## how to build

このディレクトリで `make` すると

```
findsplit
gsconv
gsconv3
```

の 3 つのプログラムが作成されるはずである．上 2 つは今までのとだいたい同じ．最後の `gsconv3` が新しいコンバーター．現在は tango 形式を得るのに古い方のプログラムを活用するので, 全部のプログラムが必要. `gsconv3` には 2 つの機能がある．

1 つは後述する `csdl` で記述された方式を `gsconv` の入力形式に 変換する機能である．`shell` で

```
./gsconv3 < test1.csdl
```

などと入力すると標準出力に `gsconv` 形式の方式が出力される．出力を適当に `gsconv` および `findsplit` に入力すると最終的に `findsplit` が `dot` ファイル (`G.dot`) を出力する．

もう 1 つは `dot` ファイルから分割情報を取り出し, 元の方式を非対称ペアリングの方式に再構成する機能である．`shell` で

```
./gsconv3 G.dot < test1.csdl
```

などと入力すると標準出力に再構成された方式が出力される．これらの一連の流れを実行するスクリプト `conv.sh` が用意されている．

```
./conv.sh test1
```

などと入力 (拡張子は入力しない) するとエラーが無ければ 最終結果 (`./findsplit --ratio=4` の場合) が `test1.result` なるテキストファイルに出力される．

## C-like Scheme Description language (csdl)

C-like Scheme Description language (csdl) はペアリングに基づく暗号方式のデータフロー解析を目的としたドメイン特化言語で, その文法はプリプロセッサの無い C 言語 (C89) と概ね同じである．(形式的文法はほとんど一緒).

csdl のコメントは C++ 言語のコメントとだいたい同じで, `/*` で始まり `*/` で終わるか, `//` で始まり次の改行で終わるかのどち らかである．

csdl と C 言語の形式的文法上の最も大きな違いは ^ 演算子の解釈 である．^ 演算子は C 言語では bitwise xor 演算子を表現し算術 演算より弱い優先度を持つ．一方 csdl では ^ 演算子は TeX のよう にべき乗を表現

し, どの算術演算より強い優先度を持ち右結合する. xor 演算を記述する場合は `xor` または `(+)` と書く. (xor 代入 演算の場合は `xor_eq` または `(+)=`).

csdl ではアルゴリズムは関数の中に記述するか

```
main(){
    ...
}
```

または, 単に複文 (compound statement) の中に記述する.

```
{
    ...
}
```

関数名は `main` でなくても良い. 同名でなければいくつ定義しても良い. 複文もいくつあっても良い. 例えば:

```
setup(){ ... }
sign(){ ... }
verify(){ ... }
// correctness
{ ... }
// reduction
{ ... }
```

方式を構成するアルゴリズムは関数として記述し, 正当性 (correctness) や完全性 (completeness) といった方式の形式的文法 (formal syntax), あるいは安全性帰着のデータフローなどは複文中に記述するのが良かろう. 方式を実際のプログラミング言語で実装する時は生の複文中の記述を単に無視すればよい.

### データ型

C の組み込みデータ型の他に, csdl は次のデータ型を持つ.

| 型名 | 意味 |
|---|---|
| `group` | ペアリングのソース群 |
| `group0` | ペアリングのソース群 0 |
| `group1` | ペアリングのソース群 1 |
| `prohibited` | ペアリングのソース群 (二重化禁止ノード) |
| `target` | ペアリングの標的群 |
| `integer` | 整数 (上記群の自己準同型環 (の可換部分環)) |
| `list` | リストデータ型 |

変数の宣言は次のように記述する. (宣言文).

```
group   A,B,C,X,Y,Z,f,g,h ;
integer alpha, beta, gamma, delta ;
target  gT ;
```

変数は宣言時に初期化する事も出来る (初期化文). 例えば

```
group W = X*Y*Z ;
```

宣言文や初期化文は C++ や C99 のように, スコープ内で同名の変数が既に宣言されていない限りは複文中のどの位置でも可能.

csdl では宣言された変数には, その変数名に似た名前の Object-ID が与えられる. 例えば csdl にて

```
integer a, b, c, d ;
```

