A GrGen.NET Solution of the Compiler Case for the Transformation Tool Contest 2011

Sebastian Buchwald and Edgar Jakumeit

Karlsruhe Institute of Technology (KIT)
buchwald@kit.edu

1 Introduction

The challenge of the Compiler Optimization Case [3] is to perform local optimizations and instruction selection on the graph-based intermediate representation of a compiler. The case is designed to compare participating tools regarding their performance. We tackle this task employing the general purpose graph rewrite system GrGen.NET (www.grgen.net).

2 What is GrGen.NET?

GrGen.NET is an application domain neutral graph rewrite system developed at the IPD Goos of Universität Karlsruhe (TH), Germany [1]. The feature highlights of GrGen.NET regarding practical relevance are:

Fully Featured Meta Model: GrGen.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types. Attributes may be typed with one of several basic types, user defined enums, or generic set, map, and array types.

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption.

Programmed Rule Application: GrGen.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering logical, sequential and iterative control plus variables and storages for the communication of processing locations between rules.

Graphical Debugging: GrShell, GrGen.NET's command line shell, offers interactive execution of rules, visualising together with yComp the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules.

User Manual: The GrGen.NET User Manual guides you through the various features of GrGen.NET, including a step-by-step example for a quick start.

3 Constant folding

The first task is to perform constant folding. Constant folding transforms operations which have only `Const` operands into a `Const` itself, e.g. to transform $1+2$ into 3. This is a local optimization requiring only rules referencing local graph contex.

Due to massive time constraints we were not able to give a self contained paper for this solution until the deadline; we are forced to hint readers at our solution [2] of the Hello World! case [4] for an introduction into the language constructs used.

3.1 Driver and data flow

Constant folding is carried out from the main/driver sequence given in Figure 1, which performs inside a main loop in each step i) first constant folding along data flow with a wavefront algorithm, and then ii) control flow folding together with clean-up tasks.

```
xgrs now:set<FirmNode>=set<FirmNode>{} ;> next:set<FirmNode>=set<FirmNode>{} ;>\
    ( [collectConstUsers(dummy, now)] ;>\
        (wavefront(now, next) ;> now.clear() ;> tmp=now ;> now=next\
            ;> next=tmp ;> isEmpty=now.empty() ;> !isEmpty)* ;>\
        [foldAssociativeAndCommutative] ;>\
        (foldCond+ | removeUnreachablePhiOperand+ |\
            removeUnreachableBlock+ | removeUnreachableNode+)* ;>\
        fixEdgePosition* ;>\
        (mergeConsts+ | removeUnusedNode+ | mergeBlocks+)\
    )*
```

Fig. 1. Driver sequence for constant folding.

Initially two empty storage sets are created for the wavefront algorithm: `now` which will contain the users of constant nodes which are visited in this step and `next` which will contain the users of constant nodes which will be visited in the next step. Then the main loop is entered. There the rule `collectConstUsers` is executed on all available matches, collecting all users of `Const` nodes available in the graph into the storage set `now` (the `dummy` variable was never assigned and the `collectConstUsers` rule was specified to search for a valid entitiy if the first parameter is undefined). This is the initial stone thrown into the water of our program graph. From it on a wavefront is iterated following the data flow edges until it comes to a halt beause no new constants were created any more. A wavefront step is encapsulated in the subsequence `wavefront` given in Figure 2; it visits the operations given in the set given as its first argument, and adds the operations which are users of the constants it created newly with constant folding

```
def wavefront(constUsersNow:set<FirmNode>,\
              constUsersNext:set<FirmNode>) {\
        for{\
                cu:FirmNode in constUsersNow;\
                ((c)=foldNot(cu) || (c)=foldCmp(cu) ||\
                 (c)=foldBinary(cu) || (c)=foldPhi(cu) & false\
                )\
                && [collectConstUsers(c, constUsersNext)]\
        } ;> constUsersNow.clear()\
}
```

Fig. 2. Wavefront step for constant folding.

into the set given as its second argument. After this subsequences was called on the `now` argument the `now` and `next` sets are swapped (the set referenced by `now` is cleared, `now` is assigned the set referenced by the `next`, and next is assigned the empty set previously pointed to by `now`). The wavefront iteration is stopped when the `now` set of the next iteration step is empty.

Let us have a look at a wavefront step defined by the subsequence given in Figure 2: it iterates with a `for` loop over the users of constants contained in the storage set parameter `constUsersNow`, binding them to an iteration variable `cu` of type `FirmNode`. This constant is used as input to the rules `foldNot`,`foldCmp`,`foldBinary` and `foldPhi` really doing the folding in case all of the arguments to the operations they handle are constant. They return the newly created constant by folding, the rule `collectConstUsers` then adds all users of this constant to the `constUsersNext` storage set.

