Dealing with arithmetic overflows in the polyhedral model

Bruno Cuervo Parrino Julien Narboux Eric Violard
Nicolas Magaud

Université de Strasbourg - INRIA Camus

IMPACT 2012, Paris

Outlines

1. Context
   - Certifying compilation
2. The problem of arithmetic overflows
   - Description of the problem
   - Solution within Polly
   - Solution proposed
3. Toward a formalization within Coq

Warning

Work in progress : proofs are not completed yet.

Motivations

- It is useless to prove a program if the compiler contains bugs.
- Compilers are more and more complex.
- Parallelizing and speculative compilers are particularly complex.

Proof assistants

- Provide a language to describe algorithms, and mathematical statements.
- Provide a way to buid proof interactively.
- Check the correctness of the proof.

Some proof assistants

- Coq
- Isabelle
- PVS
- ...

Compcert (Blazy, Leroy, Tristan)

A compiler with a machine checked proof that:
"For all correct source programs S, if the compiler terminates without error and produces executable code C, then S and C are observationally equivalent."

Project

Goal

Integrating the polytop model into CompCert (Blazy, Leroy, Tristan)

in order to:

In the medium term improve (spatial and temporal) data locality

In the long term exhibit parallelism

Project in collaboration with Alexandre Pilkiewicz - INRIA Gallium.

Implementations

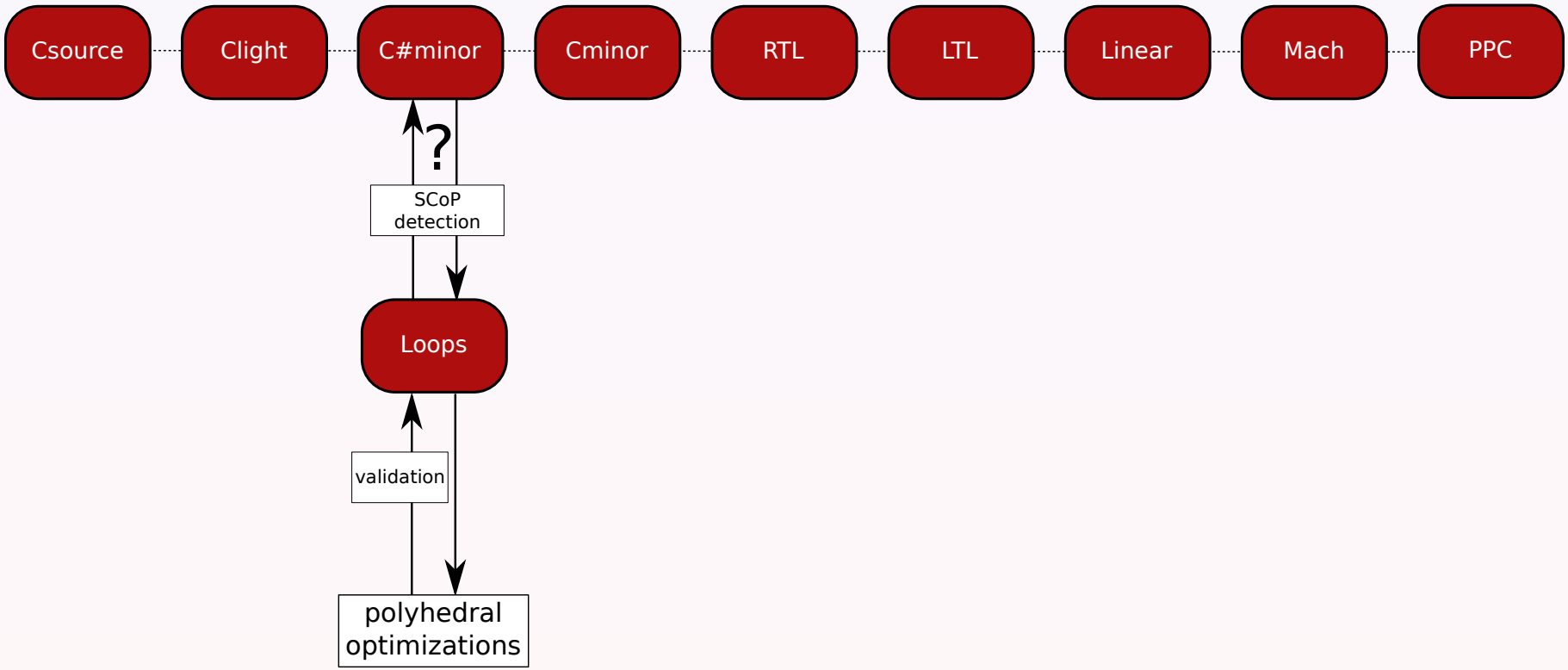
- In LLVM: Polly by Tobias Grosser (2010-2011)
- In GCC: Graphite by Konrad Trifunovic and al. (2009-2011)

