot$ are $(I, \mathbf{1})$ and $(T, \mathbf{0})$, where I, T are the identity and total relations, and $\mathbf{1}, \mathbf{0}$ are the true and false ROBDDs.

3 F implementation

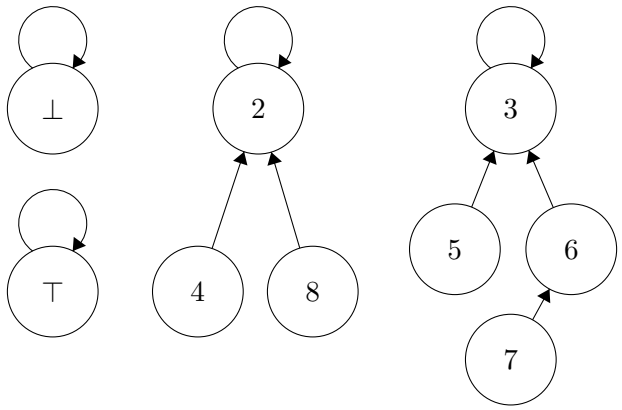
The first structure we considered to represent F was a table representation of the relation, implemented as an array of bitsets, where bit y in bitset x is a 1-bit iff $(x, y) \in F$ (a 0-bit otherwise). With a reasonable limit of 62 variables, this would mean an array of 64 bitsets, each 64 bits long, with table coordinates 0 and 1 reserved for $\bot$ and $\top$ respectively. This data structure would provide extremely logical conjunction and disjunction operations on equivalence relations, being a simple inclusive-OR or AND operation on the bits. However, this structure loses transitivity under conjunction and holds a massive amount of redundant information. For example, if *w, x, y* & *z* are all equivalent, the table tells us w is equivalent to x, y, z, x is equivalent to w, y, z, y is equivalent to w, x, z *etc.*

3.a Union-Find F-Equivalence

Noting transitivity as a priority, we instead opted for a union-find data structure. F is represented as an array of variables, where $F[x] = y$ iff $(x \leq y$ **and** $(x, y) \in F)$. In this situation, y is considered the *parent* of x, being a smaller-equivalent variable (so $parent(x) = F[x]$). Again we reserve 0 and 1 as variable indexes for $\bot$ and $\top$ respectively, but will refer to them by name below for clarity (instead of index). The least-equivalent variable we call the *root* of x, and can be found through repeated iteration of array indexing ($F[F[F[....F[x]]]]$), until a fixed point is reached (the parent of a root is itself). This data structure can be visualised as clustered DAGs, where each cluster is an equivalence class, and the sink node in each cluster is the root variable node of that equivalence class.

Figure 1: An example F-equivalence

x	$\bot$	$\top$	2	3	4	5	6	7	8
$F[x]$	$\bot$	$\top$	2	3	2	3	3	6	2




```
1: function UNION-SIMPLE(a, b)
2:     if root(a) < root(b) then
3:         F[root(b)] ← root(a))
4:     else if root(a) > root(b) then
5:         F[root(a)] ← root(b))
```

Two variables x and y are therefore equivalent iff $\mathbf{root}(x) = \mathbf{root}(y)$. To ensure that any equivalence class does not have more than one root (sink) node, we implement the union algorithm so that the roots of the inputs are linked, rather than the inputs themselves.

The use of this function for unifying variables gives us transitivity for free, as we can check equivalence of variables by checking if $\mathbf{root}(a) = \mathbf{root}(b)$. The condition that ensures the larger root is parented by the smaller root imposes a strict ordering on the variables along any path to the root node, such that the root of any variable is always the least variable in it's entire equivalence class. This is valuable, as it allows us to pick a canonical representative of each equivalence class for use in the ROBDD.

This data structure functions correctly using only the union algorithm (and its primitive root algorithm) , however, unifying many variables in descending order can lead to long chains of variables, which leaves our test for equivalence algorithm at $O(n)$ time to find the roots and compare. To avoid this, we would like to link child nodes directly to their roots wherever possible, therefore shortening the path from any grandchild-nodes. This is done during the root-finding algorithm, $\mathbf{find}(x)$, which, after following a path from a child node to the root-node, recursively links every node along that path directly to the root.

