Translation from C to logical formulas

Paul RAYNAUD M1 Informatique

Laboratary: VERIMAG

Team : PACSS

Tutors :

Michaël PÉRIN; David MONNIAUX

Magistère M1 presentation

Motivation

- Computers omnipresence
  - Computers are everywhere in our society to help us in our daily life.
  - However bugs happen really often.

- In the critical systems

  - The Health, the aviation, energy, information, finance

C → logical formulas

- Why?
  - We can automatically generate Hoare's triplet from source code
  - We can solve this formulas with a Solver Module Theorie (SMT), to check if the generated implications are valid or not

- How?
  - By choosing a compiler/semantic.
  - We need to get a Control Flow Graph (CFG) .

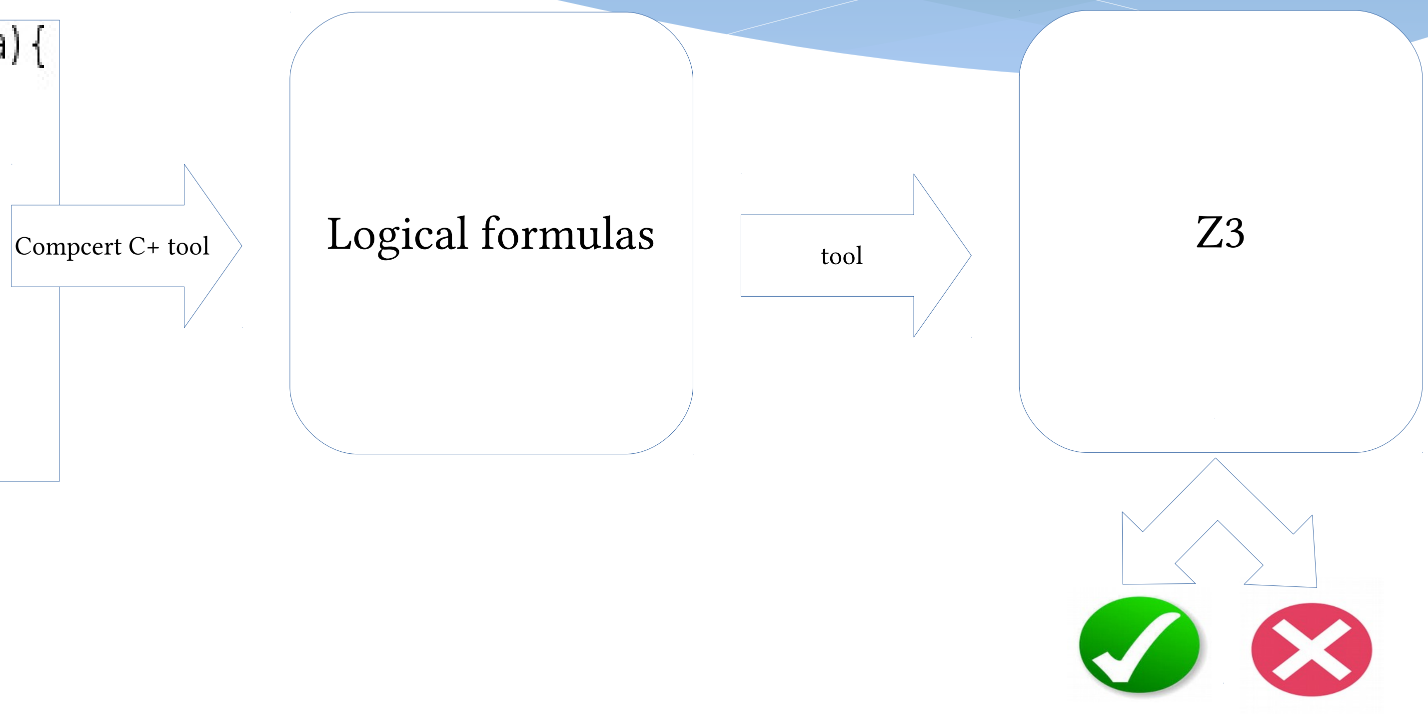
Existing tools for program verification

- Seahorn is a tool performing program verification.
  - Based on LLVM compiler, with an ambiguous semantic, and with a non-alterable logical translation

- The choice of a compiler : CompCert C
  - A well-defined semantic (in Coq).
  - Without any miscompilation.
  - Thus a robust foundation to prove the translation into logical formulas.
- Z3
  - The most general powerfull SMT solver
  - Use in a successful similar project

Goal

```
int absolute_value(int a){
  if( a > 0){
    return a;
  }
  else {
    return -a ;
  }
}
```



Detail our example

```
int absolute_value(int a){
  if( a > 0){
    return a;
  }
  else {
    return -a ;
  }
}
```

- Take an integer
- Return the absolute value.
- Thus the value is postive or null.

