Handling Overflow in MLton

2012 ACM SIGPLAN Workshop on ML

Alexander Bjerremand Hansen [1]
Jan Midtgaard [2]

[1] Mjølner Informatics A/S
[2] Aarhus University

September 13, 2012

The MLton compiler

- MLton is an Standard ML compiler
  - Open source
  - Industrial strength (Intel, MathWorks,etc.)
  - Whole-program
  - Optimizing
- It has been developed over the last 15 years

Handling overflow

In an intermediate representation integer arithmetic can take the form:

$$\ell(x_1 + x_2)\ \texttt{Overflow => }\ \ell'$$

If $x_1 + x_2$ does not overflow this could be optimized to

$$x \leftarrow \texttt{Word32\_add}(x_1, x_2)$$
$$\ell(x)$$

Handling overflow

In an intermediate representation integer arithmetic can take the form:

$$\ell(x_1 + x_2)\ \texttt{Overflow =>}\ \ell'$$

If $x_1 + x_2$ does not overflow this could be optimized to

$$\begin{aligned} & x \leftarrow \texttt{Word32\_add}(x_1, x_2) \\ & \ell(x) \end{aligned}$$

Bad news!

Handling overflow

In an intermediate representation integer arithmetic can take the form:

$$\ell(x_1 + x_2)\ \texttt{Overflow =>}\ \ell'$$

If $x_1 + x_2$ does not overflow this could be optimized to

$$x \leftarrow \texttt{Word32\_add}(x_1, x_2)$$
$$\ell(x)$$

Bad news!

MLton produces sub-optimal code regarding overflow checks.

Dead code - example

When compiling `hamlet.sml` we find this snippet among the 110k lines of intermediate code:

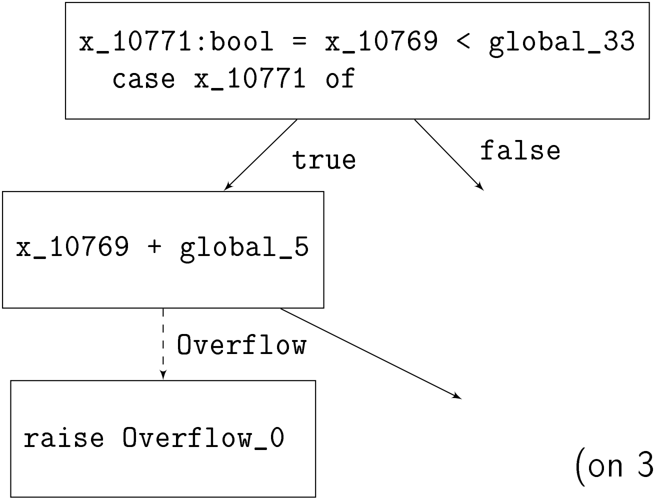
```
L_25120 (x_16013: word32)
 x_10771: bool = WordU32_lt (x_10769, global_33)
 case x_10771 of
   true => L_22171 | false => L_24271
L_22171 ()
 loop_355 (x_10769 + global_5) Overflow => L_24272()
```

Relevant constants:

```
global_5:  word32 = 0x1
global_33: word32 = 0x3B7
```

Dead code - example

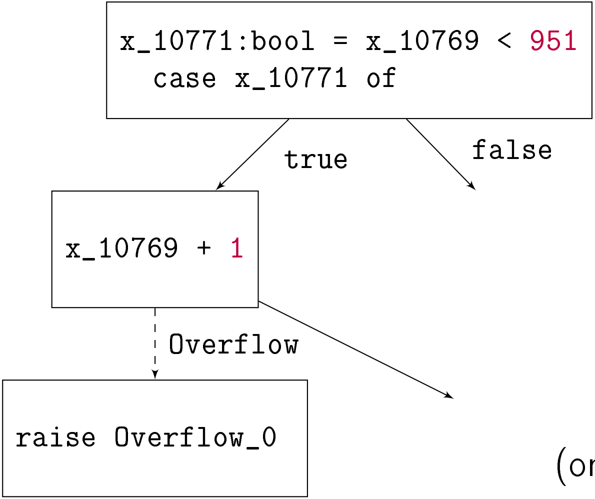
The same code typeset as a flow graph:



(on 32 bit)