We update the **union** algorithm to use **find**. Employing **find** every time a root is needed amortises the look-up cost, but we still have a worst case complexity of $O(n)$, in the case where

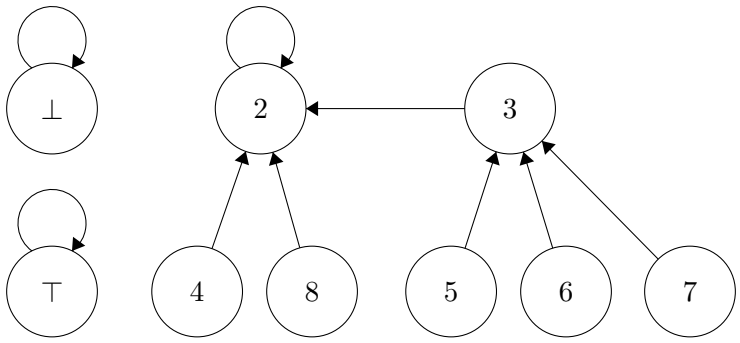
```
1: function FIND(x)
2:     if F[F[x]] ≠ F[x] then                ▷ Check if x's parent isn't a root
3:         F[F[x]] ← FIND(F[x])              ▷ Link x's parent directly to the root
4:     F[x] = F[F[x]]
5:     return F[x]
```

many variables are linked in descending order. Other implementations of this structure will often compare the *rank/size* of the roots to ensure the root of the smaller group of nodes is pointed to the root of the larger, keeping the majority of paths short. This metric gives an amortised constant time **find** operation, however, we would then have to scan the array in linear time to determine the correct least-equivalent variable.

```
1: function UNION(a, b)
2:     a_root, b_root ← FIND(a), FIND(b)
3:     if a_root < b_root then
4:         F[b_root] ← a_root
5:     else if a_root > b_root then
6:         F[a_root] ← b_root
```

Figure 2: The result of calling **union**(7,8) on the equivalence from Fig 1.

x	$\bot$	$\top$	2	3	4	5	6	7	8
$F[x]$	$\bot$	$\top$	2	2	2	3	3	3	2



The following functions naturally arises from **union** and **find**.

- **entail**(x) := **union**$(\top, x)$
- **disentail**(x) := **union**$(\bot, x)$
- **are_equivalent**(x, y) := (**find**(x) = **find**(y))

- **is_entailed**$(x) := (\textbf{find}(x) = \top)$
- **is_disentailed**$(x) := (\textbf{find}(x) = \bot)$

Lastly, we redefine $\mathcal{S}_F$ such that if **is_disentailed**$(\top)$ is true, then $\mathcal{S}_F(F) = \bot$. This happens if either **entail**(x) is called where **is_disentailed**(x) is true, or **disentail**(x) is called where **is_entailed**(x) is true. In terms of the actual code, we use a special pointer value for the false-equivalence, to which any equivalence is converted as soon as a contradiction occurs. This avoids the need to write the total equivalence to the entire array (as specified under the invariances), and allows the false-equivalence to be recognised and passed efficiently.

3.b Equivalence operations

In order to perform logical operations on (F, R) pairs, we must first define them on Equivalences.

3.b.1 equiv_and

The logical AND of two equivalences, F, G, is the transitive closure of their union. For example (ignoring symmetry) if $x \leftrightarrow y \in F$ and $y \leftrightarrow z \in G$, then $x \leftrightarrow y, y \leftrightarrow z, x \leftrightarrow z$ should all be in **equiv_and**(F, G).

