Interprocedural register allocation for the WAM based on source-to-source transformations.

Ulrich Neumerkel

Institut für Computersprachen
Technische Universität Wien
ulrich@mips.complang.tuwien.ac.at

- new intermediate language — Continuation Prolog
- transformation driven optimization, global analysis not required
- improved code generation for WAM and BinWAM
- minimal extension of WAM-code generation integrated into SICStus

Problems with current implementations

e ⟶ "t".

$$\mathrm{e}([0\text{'}t|\overrightarrow{Xs}],Xs).$$

e ⟶ "o", e, e.

$$\mathrm{e}([0\text{'}o|\overrightarrow{Xs0],Xs)\ \leftarrow\ e(Xs0},\overrightarrow{Xs1),\ e(Xs1},Xs).$$

$$\underleftarrow{\quad Xs\quad}$$

Problems in WAM:

- short lifetime of registers
- argument register bottleneck

$$\mathrm{e}([0\text{'}o|\overbrace{Xs0],Xs)\ \leftarrow e(Xs0}^{Xs0},\overbrace{Xs1),\ e(Xs1}^{Xs1},Xs).\ \%\quad \text{lifetime of variables disjoint}$$

$$\underbrace{Xs)\ \leftarrow e(Xs0,Xs1),\ e(Xs1,Xs}_{Xs}$$

- cannot be solved by mode analysis (influence of usage, e.g. ← e(Xs,[]).)
- cannot be solved by fold/unfold (lacks context between goals)

Continuation Prolog

```
e([0't|Xs],Xs).                    [ e([0't|Xs],Xs) ] ⟸
                                          [].
e([0'o|Xs0],Xs) ←                  [ e([0'o|Xs0],Xs) ] ⟸
      e(Xs0,Xs1),                         [ e(Xs0,Xs1),
      e(Xs1,Xs).                            e(Xs1,Xs) ].
```

- clauses: multiple heads — multiple goals, order significant

```
[a,b] ⟸
      [c].
```

```
← inferencediff([a,b,d], [c,d]).
```

Outline of interprocedural register allocation

contprolog (CProg0) ←

contprolog_ebctransformed (CProg0,CProg) % EBC-Transformation

(Prolog→) Continuationprolog→ Transformed

1,

Outline of interprocedural register allocation

contprolog_llprologallocated (CProg0, LLProg) ←

contprolog_ebctransformed (CProg0,CProg), % EBC-Transformation
contprolog_llprolog (CProg, LLProg). %

(Prolog→) Continuationprolog→ Transformed → Low-Level Prolog → WAM

1, 2,

Outline of interprocedural register allocation

contprolog_llprologallocated_(CProg0, LLProg, Anns) ←
    contprolog_(CProg0, Anns), % Heuristics
    contprolog_ebctransformed_(CProg0,CProg, Anns), % EBC-Transformation
    contprolog_llprolog_(CProg, LLProg, Anns). %

(Prolog→) Continuationprolog→ Transformed → Low-Level Prolog → WAM

↘ ↑ ↗

Heuristics — prunes search space
does not affect correctness

1, 2, 3,

Outline of interprocedural register allocation

contprolog_llprologallocated_(CProg0, LLProg, Anns) ←
contprolog_(CProg0, Anns), % Heuristics
contprolog_ebctransformed_(CProg0,CProg, Anns), % EBC-Transformation
contprolog_llprolog_(CProg, LLProg, Anns). %

Transf.→ Bin.Pr., BinWAM
↗
(Prolog→) Continuationprolog→ Transformed → Low-Level Prolog → WAM
↘ ↑ ↗
Heuristics — prunes search space
does not affect correctness

1, 2, 3, 4.