Project: CompCert + Loops

Approach based on a posteriori validation (Zuck, Pnueli and al - 2002) proved in Coq (Leroy, Tristan - 2008).

- This enables to reuse existing tools.
- This is enough to obtain the same correctness guarantee as with a formally proved compilation pass.
- But this does not guarantee that the compiler will provide a result.

Architecture



Problem

The main advantage of the polyhedral model is to provide a simple mathematical representation for program SCoP.
The drawback is that this mathematical representation is different from the real machine implementation.

Arithmetic overflows

1. The original code may produce arithmetic overflows[a].
2. The optimized code may produce arithmetic overflows as well.

[a] Here we only consider arithmetic overflow during loop bounds and array access computations.

We need a bridge between the world with or without overflows.

Example: a loop fusion[1]

```
for (i=0;i<N+1;i++)
   S1(i);

for (i=0;i<N  ;i++)
   S2(i);
```

```
for (i=0;i<N;i++)
  {
    S1(i);
    S2(i);
  }
S1(N);
```

This transformation is not correct when N=MAXINT.

[1] Courtesy of Tobias Grosser

Solution within Polly

1. Test if the loop variables use only signed types (undefined semantics in C). Ignore loops containing unsigned loop variables.
   We can not use the fact the semantics is undefined because the semantics is *defined* within CompCert.
2. Find a type *large enough* to represent loop variables of the optimized program.
   We believe such a type does not always exist.
   "But for *real* programs 64 bits is enough !"
   Maybe, but formalizing this argument would require to change the semantics preservation theorem.

Another solution

Use an extension of the polyhedral model to formalize modulo arithmetic such as ISL[2].