Dead code - example

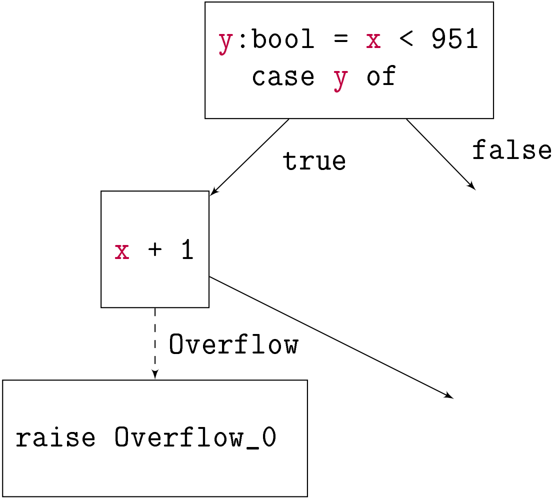
With global constants inlined:



(on 32 bit)

Dead code - example

With variables α-renamed:



(on 32 bit)

Motivation

Bad news!
MLton produces sub-optimal code regarding overflow checks.

The question is: Can we formulate a static analysis to improve the generated code?

Motivation

Bad news!
MLton produces sub-optimal code regarding overflow checks.

The question is: Can we formulate a static analysis to improve the generated code?

Good news!

Motivation

Bad news!
MLton produces sub-optimal code regarding overflow checks.

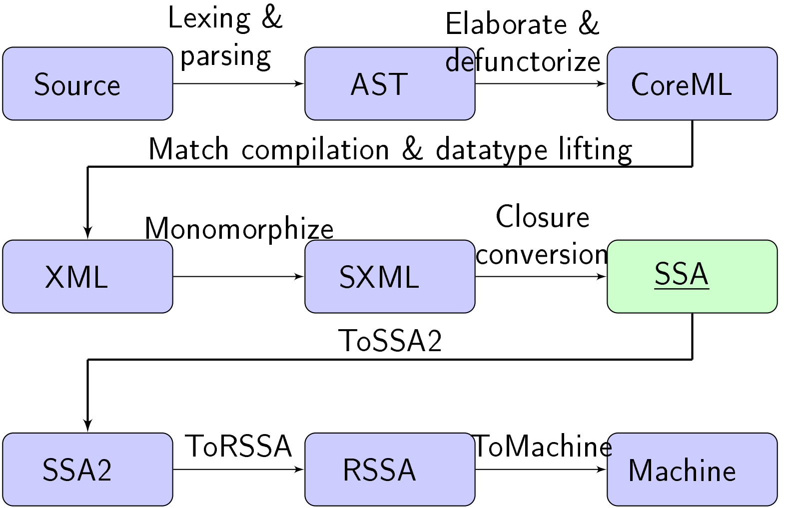
The question is: Can we formulate a static analysis to improve the generated code?

Good news!
Yes we can!

Agenda

- Motivation
- MLton in more detail
- Detecting dead code in integer arithmetic with interval analysis
- Analysis
- Results

MLton in more detail



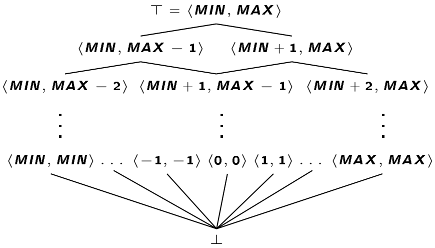
MLton in more detail - SSA form

- Static Single-Assignment form
- Each variable has one definition in the program text
  - Static property
- Simply-typed
  - Polymorphism eliminated by monomorphization
- First-order
  - Higher order functions eliminated by defunctionalization (closure conversion)
- Most optimizations at SSA level

Interval Analysis

The standard way of analysing integer variables is Interval/Range analysis as proposed by Cousot and Cousot [1976].

Lattice of intervals (*IL*):



Lattice relating variables with intervals (I):

$$I : Var \rightarrow IL$$

Initial lookup function

When we need the abstract value of a variable $\mathtt{x}$ in block ℓ:

$$L(\mathrm{x}, \ell) = I[\![\mathrm{x}]\!]$$

This version ignores the block and returns the value from the I lattice.