From C to RTL

```
 4:  (RTL.Icond ((Op.Ccompimm (Integers.Cgt, BinNums.Z0)), [BinNums.Coq_xH],
(BinNums.Coq_x0 BinNums.Coq_xH), (BinNums.Coq_xI BinNums.Coq_xH)))
 3:  (RTL.Iop (Op.Oneg, [BinNums.Coq_xH], (BinNums.Coq_x0 BinNums.Coq_xH),
BinNums.Coq_xH))
 2:  (RTL.Iop (Op.Omove, [BinNums.Coq_xH], (BinNums.Coq_x0 BinNums.Coq_xH),
BinNums.Coq_xH))
 1:  (RTL.Ireturn (Some (BinNums.Coq_x0 BinNums.Coq_xH)))
```

Preparation :

- 13 intermediary representations.
- the 4th representation, the Register Transfert Language (RTL)
- Extract this fourth representation in a usable form.

From C to an readable RTL

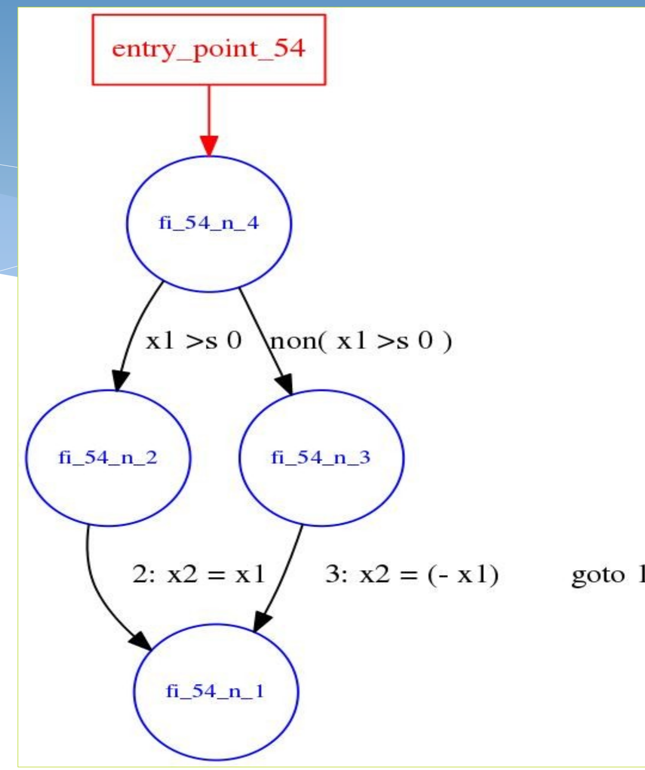
```
 4:  (RTL.Icond ((Op.Ccompimm (Integers.Cgt, BinNums.Z0)), [BinNums.Coq_xH],
(BinNums.Coq_x0 BinNums.Coq_xH), (BinNums.Coq_xI BinNums.Coq_xH)))
 3:  (RTL.Iop (Op.Oneg, [BinNums.Coq_xH], (BinNums.Coq_x0 BinNums.Coq_xH),
BinNums.Coq_xH))
 2:  (RTL.Iop (Op.Omove, [BinNums.Coq_xH], (BinNums.Coq_x0 BinNums.Coq_xH),
BinNums.Coq_xH))
 1:  (RTL.Ireturn (Some (BinNums.Coq_x0 BinNums.Coq_xH)))
```

Compcert print the data above in a readeable form like below

```
absolute_value(x1) {
    4:  if (x1 >s 0) goto 2 else goto 3
    3:  x2 = (- x1)
        goto 1
    2:  x2 = x1
    1:  return x2
}
```