Figure 3 shows our constant folding rule for `Binary` operations. It takes the node with the `Const` operand as parameter and returns the folded `Const`. The rule matches both `Const` operands and has an `alternative` statement that contains one case for each binary operation (except `Cmp`). Within the cases the value of the new `Const` is computed. Finally, at the end of rule execution, the `exec` statement relinks the users of the `Binary` to the newly created `Const` and deletes the `Binary` from the graph.

For the constant folding of a `Cmp` we use a separate rule shown in Figure 4. It uses a complex `evaluation` to compute the resulting value of the `Const`.

The constant folding for unary `Not` nodes is straight forward and omitted; n-ary `Phi` nodes are of interest again. Figure 5 shows the corresponding rule. It first matches the `Phi` node and one `Const` operand and then iteratively edges to the same operand and to the `Phi` itself. If this covers all edges we can replace the `Phi` with the `Const` operand.

```
rule foldBinary(b:Binary<FirmNode>) : (Const)
{
  if{ typeof(b)!=Cmp; }

  startBlock:StartBlock;
  b -df0:Dataflow-> c0:Const;
  b -df1:Dataflow-> c1:Const;
  hom(c0, c1);
  if{ df0.position==0 && df1.position==1; }

  alternative {
    foldAdd {
      if{ typeof(b)==Add; }

      modify {
        eval { c.value = c0.value + c1.value; }
      }
    }
    ...
  }

  modify {
    c:Const -startBlockEdge:Dataflow-> startBlock;
    eval { startBlockEdge.position = -1; }
    exec([relinkUser(b,c)] ;> deleteNode(b));
    return(c);
  }
}
```

Fig. 3. Rule for folding `Binary` nodes.

```
rule foldCmp(cmp:Cmp<FirmNode>) : (Const)
{
  startBlock:StartBlock;
  cmp -blockEdge:Dataflow->;
  cmp -df0:Dataflow-> c0:Const;
  cmp -df1:Dataflow-> c1:Const;
  hom(c0, c1);
  if{ df0.position == 0 && df1.position == 1; }

  modify {
    delete(blockEdge, df0, df1);
    c:Const<cmp>;
    c -startBlockEdge:Dataflow-> startBlock;
    eval {
      c.value = (
        cmp.relation == Relation::TRUE ||
        cmp.relation == Relation::GREATER
          && c0.value >  c1.value ||
        cmp.relation == Relation::EQUAL
          && c0.value == c1.value ||
        cmp.relation == Relation::GREATER_EQUAL
          && c0.value >= c1.value ||
        cmp.relation == Relation::LESS
          && c0.value <  c1.value ||
        cmp.relation == Relation::NOT_EQUAL
          && c0.value != c1.value ||
        cmp.relation == Relation::LESS_EQUAL
          && c0.value <= c1.value
      ) ? 1 : 0;
      startBlockEdge.position = -1;
    }
    return(c);
  }
}
```

Fig. 4. Rule for folding `Cmp` nodes.

```
rule foldPhi(phi:Phi<FirmNode>) : (Const)
{
  startBlock:StartBlock;
  phi -blockEdge:Dataflow-> block:Block;
  phi -e0:Dataflow-> c0:Const;

  iterated {
    phi -e:Dataflow-> c0;

    modify {
      delete(e);
    }
  }

  iterated {
    phi -e:Dataflow-> phi;

    modify {
      delete(e);
    }
  }

  negative {
    c0;
    block;
    phi -:Dataflow-> :FirmNode;
  }

  modify {
    delete(blockEdge, e0);

    c:Const<phi>;
    c -startBlockEdge:Dataflow-> startBlock;

    eval {
      c.value = c0.value;
      startBlockEdge.position = -1;
    }

    return(c);
  }
}
```

Fig. 5. Rule for folding `Phi` nodes.

3.2 Control flow and cleanup

Let us mentally return to the driver sequence in Figure 1; after the wavefront which folded alongside data flow has collapsed, execution continues with folding condition nodes at the interface of data flow to control flow and with folding control flow proper (jumps and blocks). In addition there are some clean up task left to be executed. If no control flow was folded leading to further data flow folding possibilities, the main loop is left.

```
rule foldCond
{
  startBlock:StartBlock;
  cond:Cond -blockEdge:Dataflow->;
  cond -df0:Dataflow-> c0:Const;
  falseBlock:Block -falseEdge:False-> cond;
  trueBlock:Block -trueEdge:True-> cond;
  hom(falseBlock, trueBlock);

  alternative {
    TrueCond {
      if { c0.value == 1; }

      modify {
        delete(falseEdge);
        -jmpEdge:Controlflow<trueEdge>->;
      }
    }
    FalseCond {
      if { c0.value == 0; }

      modify {
        delete(trueEdge);
        -jmpEdge:Controlflow<falseEdge>->;
      }
    }
  }

  modify {
    delete(df0);
    jmp:Jmp<cond>;
  }
}
```

Fig. 6. Rule for folding `Cond` nodes.