Constraints

We express the analysis using constraints. Examples:

$$\frac{(X \leftarrow v) \in P \qquad v : \mathit{word}}{I[\![X]\!] = \langle v, v \rangle} \text{ Constant Definition}$$

$$\frac{\begin{array}{cc} \ell(X) \in P & \ell(x_1 \;\texttt{op}\; x_2) = S \\ S \in P & \mathit{label}(S) = \ell' \end{array}}{(L(x_1, \ell') \;\bar{\texttt{op}}\; L(x_2, \ell')) \sqsubseteq I[\![X]\!]} \text{ Arithmetic}$$

$$\frac{\begin{array}{c} \texttt{f}(X_1, ..., X_n) \in P \qquad \ell'(X_r) \in P \\ (\texttt{f}(x_1, ..., x_n)\;\texttt{NonTail}\;\{\texttt{cont=}\ell'\texttt{,handler=}H\}) = S \\ S \in P \qquad i \in \{1, \ldots, n\} \qquad \mathit{label}(S) = \ell'' \\ x_i : \mathit{word} \end{array}}{L(x_i, \ell'') \sqsubseteq I[\![X_i]\!]} \text{ NonTail Call}$$

Constraints

We express the analysis using constraints. Examples:

$$\frac{(X \leftarrow v) \in P \qquad v : \mathit{word}}{I[\![X]\!] = \langle v, v \rangle} \text{ Constant Definition}$$

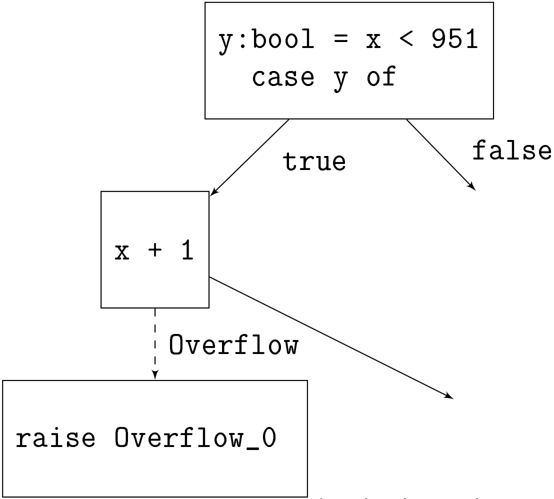
Not enough

$$(L(x_1, \ell') \; \bar{\texttt{op}} \; L(x_2, \ell')) \sqsubseteq I[\![X]\!]$$

$$\frac{\begin{array}{c} \texttt{f}(X_1, \ldots, X_n) \in P \qquad \ell'(X_r) \in P \\ (\texttt{f}(x_1, \ldots, x_n) \; \texttt{NonTail} \; \{\texttt{cont=}\ell'\texttt{,handler=}H\}) = S \\ S \in P \qquad i \in \{1, \ldots, n\} \qquad \mathit{label}(S) = \ell'' \\ x_i : \mathit{word} \end{array}}{L(x_i, \ell'') \sqsubseteq I[\![X_i]\!]} \text{ NonTail Call}$$

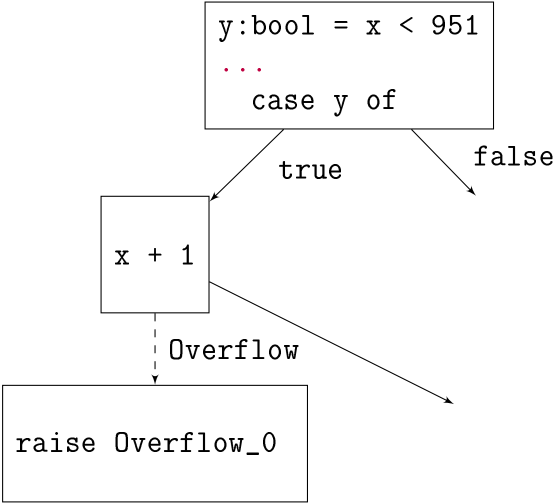
What do we need? 1/2

We need conditional information from boolean comparisons of integers.



What do we need? 2/2

The information from the comparison must be saved for later utilization.



Flow sensitivity - 1/2

$A_{\mathrm{T},y}[\![X]\!]$ is the interval for y if X is `true`

Flow sensitivity - 1/2

$A_{\mathrm{T},y}[\![X]\!]$ is the interval for y if X is `true`

$\prec$ and $\succeq$ bound intervals from above or below.