From RTL to CFG

```
 4:  (RTL.Icond ((Op.Ccompimm (Integers.Cgt, BinNums.Z0)), [BinNums.Coq_xH],
(BinNums.Coq_x0 BinNums.Coq_xH), (BinNums.Coq_xI BinNums.Coq_xH)))
 3:  (RTL.Iop (Op.Oneg, [BinNums.Coq_xH], (BinNums.Coq_x0 BinNums.Coq_xH),
BinNums.Coq_xH))
 2:  (RTL.Iop (Op.Omove, [BinNums.Coq_xH], (BinNums.Coq_x0 BinNums.Coq_xH),
BinNums.Coq_xH))
 1:  (RTL.Ireturn (Some (BinNums.Coq_x0 BinNums.Coq_xH)))
```



Making the first structure:

- We refine the RTL structure to keep only the necessary informations, and simplify the CFG.

- Development of a tool to print easily our CFGs.

Reminder about Hoare logic

Hoare triplet : $\{\rho\}program\{\gamma\}$

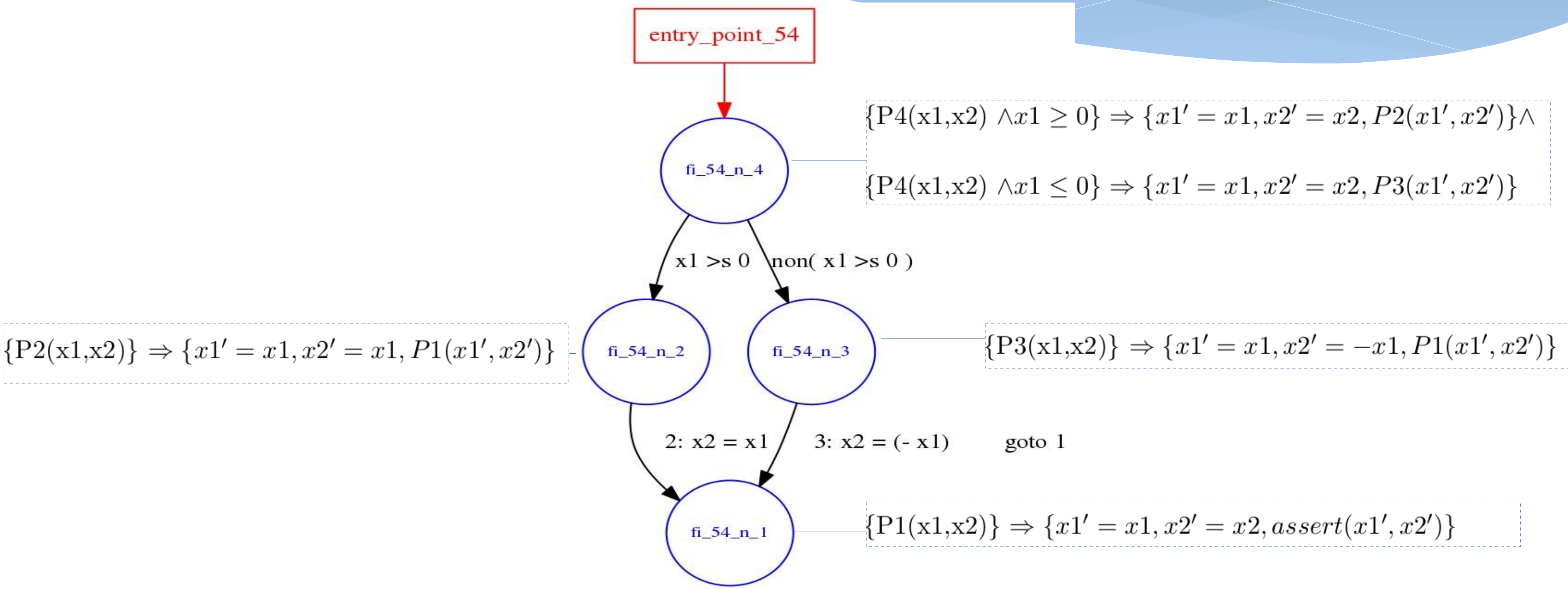
The sequence rule : $$\frac{\{\rho\}\ S1\ \{\theta\}\ \{\theta\}\ S2\ \{\gamma\}}{\{\rho\}\ S1;\ S2\ \{\gamma\}}$$