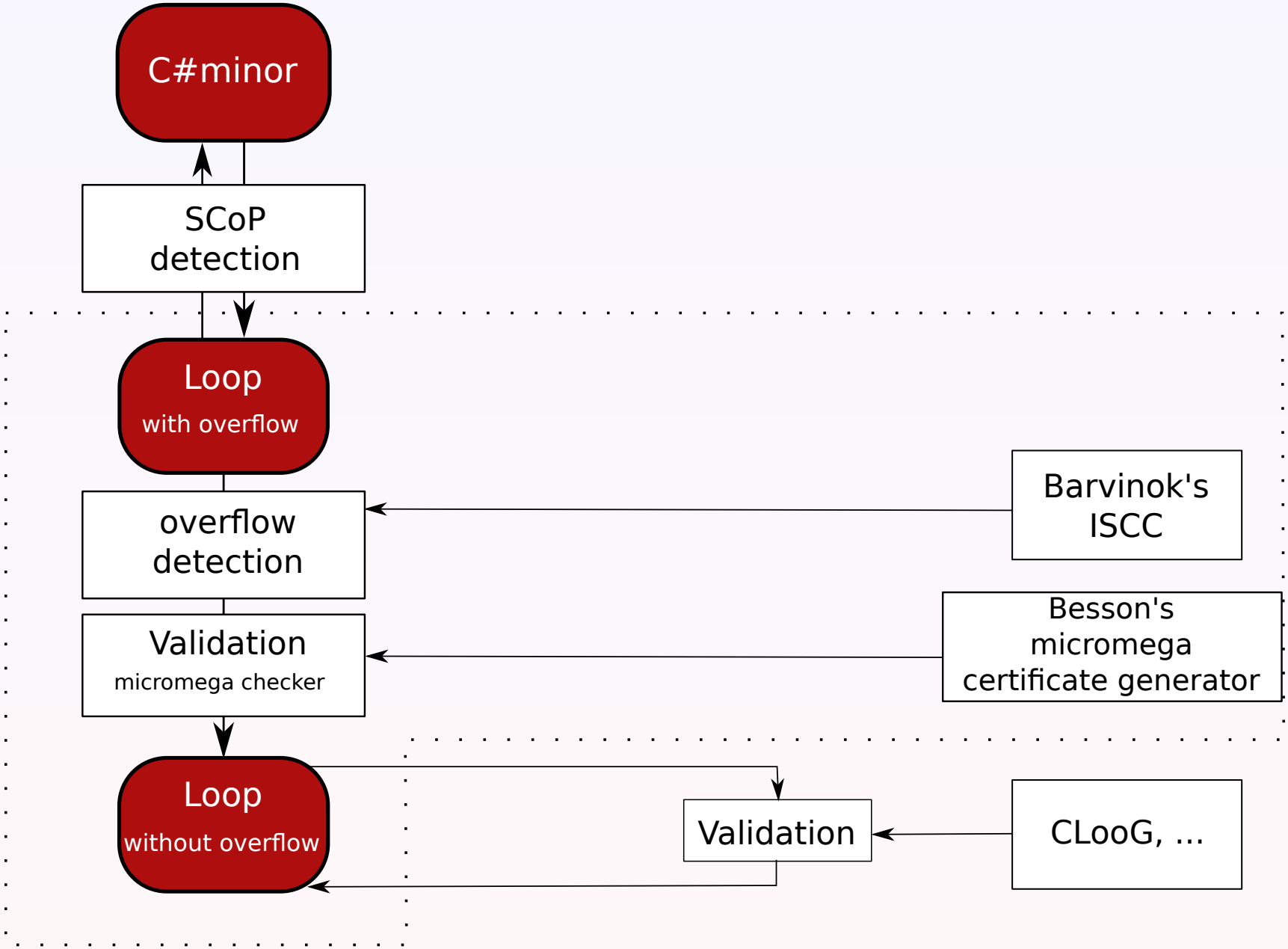
- There are more constrains to satisfy.
- Optimization can be found even in the case of overflows.

[2]We were not aware of this solution when we started this work!

Solution proposed

It is not possible to perform static analysis because loops depends on parameters which (in general) are unknown.

1. Find a sufficient condition for the absence of overflow in either the original or optimized program.
2. Test dynamically this condition.


C#minor
SCoP detection
Loop with overflow
overflow detection
Validation micromega checker
Loop without overflow
Barvinok's ISCC
Besson's micromega certificate generator
Validation
CLooG, ...

Language

Syntax

$$
\begin{array}{lcl}
l & ::= & n \mid x \mid n * l \mid l + l \mid l - l \\
e & ::= & n \mid x \mid T[l] \mid e + e \mid e - e \mid e * e \mid e/e \\
i & ::= & \text{skip} \mid T[l] := e \mid i; i \mid \text{Loop } x \text{ from } 0 \text{ to } l \text{ do } i \text{ done}
\end{array}
$$

- l linear index expressions,
- e expressions
- i instructions
- n constants
- x variables

Language II

We also add the possibility to enclose our SCoPs inside a conditional instruction:

Syntax

$$\begin{array}{lcl} b & ::= & \text{true} \mid \text{false} \mid l \leq l \mid b \text{ and } b \mid b \text{ or } b \\ j & ::= & \text{If } b \text{ then } i \text{ else } i \end{array}$$

For sake of simplicity, we assume that all variables (x) are 32-bit signed integers represented using two's complement.