Flow sensitivity - 1/2

$A_{\mathtt{T},y}[\![X]\!]$ is the interval for y if X is `true`

$\prec$ and $\succeq$ bound intervals from above or below.

$$\frac{(X \leftarrow \mathtt{Word\_lt}(x_1, x_2)) = S \qquad S \in P \qquad X : bool \qquad label(S) = \ell}{\begin{array}{c} A_{\mathtt{T},x_1}[\![X]\!] = L(x_1,\ell) \prec L(x_2,\ell) \\ A_{\mathtt{T},x_2}[\![X]\!] = L(x_2,\ell) \succeq L(x_1,\ell) \\ A_{\mathtt{F},x_1}[\![X]\!] = L(x_1,\ell) \succeq L(x_2,\ell) \\ A_{\mathtt{F},x_2}[\![X]\!] = L(x_2,\ell) \prec L(x_1,\ell) \end{array}} \text{ Primitive Definition - Less-Than}$$

Flow sensitivity - 2/2

A fact lattice F stores optional local extra information for variables.

$$F : B \to Var \to IL$$

Flow sensitivity - 2/2

A fact lattice F stores optional local extra information for variables.

$$F : B \to Var \to IL$$

A case transfer can propagate the four intervals down through the control paths:

$$\frac{\begin{array}{c}(\texttt{case } x \texttt{ of true=>}\ell \texttt{ | false=>}\ell') \in P \qquad x : bool \\ A_{\mathrm{T},y}[\![x]\!] = i \qquad A_{\mathrm{T},z}[\![x]\!] = i' \\ A_{\mathrm{F},y}[\![x]\!] = i'' \qquad A_{\mathrm{F},z}[\![x]\!] = i'''\end{array}}{\begin{array}{c} i \sqsubseteq F[\![\ell]\!][\![y]\!] \qquad i' \sqsubseteq F[\![\ell]\!][\![z]\!] \\ i'' \sqsubseteq F[\![\ell']\!][\![y]\!] \qquad i''' \sqsubseteq F[\![\ell']\!][\![z]\!]\end{array}} \text{ Case Transfer - Bool}$$

Improved lookup function

Search the ancestor tree for facts about the input variable first.
Let $anc(\ell)$ be the ancestor to the block ℓ.

$$L(x,\ell) = \begin{cases} F[\![\ell]\!][\![x]\!] & \text{, if } F[\![\ell]\!][\![x]\!] \neq \bot \\ I[\![x]\!] & \text{, if } F[\![\ell]\!][\![x]\!] = \bot \wedge \\ & \quad anc(\ell) = \emptyset \\ L(x, anc(\ell)), & \text{otherwise} \end{cases}$$

The implementation

We have implemented the analysis and transformation as an additional SSA optimization pass in MLton.

It takes approximately 1500 lines of SML.

The pass is followed by the pass `RemoveUnused` which takes care of removing potential dead blocks after our transformation.

Results - example

Transforms the simplified, α-renamed `hamlet.sml` example from earlier:

```
L
  y: bool = WordU32_lt (x, 951)
  case y of
    true => L' | false => ...
L'
 L'' (x + 1) Overflow => L''' ()
```

into

```
L
  y: bool = WordU32_lt (x, 951)
  case y of
    true => L' | false => ...
L'
  z: word32 = Word32_add (x, 1)
  L'' (z)
```

Benchmarks - Hits

Benchmarks are from `fib` to `hamlet`.

Using the MLton benchmark suite:

Handlers	Transformed
4240	913

So around 22% of the overflow-checking arithmetic in the benchmark suite is transformed into primitive operations.

Benchmarks - Size 1/2

Benchmark	# hits	without (KB)	with (KB)	Relative
mandelbrot	5	117,383	104,643	10.9%
flat-array	4	117,119	104,631	10.7%
zern	32	141,925	140,501	1.0%
barnes-hut	25	158,891	157,407	0.9%
smith-normal-form	53	249,054	247,326	0.7%
fft	12	136,160	135,264	0.7%
raytrace	69	308,684	307,084	0.5%

Benchmarks - Size 2/2

Total reduction of the binary sizes in the 43 benchmarks is 52 kB or 0.6% of the total size.

Average reduction: 1.2 kB or 0.85%.

The reduction ranges from 80 B to 12.7 kB.

Every benchmark has reduced binary size.

Benchmarks - Run-time 1/2

Benchmark	# hits	without (s)	with (s)	Relative
matrix-multiply	8	6.25	4.56	-27%
imp-for	4	8.88	6.55	-26%
tailfib	5	10.54	8.45	-20%
mandelbrot	5	8.70	8.20	-6%
md5	20	11.65	11.38	-2%
zern	32	6.03	5.89	-2%

Benchmarks - Run-time 1/2

Benchmark	# hits	without (s)	with (s)	Relative
matrix-multiply	8	6.25	4.56	-27%
imp-for	4	8.88	6.55	-26%
tailfib	5	10.54	8.45	-20%
mandelbrot	5	8.70	8.20	-6%
md5	20	11.65	11.38	-2%
zern	32	6.03	5.89	-2%