The affectation rule : $$\overline{\{\rho\}x := e\{x' := e \wedge \gamma[x \leftarrow x']\}}$$

The condition rule : $$\frac{\{\rho \wedge C\}S1\{\gamma\} \wedge \{\rho \wedge \neg C\}S2\{\gamma\}}{\{\rho\}if\ C\ then\ S1\ else\ S2\{\gamma\}}$$

The skip rule (return) : $$\overline{\{\rho\}skip\{\gamma\}}$$

Application on our example



Introduction to Z3

- Z3
  - Z3 is a typed language

- Problem :
  - RTL is'nt typed
  - We have to retrieve the type of each variable to define the predicates in Z3

Retrieve the variable types from the operations

```
x = 5;
```

x is an integer

```
if(x){
        y = 3;
}
```

x is an boolean

y is an integer

```
if(x){
        x = x+1;
}
```

x is typed like a boolean and a integer so we choose integer

- Retrieve the type by analizing the operation .
- But in order to help Z3 we get the minimum type of each variable.

Z3 formulas

```
(forall ( (x_54_2_2 Int  ) (x_54_2_1 Int  ) (x_54_4_2 Int  ) (x_54_4_1 Int  ) (x_54_3_2 Int  ) (x_54_3_1 Int  ) )
  ( and
    ( =>
      ( and
        (P_54_4 x_54_4_1 x_54_4_2 )
        ( =  x_54_2_2  x_54_4_2   )
        ( =  x_54_2_1  x_54_4_1   )
        ( > x_54_4_1  0 )
      )
      (P_54_2 x_54_2_1 x_54_2_2 )
    )
    ( =>
      ( and
        (P_54_4 x_54_4_1 x_54_4_2 )
        ( =  x_54_3_2  x_54_4_2   )
        ( =  x_54_3_1  x_54_4_1   )
        (not( > x_54_4_1  0 ) )
      )
     (P_54_3 x_54_3_1 x_54_3_2 )
    )
  )
)
(forall ( (x_54_3_2 Int  ) (x_54_3_1 Int  ) (x_54_1_2 Int  ) (x_54_1_1 Int  ) )
  ( =>
    ( and
      (P_54_3 x_54_3_1 x_54_3_2 )
      ( =  x_54_1_1  x_54_3_1   )
      ( =  x_54_1_2 (- x_54_3_1) )
    )
      (P_54_1 x_54_1_1 x_54_1_2 )
  )
)
```

```
(forall ( (x_54_2_2 Int  ) (x_54_2_1 Int  ) (x_54_1_2 Int  ) (x_54_1_1 Int  ) )
  ( =>
    ( and
      (P_54_2 x_54_2_1 x_54_2_2 )
      ( =  x_54_1_1  x_54_2_1   )
      ( =  x_54_1_2  x_54_2_1   )
    )
      (P_54_1 x_54_1_1 x_54_1_2 )
  )
)
(forall ( (x_54_1_2 Int  ) (x_54_1_1 Int  ) )
  (=>
    (P_54_1 x_54_1_1 x_54_1_2 )
    (post_54 x_54_1_1 x_54_1_2 )
  )
```

Adding of Pre-cond/ Post cond and Z3 formulas

Defintion of assumption and assertion

```
(define-fun assume_54 ((x1 Int)(x2 Int)) Bool
   true
)
(define-fun assert_54 ((x1 Int)(x2 Int)) Bool
   (<= 0 x2)
)
```

The link between the assumption and the program

```
(forall ((x1 Int) (x2 Int))
  (=>
    (assume_54 x1 x2)
    (P_54_4 x1 x2 )
  )
)
```

The link between the assertion and the program

```
(forall ( (x_54_1_2 Int  ) (x_54_1_1 Int  ) )
  (=>
    (P_54_1 x_54_1_1 x_54_1_2 )
    (assert_54 x_54_1_1 x_54_1_2 )
  )
)
```

- Link between the program and the properties
  - Adding the assumptions/assertions of the program.
  - To prove our program the assertions/assumptions has to be true.