Condition for absence of overflow

We define the sufficient condition `cond_overflow`, by induction on the structure of programs:

Skip

$$true$$

t(l):=e

$$\bigwedge_{expr \in subexpr(l) \cup lsubexpr(e)} \texttt{MININT} \leq expr \leq \texttt{MAXINT}$$

$i_1; i_2$

$$cond\_overflow(i_1) \wedge cond\_overflow(i_2)$$

$Loop(x, l)\ i$

$$0 \leq x \leq l \Rightarrow cond\_overflow(i) \wedge \bigwedge_{expr \in subexpr(l)} (\texttt{MININT} \leq expr \leq \texttt{MAXINT})$$

Example

```
for (i=0;i<=N;i++)
 {
  T[i+1]=55;
  for (j=0;j<=M;j++)
    T[i+j]=12;
  for (j=0;j<=i;j++)
    T[i+2*j] = 42;
 }
```

Example

http://www.cs.kuleuven.be/cgi-bin/dtai/barvinok.cgi

Query

```
{[n,m]} - {[n,m] | exists i: exists j:
(0 <= i <= n /\ (-32768 > i+1 \/ i+1 > 32767))
\/ (0 <= i <= n /\ 0 <= j <= m /\ (-32768 > i+j \/
                                   i+j > 32767))
\/ (0 <= i <= n /\ 0 <= j <= i /\ (-32768 > 2*j \/
            2*j > 32767 \/ -32768 > i+2*j \/ i+2*j > 32767))
};
```

Answer

```
{ [n, m] : n <= -1 or (n <= 10922 and
                       n >= 0 and
                       m <= 32767 - n) }
```

Our Hypotheses

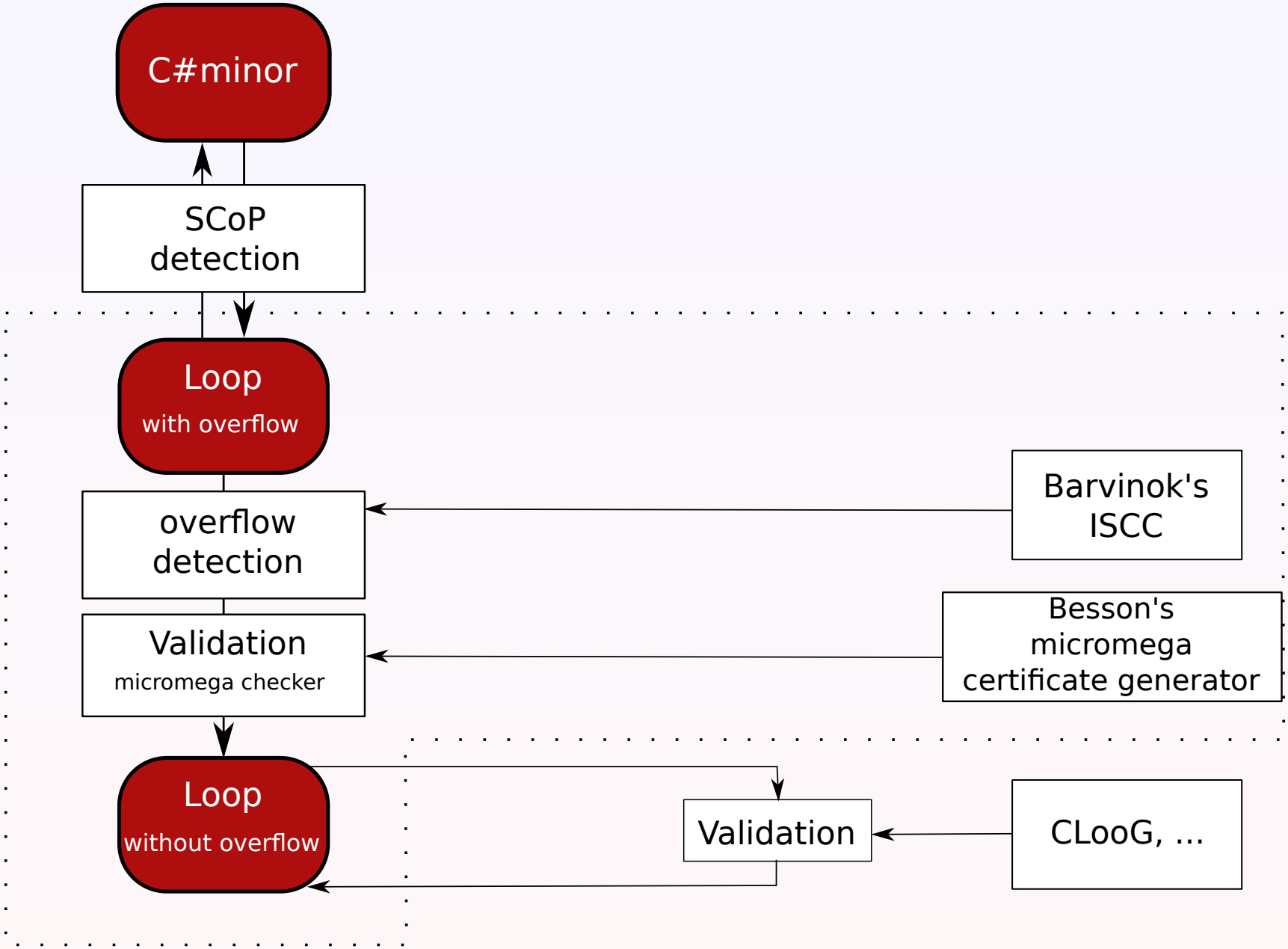
We assume we have:

