Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

A Propagation Engine for GCC

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

Several analyses and transformations work by propagating known values and attributes throughout the program. In this paper, I will describe a generalization of the propagation algorithm used by SSA-CCP and its application to copy propagation and value range propagation.

1 Introduction

Several compiler transformations are based on the concept of propagation. Values or attributes generated at various producing sites are propagated into other sites consumers of those values. For instance, in constant propagation we are interested in replacing loads from registers or memory into direct constant references.

The most commonly used algorithm for constant propagation in SSA form is known as SSA-CCP (Conditional Constant Propagation) [3]. The basic idea is very simple, constants are propagated by simulating the execution of the program while keeping track of constant assignments. Since the program is in SSA form, every constant assignment of the form $N_i = CST$ is recorded in an internal table indexed by SSA index i. During simulation, uses of N_i are replaced with CST, if that yields another constant value, the new constant value is used for further propagation. The simulation includes keeping track of conditional predicates, if they are deemed to have a statically computable value, the predicted branch is simulated, and the others ignored. Once simulation stops, the values stored in the constant value table are replaced in the program, expressions folded and the control flow graph adjusted to account for predicted branches.

This paper describes a generalization of this basic simulation scheme used by SSA-CCP so that it can be used in other transformations that can be expressed in terms of value propagation. The paper also describes some applications of this *propagation engine* to copy propagation and value range propagation. Section 2 describes the propagation engine and its implementation in GCC. Section 3 describes an extension to the traditional SSA based copy propagation that can also propagate copies across conditionals. Section 4 describes an implementation of Value Range Propagation (VRP) in GCC [2] and some infrastructure changes for doing incremental updates to the SSA form.

2 Propagation Engine

Propagation is performed by simulating the execution of every statement that produces interesting values. In this context, an *interesting*

value is anything that the specific implementation is looking to propagate: constants in SSA-CCP, copies in copy propagation, range information in VRP, etc.

Both control and data flow are simulated using two separate work lists: a list of control flow edges (*CFG work list*) and a list of def-use edges (*SSA work list*). Simulation proceeds as follows:

1. Initially, every edge in the CFG is marked not executable and the CFG work list is seeded with all the statements in the first executable basic block. The SSA work list is initially empty.

2. A basic block B is taken from the CFG work list and every statement S in B is evaluated with a call to a user-provided callback function (`ssa_prop_visit_stmt`). This evaluation may produce 3 results:

   `SSA_PROP_INTERESTING`: S produces a value deemed interesting by the callback function and that can be computed at compile time. When this occurs, `ssa_prop_visit_stmt` is responsible for storing the value in a separate table and returning a single SSA name N_i associated to that value[1].

   All the statements with immediate uses of N_i are then added to the SSA work list so that they can also be simulated. Furthermore, if S is a conditional jump and `ssa_prop_visit_stmt` has determined that it always takes the same edge E, then only the basic block reachable through E is added to the CFG work list.

   If S is not a conditional jump, or if S is a conditional jump whose value cannot be determined, all the immediate successors of B are added to the CFG work list.

   `SSA_PROP_NOT_INTERESTING`: Statement S produces nothing of interest and does not affect any of the work lists. The statement may be simulated again if any of its input operands change in future iterations of the simulator.

   `SSA_PROP_VARYING`: The value produced by S cannot be determined at compile time and further simulation of S is not needed. If S is a conditional jump, all the immediate successors of B are added to the CFG work list. Once a statement yields a varying value, it is never simulated again.

   Once all the statements in basic block B have been simulated, its statements are not traversed again. Statements are only visited more than once if they are added to the SSA work list when visiting other statements.

3. If block B has any ϕ nodes, they are simulated with a call to the callback function `ssa_prop_visit_phi`. As opposed to regular statements, ϕ nodes are *always* simulated every time B is added to the CFG work list. This is because ϕ nodes receive their inputs from different incoming edges, so every time a new edge is marked executable, a new argument of each ϕ node will become available for simulation.

   It is up to `ssa_prop_visit_phi` to only consider ϕ arguments flowing through executable edges (marked with flag `EDGE_EXECUTABLE`). The return value from `ssa_prop_visit_phi` has the same semantics described in 2.

   Also, the evaluation of ϕ nodes is different from other statements. A ϕ node is a merging point of potentially different values from different SSA names. In general, the resulting value of a ϕ node will be the

[1] In some propagation problems it may be useful to allow statements to return more than one interesting name. But the current implementation is limited to just one.

"intersection" of all the incoming values. Each propagator will have a different concept of intersection according to its own lattice value rules.