Detail about our results

- Expected output of Z3
  - Sat : Z3 has find one invariant for each node of the CFG which respect the assume and the assert.
  - Unsat : Z3 shows that you can't have one invariant for each node of the CFG which can't respect the assume and the assert.
- Unexpected output
  - Time-out : Z3 don't have the time, memory or something else to compute the answer
  - Unknown : Z3 can't find an answer with the given information

Note about the results

- Other strategys
  - Timeout : restrain de starting space to get a local answer.
- To improve:
  - Unsat : currently we are not able to get some valuable informations about Z3 when he make an unsat.

May RTL be the good representation for a verifier ?

- Good points:
  - CFG
  - No register strategy, 1 variable in C mean 1 variable in RTL with added variables
  - Many informations avalaible, to find the lost informations
- Bad point:
  - Informations are not always directly accessible

Future extensions

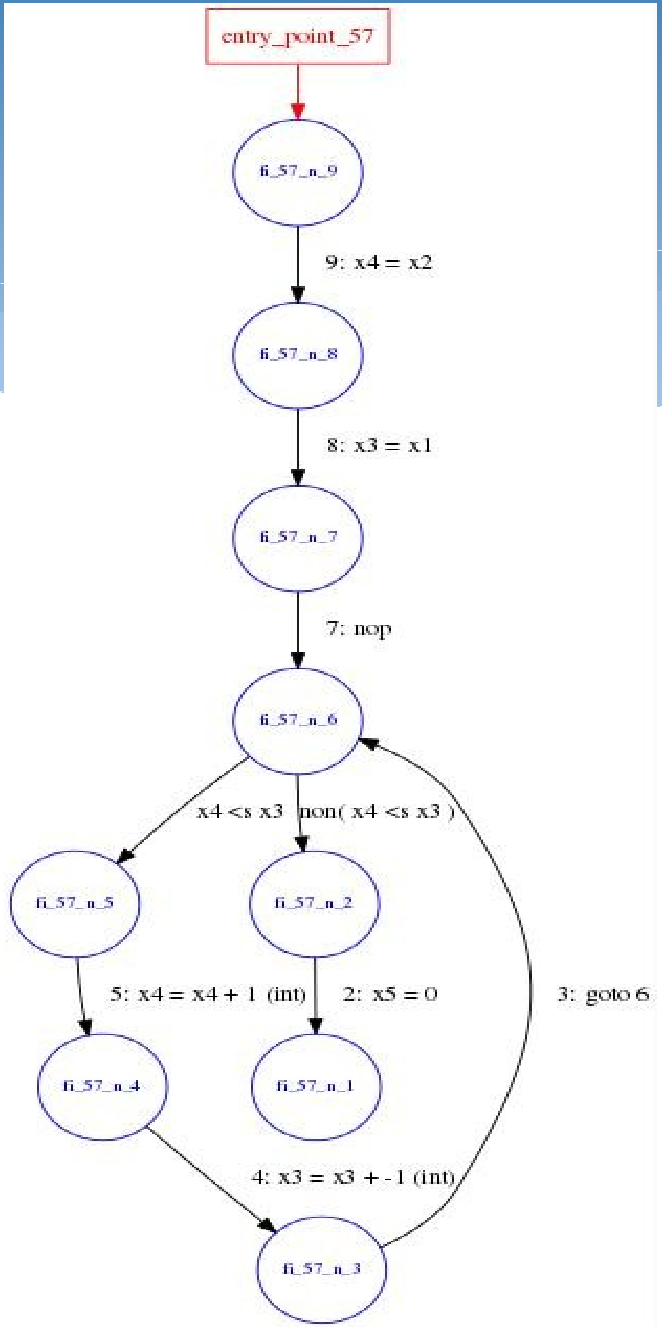
- Other features
  - Add propriety for particular instructions like the division to forbid the division by 0.
  - Replace the assertions/assumptions by checking directly the properties inside the code
  - In order to help it, give to Z3 some properties and don't let it calculate
- Make the tool usable
  - We have to finish all the RTL instruction define by CompCert C firstly.
  - Prove the tool using Coq. La generation de formules logiques

Questions

Less trivial example (1)