- a compiler optimization pass based on the polyhedral model,
- and a proof that this optimization pass is correct in a world without overflows (on the indexes of the loops).

En Coq

```
Definition polyhedral_optim (p: instr) := ...

Axiom polyhedral_optim_correct :
 forall p p', polyhedral_optim p = Some p'->
 loop.equiv NO p p'.
```


C#minor
SCoP detection
Loop with overflow
overflow detection
Validation micromega checker
Loop without overflow
Barvinok's ISCC
Besson's micromega certificate generator
Validation
CLooG, ...

Compiler

```
(* Implemented in ml using iscc *)
Parameter oracle : instr -> instr -> bexpr.

Definition compile p :=
  match (polyhedral_optim p) with
  None => None
| Some opt =>
   let new := If (oracle p opt) opt p in
    if validator p opt new then Some new else None
end.
```

Validator

Input:

- **org** the original program
- **opt** the optimized program produced by the polyhedral optimizer
- **new** the program we produced (which either executes org or opt depending on overflows).

Output: If it returns `true`, this means the new program is equivalent to the original one.

Algorithm: It checks the program, if it has not the shape : `If b org opt` then it returns `false`, otherwise its checks whether b implies (cond_overflow org) and (cond_overflow opt).
In addition, we check that b does only feature parameters.

Problem

Evaluating the condition to ensure the absence of overflows may itself lead to an overflow...

Ideas:

1. Find a sufficient condition such that checking the condition to ensure no overflows happen does not trigger overflows.
   - Problem: Evaluating the condition to ensure no overflows occur may lead to an overflow...
     1. Find a sufficient condition such that checking the condition to ensure no overflows happen does not trigger overflows.

     ...
2. Find a sufficient condition whose evaluation does not trigger overflows.
3. Evaluate the condition to ensure no overflows happen using sufficient/arbitrary precision.
4. Check whether evaluating b triggers an overflow.

Checking whether evaluationg b triggers an overflow.

A pragmatic approach

If evaluating b triggers an overflow,
we execute the initial program.

- We modify the evaluation semantics of linear expressions

$$\mathcal{L} \vdash_{oc} l \longrightarrow (n, e)$$

n : value computed for l
e : bool = *true* if evaluating *l* does not trigger overflows,
false otherwise.

- Tests ($\leq$) on linear expr. return *false* if overflows occur.

Conclusions/Future Works

Conclusions

- 3000 lines of Coq, 300 lines of Ocaml.
- Reusing the internals of Coq tactic `micromega` was rather straightforward.

Overflows

- Complete the formal proofs.
- Scalability ?
- Integrate the proofs into a more realistic compiler (CompCert).

Thank you.