To simplify the access of root variables, we first apply the algorithm **flatten** to both F and G.

```
1: function FLATTEN(F)
2:     for x in Vars do
3:         find(x)
```

To perform the union, we allocate a new Equivalence H to write the result, and proceed in ascending order along the F and G arrays in parallel. For each variable index x, we inspect it's root in F, $\textbf{root}_\textbf{F}(x)$ and its root in G, $\textbf{root}_\textbf{G}(x)$ (remember these are just $F[x]$ and $G[x]$ because of the **flatten** calls). If $\textbf{root}_\textbf{F}(x)$ and $\textbf{root}_\textbf{G}(x)$ are different, we have 3 different variables we must unify. Writing either of these roots of x at $H[x]$ will lose the connection to the other, so we must first unify the roots in H. This means simply finding the smaller of the roots of the F and G roots in H, i.e. $common\_root = \textbf{min}(H[\textbf{root}_\textbf{F}(x)], H[\textbf{root}_\textbf{F}(x)])$. This will always be defined, because $\textbf{root}_\textbf{F}(x)$ and $\textbf{root}_\textbf{G}(x)$ must both be smaller than x and we have already assigned parents for every variable up to x in H. We now set $H[\textbf{root}_\textbf{F}(x)], H[\textbf{root}_\textbf{F}(x)]$ and $H[x]$ all equal to $common\_root$, and we have the unification. Now any variable linked to $\textbf{root}_\textbf{F}(x)$ or $\textbf{root}_\textbf{G}(x)$ in either F or G will be transitively linked to $common\_root$. For the case where $\textbf{root}_\textbf{F}(x)$ $\textbf{root}_\textbf{G}(x)$, we can actually use this same process, as we are essentially just writing $H[F[x]]$ to $H[x]$ instead of just $F[x]]$, which points x directly to its root, rather than going through $F[x]$. One extra step The below algorithm is presented in simple form, but can be easily expanded to reduce the amount of redundant writing.

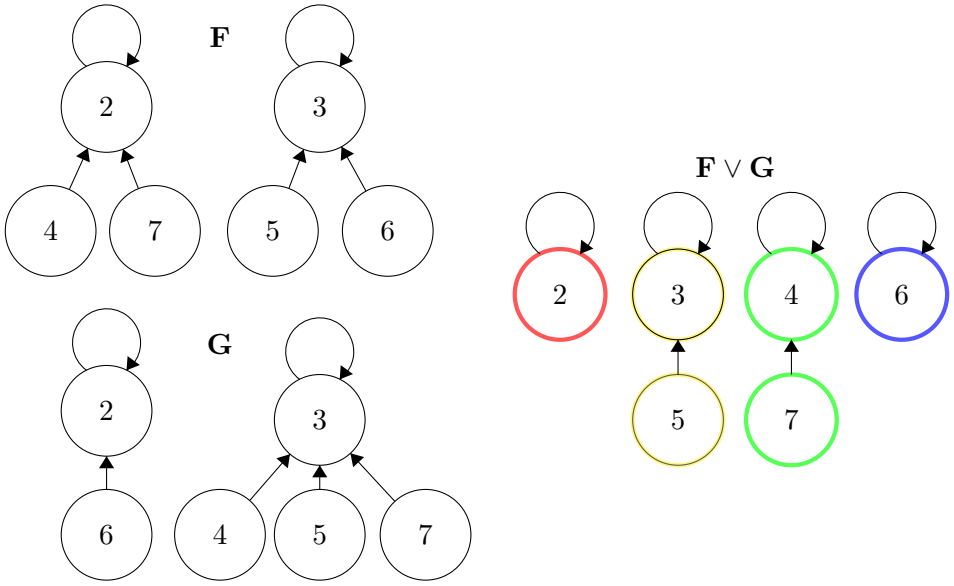
```
1: function EQUIV_AND(F, G)
2:     ALLOCATE_EQUIV(H)
3:     for x in Vars do
4:         H[x], H[F[x]], H[G[x]] ← min(H[F[x]], H[G[x]])
```

3.b.2 equiv_or

The logical OR of two equivalences is a bit more complicated, as we must only retain the variable equivalences found in both F and G. The complication arises in the fact that 2 variables can share an equivalence class in both F and G, but those classes need not have the same root variable. Therefore we cannot just intersect roots.