EBC- (*equality based continuation*) transformation system

Preserves operational equivalence of Continuation Prolog programs

Equations $s \doteq t$ **for alternative representations**

$u \cdot s\theta \cdot v$ is equivalent to $u \cdot t\theta \cdot v$

$[\mathrm{old}(a_1, ...a_n, b_1, ...b_m)] \doteq [\mathrm{new1}(a_1, ...a_n), \mathrm{new2}(b_1, ...b_m)]$ $\quad a_i, b_i$ distinct vars

Compilation

body $= u \cdot s\theta \cdot v$ is translated into $u \cdot t\theta \cdot v$

heads All heads are rewritten.

add interface predicate (required for meta-calls and entry-points)

$[\mathrm{old}(a_1, ...a_n, b_1, ...b_m)] \twoheadleftarrow [\mathrm{new1}(a_1, ...a_n), \mathrm{new2}(b_1, ...b_m)]$

Simplification $h \cdot r \twoheadleftarrow b \cdot r$ simplifies to $h \twoheadleftarrow b$

if r contains only variables not in h and b

Example: Translation of arithmetical expressions

```
%
expr(A+B,T) ⟶
    expr(A,TA),
    expr(B,TB),
    ['add'(TA,TB,T)].
```

```
% Output unifications regrouped
[expr(A+B,T,Xs0,Xs)] ↞
  [   expr(A,TA,Xs0,Xs1),
      expr(B,TB,Xs1,Xs2),
      Xs2 = ['add'(TA,TB,Tx)|Xs3],
      [Tx,Xs3] = [T,Xs]
  ].
```

Example: Splitting output arguments

[expr(E,T,Xs0,Xs)] $\doteq$ [expr1(E,Xs0), r(T,Xs)].
[[Tx,Xs0] = [T,Xs]] $\doteq$ [=([Tx,Xs0]), r(T,Xs)].

```
[expr(E,T,Xs0,Xs)] ⟵
  [   expr1(E,Xs0),
      r(T,Xs)
  ].

[=([T,Xs]), r(T,Xs)] ⟵
  [].
```

```
[expr1(A+B,Xs0), r(T,Xs)] ⟵
  [   expr1(A,Xs0),
      r(TA,Xs1),
      expr1(B,Xs1),
      r(TB,Xs2),
      Xs2 = ['add'(TA,TB,Tx)|Xs3],
      =([Tx,Xs3]),
      r(T,Xs)
  ].
```

1,

Example: Splitting output argumentsand simplification

[expr(E,T,Xs0,Xs)] $\underset{\cdot}{=}$ [expr1(E,Xs0), r(T,Xs)].
[[Tx,Xs0] = [T,Xs]] $\underset{\cdot}{=}$ [=([Tx,Xs0]), r(T,Xs)].

[expr(E,T,Xs0,Xs)] $\twoheadleftarrow$
  [expr1(E,Xs0),
    r(T,Xs)
  ].

[=([T,Xs]), r(T,Xs)] $\twoheadleftarrow$
  [].

[expr1(A+B,Xs0), ~~r(T,Xs)~~] $\twoheadleftarrow$
  [expr1(A,Xs0),
    r(TA,Xs1),
    expr1(B,Xs1),
    r(TB,Xs2),
    Xs2 = ['add'(TA,TB,Tx)|Xs3],
    =([Tx,Xs3]),
    ~~r(T,Xs)~~
  ].

1, 2.

Continuation Prolog to low-level Prolog.

- AND-control mapped onto stack
- only a subset can be translated
  - Predicate functors. Occur as the first element in the head.
  - Continuation functors. Occur as the second element in the head.

  must be disjoint

  - empty body for clauses with continuation functor in the head
  - continuation functor must be the same for all clauses of the same predicate

$$[\ e(Xs), r(Xs0)\ ] \twoheadleftarrow$$
$$[\ e(Xs1), r(Xs0), e(Xs), r(Xs1)\ ].$$

Example: translation into low-level Prolog

```
[expr1(A+B,Xs0)] ⇐—
  [  expr1(A,Xs0),
     r(TA,Xs1),
     expr1(B,Xs1),
     r(TB,Xs2),
     Xs2 = ['add'(TA,TB,Tx)|Xs3],
     =([Tx,Xs3])
  ].
```

```
expr1(A+B,Xs0) ←
     expr1(A,Xs0),
     'SOURCE'(+TA,+Xs1),
     expr1(B,Xs1),
     'SOURCE'(+TB,+Xs2),
     Xs2 = ['add'(TA,TB,T)|Xs3],
     'SINK'(+T,+Xs3).
```

Low level-Prolog to WAM-code

Two low level built-ins.