という変数宣言を行うと, 大体次のような Object-ID が与えられる.

```
integer $a$0, $b$0, $c$0, $d$0 ;
```

基本的に変数にはその Object-ID が格納され, csdl が解釈されている間に Object-ID が変更される事は無い. (どちらかというと変数というより定数). csdl では異なる変数には必ず異なる Object-ID が割り当てられる. 初期化文でもこの原則は変わらない.

```
integer a = b ;
```

と記述したからと言って a の Object-ID が b の Object-ID に設定される事は無い. C 言語で上記のように記述した時, 形式的には a のアドレスが b のアドレスになる訳では無いのと同様に考えれば良い. スコープの異なる同じ名前の変数でも同様. 例えば

```
integer a ;
{
    integer a ;
}
```

と記述すると

```
integer $a$0 ;
{
    integer $a$1 ;
}
```

等と解釈される. csdl において変数はスコープから外れるとアクセス出来ないが, 一度生成された Object-ID 自体は csdl が解釈されている間ずっと生き残る.

csdl では一部の例外を除き, 一般に変数が式の中で評価される時は必ずその Object-ID として評価される. 例えば group 変数 Y は$Y$0 のよう な評価値を持つ. そして Object-ID の代数式はシステムの代数式エンジンが評価できる範囲内でのみ評価される. 例えば

```
Y^alpha * Y^beta
```

という式は最終的に

```
$Y$0^($alpha$0 + $beta$0)
```

と評価される. 特に環 integer においては, 割り算やべき乗は必ずしも 評価できるとは限らない. そのような場合には新しい変数が生成されるか, またはエラーになるかまたはバグる (基本的に integer 同士の割り算は除数が単項式の時以外は信用しない方が良い. また integer 同士のべき 乗は冪が小さい整数である時か, 底が単項式で冪が整数の時以外は信用しない 方が良い).

変数に値を割り当てるには, 次のように記述する (代入文).

```
X = Y^alpha * Z^beta ;
```

group, target, integer の変数を使って代入文を記述すると, 右辺を Object-ID の代数式として可能な範囲で評価し, Object-ID により指定される Object 間にそのような代数的関係及びデータフローが存在すると解釈される.

```
$X$0 <- $Y$0^$alpha$0 * $Z$0^$beta$0 ;
```

左辺はいわゆる左辺値である必要がある. 例えば

```
X^2 = Y^alpha * Z^beta ;
```

のような代入は出来ない.

### リスト値とリスト変数

csdl と C 言語の間には形式的文法上の違いはほとんど無いが, 意味論上は大きな違いがある. この意味論上の違いの為, 両者は全く異なる言語のように感じられるかもしれない. 特に,(コンマ) 演算子の解釈の仕方には大きな違いがある. ,(コンマ) 演算子は C 言語では左から右に評価され, 左の式の値は捨てられる. 一方 csdl では左の式の値は捨てられない.

```
a, b, c
```

はリスト値 ($a$0,$b$0,$c$0) として評価される. ,(コンマ) 演算子の優先度は C 言語と同様に最も低く設定されているので, リスト値に何らかの操作を行う場合は

```
(a, b, c)
```

のようにカッコで括る. リスト値の要素が全て左辺値なら, そのリスト値は左辺値 となる. 即ち代入出来る.

```
integer a, b, c, d, e, f ;
(a,b,c) = (d^2, e^3*f, e*d/f) ;
```

リストは入れ子にする事が出来る. (代入式で入れ子が合ってないとバグる).

```
integer a, b, c, d, f ;
((a,b),c) = ((d^2, d^3*f), d/f) ;
```

リスト値の要素には [] 演算子でアクセスする事が出来る. 例えば

```
(a,b,c)[2]
```

は $c$0 等と評価される. しかし,

```
integer i = 2 ;
(a,b,c)[i] ;
```

などという事は出来ない. (i は単なる Object-ID であり, 数では無い). またリスト 値の要素に左辺値がある場合.(ドット) 演算子を用いて Object-ID の一致する要素にアクセスする事も出来る.

```
(a,b,c).b
```

は $b$0 等と評価される.

list 型の変数 (リスト変数) にはリスト値を代入することが出来る.

```
integer a, b, c ;
list L = (a, b, c) ;
```

リスト変数が評価される時は, 出来るだけ最後に代入されたリスト変数以外の Object-ID の式に還元しようと努力する. 例えば

```
L
```

は

```
($a$0, $b$0, $c$0)
```

と評価される. リスト変数にリスト値が代入された場合, その要素 には L[0], L[1] や L.c などとしてアクセスする事が出来る.