The rule shown in Figure 6 is responsible for folding `Cond`itional jumps. Depending on the value of the constant, either the edge of type `True` gets deleted and the edge of type `False` retyped to an edge of type `Controlflow`, or the edge of type `False` gets deleted and the edge of type `True` retyped to an edge of type `Controlflow`. Since there is only one jump target left, we also retype the conditional jump to a simple jump of type `Jmp`.

Due to the folding of condition jumps, there may be unreachable blocks which can be deleted. We use two rules to remove unreachable code: the rule shown in Figure 7 deletes unreachable `Block`s, the rule shown in Figure 8 deletes nodes without a `Block`.

```
rule removeUnreachableBlock
{
  block:Block\StartBlock;

  negative {
    block -cfgEdge:Controlflow->;
  }

  modify {
    delete(block);
  }
}
```

Fig. 7. Rule for removing unreachable `Block`s.

```
rule removeUnreachableNode
{
  n:FirmNode\Block;

  negative {
    n -blockEdge:Dataflow-> :Block;

    if { blockEdge.position == -1; }
  }

  modify {
    delete(n);
  }
}
```

Fig. 8. Rule for removing nodes without a `Block`.

Furthermore, we also need to adapt `Phi` nodes if they have an operand without a `Controlflow` counterpart in the `Block`. Figure 9 shows the corresponding rule that matches a `Phi` and deletes an operand that has no `Controlflow` counterpart. After the deletion we fix the position of all edges using the `fixEdgePosition` rule.

```
rule removeUnreachablePhiOperand
{
  phi:Phi -blockEdge:Dataflow-> block:Block;
  phi -operandEdge:FirmEdge-> operand:FirmNode;

  negative {
    block -cfgEdge:Controlflow->;

    if { cfgEdge.position == operandEdge.position; }
  }

  modify {
    delete(operandEdge);
  }
}
```

Fig. 9. Rule for removing nodes without a `Block`.

The solution contains two rules that simplify the graph:

removeUnusedNode Removes a node that is not used, i.e. that has no incoming edges. This rule is similar to the `removeUnreachableBlock` rule.

mergeBlocks If there is a `Block block1` with only one outgoing edge of type `Controlflow` and this edge leads to a node of type `Jmp` contained in `Block block2`, then we remove the `Jmp` and merge `block1` and `block2`. This rule is able to remove the chain of "empty" `Block`s that occurs in the running example of the case description.

The verifier mentioned in the case descpription was implemented with the tests in `Verifier.gri`, called from a subsequence `verify` defined in `Verifier.grsi`; the subsequence is used with the statement `validate xgrs verify` before and after the transformations checking the integrity of the graph.

3.3 Folding More Constants

Figure 10 shows a rule that fold constants for the term $(x * c1) * c2$ where $*$ is a associative and commutative operation. This rule enables us to solve the test cases provided by the GReTL solution. In contrast to the GReTL rule, this rule does not change the structure of graph.

```
rule foldAssociativeAndCommutative
{
  bin0:Binary;
  bin0 -:Dataflow-> block:Block;
  bin0 -binEdge:Dataflow-> bin1:typeof(bin0);
  bin0 -constEdge:Dataflow-> c0:Const;
  bin1 -:Dataflow-> operand:FirmNode\Block;
  bin1 -:Dataflow-> c1:Const;
  hom(c0, c1);

  if { bin0.associative && bin0.commutative; }

  modify {
    delete(binEdge, constEdge);

    bin:typeof(bin0);
    bin -blockEdge:Dataflow-> block;
    bin -left:Dataflow-> c0;
    bin -right:Dataflow-> c1;
    bin0 -binLeft:Dataflow-> operand;
    bin0 -binRight:Dataflow-> bin;

    eval {
      blockEdge.position = -1;
      left.position = 0;
      right.position = 1;
      binLeft.position = binEdge.position;
      binRight.position = constEdge.position;
    }
  }
}
```

Fig. 10. Rule for constant folding of associative and commutative operations.