- 'SINK'/n: ensure that n values are in registers
- 'SOURCE'/n: assume that n values are in registers

```
                        [pop(C,P)] ↞                  pop(C,P) ←                  [allocate
                          [  pop(C),                      pop(C),                 ,get_y_variable(0,1)
                             r(P) ].                      'SOURCE'(-,+P).         ,init([])
                                                                                  ,call(pop/1,1)
                                                                                  ,get_y_value(0,1)
                                                                                  ,deallocate
                                                                                  ,execute(true/0)].
pop(china, 8250).       [pop(china), r(8250)] ↞       pop(china) ←
                          [].                             'SINK'(-,+(8250)).      [get_constant_x0(china)
                                                                                  ,put_constant(8250,1)
                                                                                  ,proceed].
pop(india, 5863).       [pop(india), r(5863)] ↞       pop(india) ←
... .                     [].                             'SINK'(-,+(5863)).      [get_constant_x0(india)
                        ... .                         ... .                       ,put_constant(5863,1)
                                                                                  ,proceed].
```

Final example code

```
% Prolog code            % Original WAM              % Optimized WAM               % ll-Prolog
expr(I, T, Xs0,Xs) ←                                                               expr1(I, Xs0) ←
  integer(I),            [builtin(integer(0),else)   [builtin(integer(0),else)       integer(I),
  !,                     ,cutb                       ,cutb                           !,
  I = T,                 ,get_x_val(1,0)             ,proceed].                      'SINK'(+I, +Xs0).
  Xs0 = Xs.              ,get_x_val(2,3)
                         ,proceed].


                                                                                   % ll-Prolog
expr(A+B,                [get_str_x0((+)/2)          [get_str_x0((+)/2)            expr1(A+B, Xs0) ←
                         ,allocate                   ,unify_x_var(0)
  T, Xs0,                ,get_y_var(4,3)             ,allocate
  Xs) ←                  ,get_y_var(3,1)             ,unify_y_var(1)
                         ,unify_x_var(0)
                         ,unify_y_var(5)
                         ,put_y_var(1,1)
                         ,put_y_var(6,3)
                         ,init([0,2])                ,init([0])
  expr(A,TA,Xs0,Xs1),    ,call(expr/4,7)             ,call(expr1/2,2)                expr1(A, Xs0),
                         ,put_y_val(5,0)             ,get_y_first_val(0,0)%%         'SOURCE'(+TA,+Xs1),
                         ,put_y_first_val(2,1)
                         ,put_y_unsafe_val(6,2)
                         ,put_y_first_val(0,3)       ,put_y_val(1,0)%%
  expr(B,TB,Xs1,Xs2),    ,call(expr/4,5)             ,call(expr1/2,1)                expr1(B, Xs1),
  Xs2 = [                ,put_y_val(0,0)
                         ,get_list(0)                ,get_list(1)                    'SOURCE'(+TB,+[
                         ,unify_temp_var(0)          ,unify_temp_var(2)
                         ,unify_y_local_val(4)       ,unify_x_var(1)
      'add'(             ,get_temp_str('add'/3,0)    ,get_temp_str('add'/3,2)            'add'(
        TA,              ,unify_y_local_val(1)       ,unify_y_local_val(0)                 TA,
        TB,              ,unify_y_local_val(2)       ,unify_x_local_val(0)                 TB,
        T) | Xs].        ,unify_y_local_val(3)       ,unify_x_var(0)                       T) | Xs]),
                         ,deallocate                 ,deallocate
                         ,execute(true/0)].          ,execute('SINK'/2)].            'SINK'(+T,+Xs).
```

- + fewer instructions
- + simpler instructions (less trail checking, fewer general unifications)
- + 30% faster
- + smaller environments (7+4 vs. 2+1 vars)
- + fewer argument registers (4 vs. 2)

Blocked goals/Constraints

- life registers in call, execute and heapmargin_call
- Simulating block

```
← block predicate(-, ...).
predicate(I1, ...) ←
    var(I1),
    !,
    blockable_predicate(I1,..., O1,...),
    'SINK'(+O1,...).
predicate(_I1, ...) ←
    ... .
```

```
← block blockable_predicate(-, ?, ?, ?).
blockable_predicate(I1,..., O1,...) ←
    predicate(I1,...),
    'SOURCE'(+O1,...).
```