### 等式および連言の評価

group, target 型の等式 left == right は代数式 left/right と評価される. integer 型の等式 left == right は代数式 left - right と評価される. それぞれ 乗法または加法の単位元になる時, 等号が成立すると解釈する. 連言の論理式

```
left1 == right1 && left2 == right2
```

は連言リスト (リスト値と概ね同じデータ構造だが実装上の都合により幾分扱いが異なる)

```
( left1/right1 && left2 - right2 )
```

として評価される. 等式の代数式による表現が大量にある時, 比較的簡単にバッチ式を構成 できる (と思う).

### 関数

関数は C 言語っぽく定義する. 返り値を指定しない場合は list と解釈される. 関数の頭に macro と記述すると呼び出す度にインライン展開される.

```
setup(){
  group   g   ;
  integer msk ;
  group   y   = g^msk ;
  list    pp  = (g,y) ;
  return  (pp,msk) ;
}

keygen(group ID, integer msk){
  group  d_ID = ID^msk ;
  return d_ID ;
}

enc(target m, group ID, list pp){
  group g, y ;
  (g,y) = pp ;
  integer r ;
  group   c1 = g^r ;
  target  c2 = m*e(ID,y^r)^2 ;
  return (c1,c2) ;
}

dec(list c, group d_ID){
```

```
  group c1; target c2 ;
  (c1,c2) = c ;
  target m = c2/e(c1,d_ID)^2 ;
  return m ;
}

{
correctness:
  group ID ;
  target m ;

  list pp ; integer msk ;
  (pp,msk) = setup() ;
  group c1; target c2 ;
  (c1,c2) = enc(m,ID,pp) ;

  m == dec((c1,c2), keygen(ID, msk)) ;

proof: ;

  integer a, b, c ;
  group g, y ;
  (g,y) = pp ;
  group A = g^a ;
  group B = g^b ;
  group C = g^c ;

  y  = A ;
  ID = B ;
  c1 = C ;
}
```

特殊形式/定義済みマクロ

以下の特殊形式/定義済みマクロが存在する. 同じ名前の変数定義や関数宣言をしてしまうと, そのスコープでは使用出来なくなるので注意が必要である. ついうっかり変数 `e` を宣言してしまう事が良くある.

| 関数 | 機能 |
|---|---|
| `target e (group g0, group g1)` | pairing |
| `list GS_setupB (group g)` | Groth-Sahai setup (binding) |
| `list GS_setupH (group g)` | Groth-Sahai setup (hiding) |
| `list GS_proofwi(list crs, list predicate)` | Groth-Sahai proof (WI) |

| 関数 | 機能 |
|---|---|
| `list GS_proofzk(list crs, list predicate)` | Groth-Sahai proof (ZK) |
| `list GS_verifywi(list crs, list predicate list proof)` | Groth-Sahai verify (WI) |
| `list GS_verifyzk(list crs, list predicate list proof)` | Groth-Sahai verify (ZK) |
| `list statistics (list x)` | report statistical information |
| `list setweight(integer n, list group_variables)` | set weight of `group_variables` to `n` |
| `list setpriority(integer n, list group_variables)` | set priority of `group_variables` to `n` |
| `list exclude (list X, list Y)` | $X_i \neq Y_j$ for all group elements in `X`,`Y` |
| `list PI (list x)` | retrieve raw proof string from a list |
| `list COM (list x)` | retrieve commitments from a list |

Groth-Sahai 証明系は 例えば以下のように使う.

```
group A, B, X, Y ;
list pair = GS_setupB(g) ;

Proof(){
    return GS_proofzk(pair.crs, e(A,[X]) * e(B,[Y]) == 1 ) ;
}

Verify(list proof){
    return GS_verifyzk(pair.crs, e(A,[X]) * e(B,[Y]) == 1, proof ) ;
}

{
    Verify(Proof()) ;
}
```

`GS_proofzk()` 内の `[X]` は直前に定義されている `group` 変数 `X` の `crs` に関するコミットを表現している. 整数型のコミットも同様. 定義されていないコミットが使用されると自動定義されるので, `GS_proofzk()` は `GS_verifyzk()` の前に置くのが自然であろう.