4 Instruction Selection

Instruction selection is the task that transforms the IR (intermediate represenation) into a target-dependent representation (TR). It can be considered as some kind of model transformation.

The TR supports immediate instructions, i.e. a `Const` operand can be encoded within the instruction. Our solution consists of one rule per target instruction which means we have at most two rules for each IR operation: one for the immediate variant and on for the variant without immediate.

Before we start matching all immediate operations, we apply the rule shown in Figure 11. It matches commutative operations with a `Const` operand at `position` 0, but not at `position` 1, and then exchanges these operands to ensure that the constant operand is at `position` 1. Thus, the rule ensures deterministic behaviour in case of two constant operands.

```
rule NormalizeConsts
{
  bin:Binary;
  bin -op0Edge:Dataflow-> op0:Const;
  bin -op1Edge:Dataflow-> op1:FirmNode\Const;

  if { bin.commutative && op0Edge.position == 0 && op1Edge.position == 1; }

  modify {
    eval {
      op0Edge.position = 1;
      op1Edge.position = 0;
    }
  }
}
```

Fig. 11. Rule for normalizing commutative operations with a `Const` operand.

Figure 12 shows exemplarily the rule that creates an `AddI` operation. It retypes the `Add` to an `TargetAddI`, deletes the `Dataflow` edge to the constant and stores the value of the `Const` node in the `value` attribute. Since the other rules for immediate operations are very similar, we omit them here.

Instead we want to introduce the rule for creating `TargetLoad`s given in Figure 13. Again, we retype the original operation to the corresponding target operation. Since `Load` is a memory operation, we also need to set the `volatile` attribute.

The instruction selection rules are applied by the sequence shown in Figure 14. The `[rule]` operator matches all occurrences of the given `rule`. Thus, we first create all immediate operations and then non-immediate operations for the remaining nodes.

```
rule ToTargetAddI
{
  a:Add -df:Dataflow-> c:Const;
  if { df.position==1; }

  modify {
    ta:TargetAddI<a>;
    delete(df);
    eval {
      ta.value = c.value;
    }
  }
}
```

Fig. 12. Rule for an `TargetAddI` operation.

```
rule ToTargetLoad
{
  l:Load;
  modify {
    t:TargetLoad<l>;
    eval { t.volatile = l.volatile; }
  }
}
```

Fig. 13. Rule for an `TargetLoad` operation.

```
xgrs [NormalizeConst]
xgrs [ToTargetLoadI] | [ToTargetStoreI] | [ToTargetAddI] |\
   [ToTargetSubI]  | [ToTargetMulI]   | [ToTargetDivI] |\
   [ToTargetModI]  | [ToTargetAndI]   | [ToTargetOrI]  |\
   [ToTargetEorI]  | [ToTargetShlI]   | [ToTargetShrI] |\
   [ToTargetShrsI] | [ToTargetCmpI]

xgrs [ToTargetJmp]   | [ToTargetCond]  | [ToTargetLoad]     |\
   [ToTargetStore] | [ToTargetConst] | [ToTargetSymConst] |\
   [ToTargetNot]   | [ToTargetAdd]   | [ToTargetSub]      |\
   [ToTargetMul]   | [ToTargetDiv]   | [ToTargetMod]      |\
   [ToTargetAnd]   | [ToTargetOr]    | [ToTargetEor]      |\
   [ToTargetShl]   | [ToTargetShr]   | [ToTargetShrs]     |\
   [ToTargetCmp]
```

Fig. 14. Extended graph rewrite sequence for instruction selection.

5 Conclusion

We presented a GrGen solution for both tasks: constant folding as well as instruction selection. The performance for the largest GReTL test case – consisting of 27993 nodes and 55981 edges – is 6.3 seconds for constant folding and 111 milliseconds for instruction selection. For the largest test shipped with this case – consisting of 277529 nodes and 824154 edges – we need 11 seconds for constant folding and 1.2 seconds for instruction selection. The measurements were made using Ubuntu 11.04 and Mono 2.6.7 on a Core2Duo E6550 with 2.33 GHz.

References

1. Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net (April 2011), http://www.grgen.net
2. Buchwald, S., Jakumeit, E.: A GrGen.NET solution of the Hello World Case for the Transformation Tool Contest 2011 (2011)
3. Buchwald, S., Jakumeit, E.: Compiler Optimization Case. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/cases/ttc2011_submission_5.zip (2011), http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/cases/ttc2011_submission_5.zip
4. Mazanek, S.: Hello World! An Instructive Case for TTC. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/cases/ttc2011_submission_3.pdf (2011), http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/cases/ttc2011_submission_3.pdf