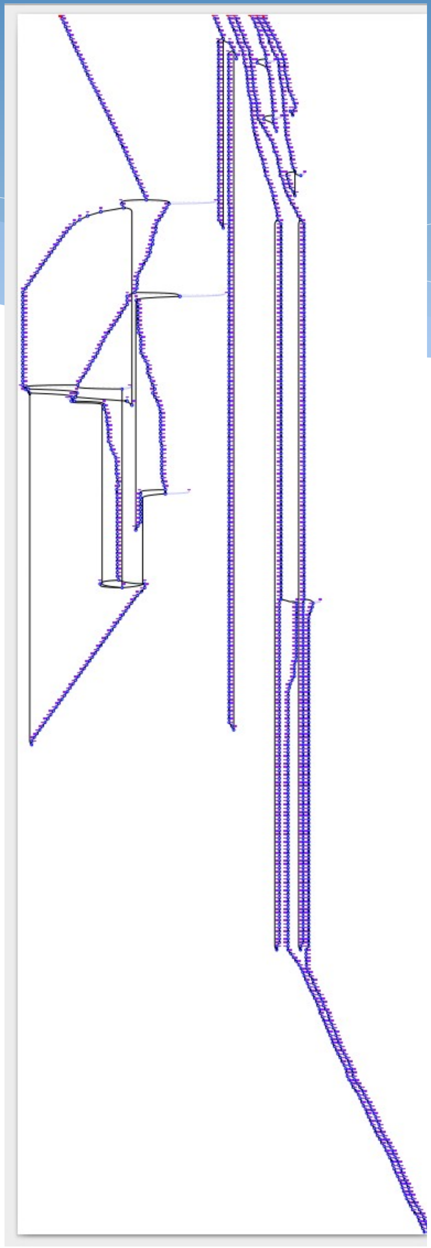
```c
int milieu(int a , int b){
        int x = a;
        int y = b;
        while(x < y){
                x = x + 1;
                y = y - 1;
        }

        return 0;
}
```



Less trivial example (2)

1500 lines program



Compcert stuff (operation)

```
type instruction =
| Inop of node
| Iop of operation * reg list * reg * node
| Iload of memory_chunk * addressing * reg list * reg * node
| Istore of memory_chunk * addressing * reg list * reg * node
| Icall of signature * (reg, ident) sum * reg list * reg * node
| Itailcall of signature * (reg, ident) sum * reg list
| Ibuiltin of external_function * reg builtin_arg list * reg builtin_res
   * node
| Icond of condition * reg list * node * node
| Ijumptable of reg * node list
| Ireturn of reg option
```

Compcert stuff (operation)

```
type operation =
  Omove
  Ointconst of Int.int
  Olongconst of Int64.int
  Ofloatconst of Int64.int
  Osingleconst of float32
  Oindirectsymbol of ident
  Ocast8signed
  Ocast8unsigned
  Ocast16signed
  Ocast16unsigned
  Oneg
  Osub
  Omul
  Omulimm of Int.int
  Omulhs
  Omulhu
  Odiv
  Odivu
  Omod
  Omodu
  Oand
  Oandimm of Int.int
  Oor
  Oorimm of Int.int
  Oxor
  Oxorimm of Int.int
  Onot
  Oshl
  Oshlimm of Int.int
  Oshr
  Oshrimm of Int.int
  Oshrximm of Int.int
  Oshru
  Oshruimm of Int.int
  Ororimm of Int.int
  Oshldimm of Int.int
  Olea of addressing
  Omakelong
  Olowlong
  Ohighlong
  Ocast32signed
  Ocast32unsigned
  Onegl
  Oaddlimm of Int64.int
  Osubl
  Omull
  Omullimm of Int64.int
  Omullhs
  Omullhu
  Odivl
```

```
Odivlu
Omodl
Omodlu
Oandl
Oandlimm of Int64.int
Oorl
Oorlimm of Int64.int
Oxorl
Oxorlimm of Int64.int
Onotl
Oshll
Oshllimm of Int.int
Oshrl
Oshrlimm of Int.int
Oshrxlimm of Int.int
Oshrlu
Oshrluimm of Int.int
Ororlimm of Int.int
Oleal of addressing
Onegf
Oabsf
Oaddf
Osubf
Omulf
Odivf
Onegfs
Oabsfs
Oaddfs
Osubfs
Omulfs
Odivfs
Osingleoffloat
Ofloatofsingle
Ointoffloat
Ofloatofint
Ointofsingle
Osingleofint
Olongoffloat
Ofloatoflong
Olongofsingle
Osingleoflong
Ocmp of condition
```