現状の `csdl` には, `crs` 内変数, 係数変数, 術語変数およびその コミットの Object-ID を `GS_proofzk()` と `GS_verifyzk()` の間で 自然に共有する為の構文は特に存在しない. それらの Object-ID を共有 する為には, 上記の `Proof()` 関数と `Verify()` 関数の間で共有されるスコープ (即ちグローバルスコープ) に, これらの変数を定義する必要がある.

Object-ID の共有を諦め, データフローを自前で定義する場合は, 例えば次のように記述する.

```
Proof(list crs){
    group A, B, X, Y ;
    return GS_proofzk(crs, e(A,[X]) * e(B,[Y]) == 1 ) ;
```

```
}

Verify(list crs, list proof){
    group A, B, X, Y ;
    return GS_verifyzk(crs, e(A,[X]) * e(B,[Y]) == 1, proof ) ;
}

{
    group g ;
    list pair = GS_setupB(g) ;
    Verify(pair.crs, Proof(pair.crs)) ;
}
```

このとき, Proof() 関数と Verify() 関数で宣言されている変数は それぞれ何の関係もない Object-ID を持つ. 下の方で定義された 方式の syntax によって, それらの Object-ID 間のデータフロー が定義される. GS_verifyzk() 関数は元々存在しない proof や crs から検証式を生成する為, 仮想的な proof や crs を無理やり 導出する. 従って, 依存関係グラフ中に余分なノードが大量に生成される. これらのノードは重み 0 が設定されない限り最適化に影響を与える可能性がある.

術語内で使用する変数の対応は必ずしも同じ名前である必要は無いが, 実装上の都合により, 少なくとも対応する変数の (Object-ID の) 辞書式順序が一致する必要がある. GS_verifyzk() が自動生成する仮想変数の 中には本来秘匿されるべき秘密も宣言される. これは情報が漏洩してい る訳ではなく, コミットを媒介とした仮想的データフローが記述されて いるに過ぎない.

GS_proofzk() では術語変数だけではなく係数変数のコミットも勝手に 作られる可能性がある. 明示的にコミットを生成するには,

```
crs.[X] ;
```

などと書く. GS_proofzk() が勝手にコミットを生成する場合は既にコミットされている変数が優先的に選ばれる.

従って, Proof() 内でコミット優先順位に変更を与えるような文を記述した場合は同じコミットを Verify() の中でも実行しないと正しい GS_proofzk() と GS_verifyzk() のペアが生成出来ない 事になる. これはかなり奇妙な制約であるから, 素直にグローバル スコープを使って Object-ID を共有した方が良いかもしれない.

等号は C 言語と同じく, = ではなくて == と書く. 零知識証明ではなく証拠識別不能証明の場合は GS_proofwi() および GS_verifywi() を使う. 連言命題は GS_proofzk() を列挙するか, または連言を次のように記述する.

```
list proof = GS_proofzk( crs1,
    alpha * [beta]                    == 1          &&
    alpha * [beta] + [gamma] * delta == 1          &&
    [alpha] * [beta]                  == 1          &&
```

```
    [g]^alpha * [h]^beta             == 1          &&
    g^[alpha] * h^[beta]             == f^[gamma] &&
    [g]^[alpha] * h^[beta]           == [f]^gamma &&
    e(A,[X]) * e(B,[Y])              == 1          &&
    e(A,[X]) * e([B],[Y])            == 1
) ;
```

証明文字列 proof を検証するには

```
GS_verifyzk(crs,
    alpha * [beta]                   == 1          &&
    alpha * [beta] + [gamma] * delta == 1          &&
    [alpha] * [beta]                 == 1          &&
    [g]^alpha * [h]^beta             == 1          &&
    g^[alpha] * h^[beta]             == f^[gamma] &&
    [g]^[alpha] * h^[beta]           == [f]^gamma &&
    e(A,[X]) * e(B,[Y])              == 1          &&
    e(A,[X]) * e([B],[Y])            == 1
proof) ;
```

と書く. GS_verifyzk() は等式の代数式表現のリスト (正確には代数式表現を葉にもつ木) を返す. 連言リストではなく単なる リストである. 比較的簡単にバッチ式を構成できる (と思う).