The below is a clear example of what should happen. 4 and 7 are linked to each-other in both Equivalences, however in F their root is 2, and in G it's 3.

x	2	3	4	5	6	7
$F[x]$	2	3	2	3	3	2
$G[x]$	2	3	3	3	2	3
$(F \vee G)[x]$	2	3	4	3	6	4



So how do we recognise this situation? Firstly we apply **flatten** to both F and G so we can see the roots directly. Now if we look each $(\mathbf{root_F}(x), \mathbf{root_G}(x))$ as an ordered pair, we can recognise the unique signature of each equivalence class resulting from the intersection of F and G. The job of the algorithm is to find the least variable for each signature, and write that as the root of all other variables with the same signature. This can be done in $O(n^2)$ time by looking back through the array every time to find the least variable with the same signature, but we can do better by using a hash table. For each x in $Vars$ (ascending), we check the hash table for a stored root-value at $(\mathbf{root_F}(x), \mathbf{root_G}(x))$, writing it at $H[x]$ if it exists. Otherwise this is the first variable with signature $(\mathbf{root_F}(x), \mathbf{root_G}(x))$, and must be the root of any later variables with the same signature. Therefore we store x at key $(\mathbf{root_F}(x), \mathbf{root_G}(x))$ in the hash-table and write x at $H[x]$.

```
1: function EQUIV_OR(F, G)
2:     ALLOCATE_EQUIV(H)
3:     INIT_HASH_TABLE(T)
4:     for x in Vars do
5:         if T[F[x]][G[x]] is valid variable then
6:             H[x] ← T[F[x]][G[x]]
7:         else
8:             H[x] ← x
9:             T[F[x]][G[x]] ← x
```

4 Canonicalisation

Before we can implement binary operations on (F, R) pairs, we need algorithms for canonicalisation to ensure that the invariant properties described in section 2.b hold.

We define two similar functions: **canon_ROBDD**(R) and **canon_wrt**(F, R).

4.a canon_ROBDD

This function takes any ROBDD, R, and returns a canonicalised pair (F, R'), such that $\mathcal{S}(F, R') \equiv \mathcal{S}_R(R)$. There are two stages for this function. We first extract all equivalent, entailed and disentailed variables from the ROBDD storing them in the new F component of the returned pair, then we minimise the ROBDD with respect to the equivalent variables in F, leaving only root variables in the ROBDD (if necessary).

```
1: function CANON_ROBDD(R)
2:     if IS_TERMINAL(R) then
3:         if R = 0 then return (FALSE_EQUIVALENCE, 0)
4:         else return (TRUE_EQUIVALENCE, 1)
5:     else
6:         F ← EXTRACT_EQUIVS(R)
7:         R' ← SELF_MINIMISE(F, R)
8:         return (F, R')
```

4.a.1 extract_equivs

As is natural for any ROBDD algorithm, we extract the Equivalences recursively. For terminal nodes, we return the appropriate trivial Equivalence. For non-terminal nodes, R, we check if either child node is the terminal **0**-node. This either entails or disentails R_{var}, so we recursively retrieve the equivalences found on the non-zero child-node and add the relevant entailment/disentailment of R_{var}. Otherwise we can neither say R_{var} is definitely true or definitely false. In this case, we intend to intersect the equivalences found on R_{then} and R_{else} (those variables that are equivalent regardless of R_{var}'s value). Not only this, we also need to recognise variables that are entailed when R_{var} is true and disentailed when R_{var} is false (these variables are logically equivalent to