Benchmark	# hits	without (s)	with (s)	Relative
hamlet	41	9.60	9.59	0%
lexgen	29	5.46	5.45	0%

Benchmarks - Run-time 1/2

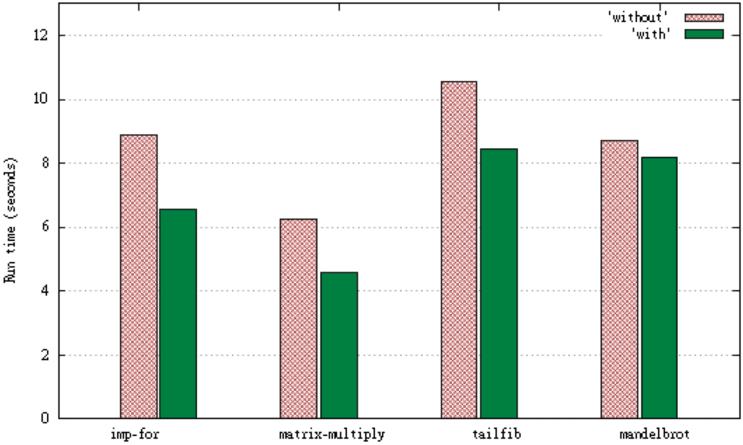
Benchmark	# hits	without (s)	with (s)	Relative
matrix-multiply	8	6.25	4.56	-27%
imp-for	4	8.88	6.55	-26%
tailfib	5	10.54	8.45	-20%
mandelbrot	5	8.70	8.20	-6%
md5	20	11.65	11.38	-2%
zern	32	6.03	5.89	-2%

Benchmark	# hits	without (s)	with (s)	Relative
hamlet	41	9.60	9.59	0%
lexgen	29	5.46	5.45	0%

Benchmark	# hits	without (s)	with (s)	Relative
vector-rev	5	9.08	9.42	4%
tak	4	6.16	6.69	9%

Benchmarks - Run-time 2/2

Average reduction in run-time: 1.8%.



Compile time

Benchmark	# hits	CT without	CT with	Relative
hamlet	41	10.42	29.57	184%
mlyacc	79	5.92	14.29	141%
model-elimination	67	5.44	9.21	70%
vliw	44	4.16	6.75	62%
raytrace	69	3.31	4.20	27%

Compile time

Benchmark	# hits	CT without	CT with	Relative
hamlet	41	10.42	29.57	184%
mlyacc	79	5.92	14.29	141%
model-elimination	67	5.44	9.21	70%
vliw	44	4.16	6.75	62%
raytrace	69	3.31	4.20	27%

Benchmark	# hits	CT without	CT with	Relative
zern	32	2.08	2.16	4%
nucleic	4	3.38	3.49	3%
tsp	19	2.06	2.12	3%
knuth-bendix	25	2.28	2.33	2%

Average increase in compile time: 15%.

Conclusion

- With relatively straightforward application of static analysis we improve the generated code by MLton.
- Our improvement:
  - 22% of overflow checks transformed
  - 0%-10% reduced binary size
  - Up to 27% reduction in run-time.
- There is still room for improvement though:
  - in the choice of data structures.
  - in analysis precision.
  - in the compile-time for the analysis.

Conclusion

- With relatively straightforward application of static analysis we improve the generated code by MLton.
- Our improvement:
  - 22% of overflow checks transformed
  - 0%-10% reduced binary size
  - Up to 27% reduction in run-time.
- There is still room for improvement though:
  - in the choice of data structures.
  - in analysis precision.
  - in the compile-time for the analysis.

Master's Thesis:
http://cs.au.dk/~alx/

Conclusion

- With relatively straightforward application of static analysis we improve the generated code by MLton.
- Our improvement:
  - 22% of overflow checks transformed
  - 0%-10% reduced binary size
  - Up to 27% reduction in run-time.
- There is still room for improvement though:
  - in the choice of data structures.
  - in analysis precision.
  - in the compile-time for the analysis.

Master's Thesis:
http://cs.au.dk/~alx/

Thank you.

Widening

When a join of intervals coarsen the bounds widening is performed.

Widening consist of jumping the bounds of the resulting interval to the nearest integer from a set B.

$$w(\langle l, h\rangle) = \langle \max\{x \in B | x \leq l\}, \min\{x \in B | x \geq h\}\rangle$$

B is all integer constants in the program.