Direct compilation of DCGs

$$a \longrightarrow$$
$$\quad [].$$

$$a(Xs) \leftarrow$$
$$\quad 'SINK'(-,+Xs).$$

$$b(1) \longrightarrow$$
$$\quad [].$$

$$b(1, Xs) \leftarrow$$
$$\quad 'SINK'(-,+Xs).$$

$$c(1,2) \longrightarrow$$
$$\quad [].$$

$$c(1, Xs, 2) \leftarrow$$
$$\quad 'SINK'(-,+Xs).$$

```
qsorted([E|Es]) ⟶
    {partition(Es,E,Es1,Es2)},
    qsorted(Es1),
    [E],
    qsorted(Es2).
qsorted([]) ⟶
    [].
```

```
qsorted([E|Es], Xs0) ←
    partition(Es,E,Es1,Es2),
    qsorted(Es1, Xs0),
    'SOURCE'(-,+[E|Xs1]),
    qsorted(Es2, Xs1).
qsorted([], Xs0) ←
    'SINK'(-,+Xs0).

phrase_qsorted(Es, Xs0,Xs) ←
    qsorted(Es, Xs0),
    'SOURCE'(-,+Xs).
```

Benchmark results

no code duplication, number of WAM instructions reduced

existing			
	modif/orig	EBC/modif	tuned/modif
expr	1.53	1.20	1.30
fact 10	—	1.06	2.05
fact 100	—	1.02	1.05
qsort	1.00	1.02	1.14
query	—	1.11	1.21
serial	—	1.00	1.16
d-divide	1.18	—	1.03
d-log	1.55	—	1.00
d-ops	1.25	—	1.05
d-times	1.18	—	1.05
chat	—	1.23	1.31

artificial			
	modif/orig	EBC/modif	tuned/modif
e	—	1.09	—
e+1	—	1.55	—
e+2	—	1.99	—
e+3	—	2.29	—
de	—	1.27	—
de+1	—	1.67	—
de+2	—	2.13	—
de+3	—	2.41	—

⊔

Related Work

Mode analysis: tied to actual usage & control strategy

– depends on control strategy

– depends on actual usage

+ optimizes not steadfast programs

Aquarius Prolog: uninitialized register conversion

Output value placement: cost model to minimize tradeoffs LCO/registers

Destructive updates: HAG (Hidden Accumulator Grammar) uses backtrackable destructive assignment

– coroutining impossible

- EBC+register trailing produces same effect without destructive assignment

Limitations

Steadfastness required

```
expr((X is Expr), Code) ←
    phrase(expr(Expr, X), Code).

expr(V, V) ⟶ {var(V)}, !. % blocking optimization

expr(I, I) ⟶ {integer(I)}, !. % blocking optimization

expr(A+B, T) ⟶ expr(A, TA), expr(B, TB), ['add'(TA,TB,T)].
expr(A-B, T) ⟶ expr(A, TA), expr(B, TB), ['sub'(TA,TB,T)].
expr(A*B, T) ⟶ expr(A, TA), expr(B, TB), ['mul'(TA,TB,T)].
expr(A/B, T) ⟶ expr(A, TA), expr(B, TB), ['div'(TA,TB,T)].
← expr((1 is Expr), Code).
```

1,

Limitations

Steadfastness required

```
expr((X is Expr), Code) ←
    phrase(expr(Expr, X), Code).

~~expr(V, V) ⟶ {var(V)}, !. % blocking optimization~~
expr(V, T) ⟶ {var(V)}, !, {V = T}.
~~expr(I, I) ⟶ {integer(I)}, !. % blocking optimization~~
expr(I, T) ⟶ {integer(I)}, !, {I = T}.
expr(A+B, T) ⟶ expr(A, TA), expr(B, TB), ['add'(TA,TB,T)].
expr(A-B, T) ⟶ expr(A, TA), expr(B, TB), ['sub'(TA,TB,T)].
expr(A*B, T) ⟶ expr(A, TA), expr(B, TB), ['mul'(TA,TB,T)].
expr(A/B, T) ⟶ expr(A, TA), expr(B, TB), ['div'(TA,TB,T)].
← expr((1 is Expr), Code).
```

1, 2.

Conclusion

- supports differences, and steadfast programs
- simulating states gets cheaper
- independent of usage of predicates — optimization of libraries possible
- supports Prolog with blocked goals, constraints

Further work

- adaptations of formalisms like EDCGs (require no analysis)
- extending the lifetime of registers
  - allocation of context arguments
  - caller/callee saved registers