R_{var}). Both of these jobs are handled by **equiv_then_or_else**$(R, then, else)$, which identical to **equiv_or**(F, G), except when the signature $(\mathbf{root_{then}}(x), \mathbf{root_{else}}(x))$ is $(\top, \bot)$, it writes R_{var}.

```
1:  function EXTRACT_EQUIVS(R)
2:      if IS_TERMINAL(R) then
3:          if R = 0 then return IDENTITY_EQUIVALENCE
4:          else if R = 1 then return FALSE_EQUIVALENCE
5:      else
6:          if R_else = 0 then
7:              result ← EXTRACT_EQUIVS(R_then)
8:              result.entail(R_var)
9:              return result
10:         else if R_then = 0 then
11:             result ← EXTRACT_EQUIVS(R_else)
12:             result.disentail(R_var)
13:             return result
14:         else
15:             equiv_then ← EXTRACT_EQUIVS(R_then)
16:             equiv_else ← EXTRACT_EQUIVS(R_else)
17:             return EQUIV_THEN_OR_ELSE(R, equiv_then, equiv_else)
```

4.a.2 self_minimise

After extracting the equivalences into the F-structure, we need to remove them from the ROBDD to enforce the invariant property that only F store this information. **minimise**(F, R) does this job recursively, returning the most "reduced" ROBDD, R', such that $\mathcal{S}(F, R) \equiv \mathcal{S}(F, R')$. For example, if $\mathcal{S}_R(R) \equiv w \wedge (x \leftrightarrow y) \wedge (y \vee z)$, then $F = \mathbf{extract\_equivs}(R)$ would be semantically $w \wedge x \leftrightarrow y$. After calling $R' = \mathbf{minimise}(F, R)$, we want R' to simply represent $x \vee z$. This is because, both w and $x \leftrightarrow y$ are already represented by F, and to keep it canonical, we need to have the ROBDD represent $x \vee z$ rather than $y \vee z$, as x is the least variable equivalent to y.

Since the equivalence information is still present in ROBDD, we can easily remove entailed, disentailed, or non-root-variables by restricting those variables to the appropriate boolean value. While executing, if the current node is non-terminal and either entailed or disentailed in F then one of it's children must be the zero-node, so we simply replace it with the non-zero child. We do the same thing when we find a node, R, where $R_{var} \neq F.\mathbf{find}(R_{var})$, as a choice must have been made for R_{var}'s root-variable along the path to this node, meaning one of children of R must be the zero-node. For all other non-terminal nodes, R, which are root-variables of their equivalence classes (possibly singleton sets), we simply minimise both children nodes, then reconstruct the node with **make_node** to preserve the reduced nature of the ROBDD.

Note this algorithm is equivalent to performing existential quantification over every non-root variable in the Equivalence (see the *projection* algorithms in [2]).

```
1: function SELF_MINIMISE(F, R)
2:     if IS_TERMINAL(R) then return R
3:     else
4:         root_var ← F.find(R_var)
5:
6:         if root_var ≠ R_var then
7:             if R_else = 0 then return SELF_MINIMISE(R_then)
8:             else return SELF_MINIMISE(R_else)
9:
10:        else
11:            return make_node(R_var, MINIMISE_AUX(F, R_then), , MINIMISE_AUX(F, R_else))
```

```
1: function MINIMISE_WRT(F, R)
2:     entailed[] ← INIT_ZEROES( )
3:     disentailed[] ← INIT_ZEROES( )
4:     return MINIMISE_AUX(F, R, entailed, disentailed)
5:
6: function MINIMISE_AUX(F, R, entailed[], disentailed[])
7:     if IS_TERMINAL(R) then return R
8:     else
9:         root_var ← F.find(R_var)
10:
11:        if root_var = ⊤ or entailed[root_var] = true then
12:            return MINIMISE_AUX(F, R_then, entailed, disentailed)
13:
14:        else if root_var = ⊥ or disentailed[root_var] = true then
15:            return MINIMISE_AUX(F, R_else, entailed, disentailed)
16:        else
17:
18:            entailed[root_var] = true
19:            new_then ← MINIMISE_AUX(F, R_then, entailed, disentailed)
20:            entailed[root_var] = false
21:
22:            disentailed[root_var] = true
23:            new_else ← MINIMISE_AUX(F, R_else, entailed, disentailed)
24:            entailed[root_var] = false
25:
26:            return MAKE_NODE(R_var, new_then, new_else)
```

References

[1] R. Bagnara and P. Schachte. Efficient Implementations of *Pos*

[2] R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of *Pos*.