4. Simulation terminates when both work lists are drained.

For efficiency of implementation, the SSA work list is split in two separate lists: one to hold all the SSA names with a result of `SSA_PROP_VARYING` and another one to hold those with `SSA_PROP_INTERESTING` values. The rationale is that the majority of names will not actually yield interesting values, so it is more efficient to dispose of the varying values by simulating the affected statements as soon as possible.

2.1 Keeping track of propagated values

As discussed earlier, during propagation two user provided functions are called: `ssa_prop_visit_stmt` and `ssa_prop_visit_phi`. The propagator itself is only interested in the three return values to determine which blocks and statements to add in the work lists. However, the real work is in keeping track of propagated values. Every interesting value produced by simulation must be associated to a single SSA name N_i, but the final values must not be replaced in the IL until propagation has finished. During propagation, names may get more than a single value.

Once propagation has finished, final values may be replaced into the IL with a call to `substitute_and_fold`. The only argument it receives is an array *PV* of propagated values indexed by SSA index. If name N_i has final value V then $PV[i] == V$. The call to `substitute_and_fold` is optional, individual users are free to use the final propagated values in any other way.

2.2 Propagating memory loads and stores

SSA names for GIMPLE registers (also known as *real names*) represent a single object, so when the propagator associates a value with a real name N_i, uses of N_i can be replaced directly. On the other hand, an SSA name for partial or aliased stores (also known as *virtual names*) may represent different objects or parts of an object. For instance, given

```
1  # A_3 = V_MAY_DEF <A_2>
2  A[i_9] = 13
3
4  [ ... ]
5
6  # VUSE <A_3>
7  x_3 = A[i_9]
```

Depending on the exact value of i_9 at runtime, different locations of *A* may be used to store 13. However, both the memory store represented by A_3 at line 1 and the subsequent memory load at line 7 are guaranteed to read the same value because they are both loading from array slot i_9.

To support propagation in these cases, the array of propagated values also includes a field denoting what memory expression was used in the store operation that created the associated name. When simulating the memory store to A_3 in line 2, the implementation of `ssa_prop_visit_stmt` in SSA-CCP will associate two things to A_3, namely the value 13 and the memory expression $A[i_9]$.

Once simulation has finished, the call to `substitute_and_fold` will proceed as follows: On finding the VUSE for A_3 at line 7, it will compare the memory load expression on the RHS of the assignment with the memory store expression from line 2. In this case, both

expressions are identical so line 7 will be converted to $x_3 = 13$.

If the propagator is interested in working with memory loads and stores, then it needs to handle them in both the statement and the ϕ simulator. For instance, given the following, admittedly contrived, code snippet

```
1   if (...)
2     # A_4 = V_MAY_DEF <A_3>
3     A[i_3] = 42;
4   else
5     # A_5 = V_MAY_DEF <A_3>
6     A[i_3] = 42;
7   # A_6 = PHI <A_4, A_5>
8
9   if (...)
10    # A_7 = V_MAY_DEF <A_6>
11    A[i_3] = 42;
12  # A_8 = PHI <A_6, A_7>
13
14  # VUSE <A_8>
15  x_9 = A[i_3];
```

When visiting the ϕ node A_6 at line 6, `ssa_prop_visit_phi` will examine ϕ arguments A_4 and A_5. Since they both represent stores to the same memory expression, $A[i_3]$, it will store value 42 and memory expression $A[i_3]$ into A_6. Similarly, the visit to ϕ node A_8 will assign value 42 and memory expression $A[i_3]$ to A_8.

Notice that propagation of memory stores and loads is necessarily slower than propagation of GIMPLE register values because of the additional comparison of memory expressions. Therefore, the "*store*" versions of the propagators are usually implemented as separate passes.

Propagated values are represented using an array indexed by SSA name index. Each element of the array is of type `prop_value_t` defined in `tree-ssa-propagate.h`:

```
struct prop_value_d {
  /* Lattice value.  Each propagator is
     free to define its own lattice and
     this field is only meaningful while
     propagating. It will not be used by
     substitute_and_fold. */
  unsigned lattice_val;

  /* Propagated value.  */
  tree value;

  /* If this value is held in an SSA
     name for a non-register variable,
     this field holds the actual memory
     reference associated with this
     value.  This field is taken from
     the LHS of the assignment that
     generated the associated SSA name. */
  tree mem_ref;
};

typedef struct prop_value_d prop_value_t;
```

To summarize, every propagation algorithm should define three basic elements:

1. An array of values V of type `prop_value_t` indexed by SSA index number.

2. Statement simulation (`ssa_prop_visit_stmt`). Evaluates the expression computed by the statement, if the statement produces an interesting result, it must be in the form of an SSA name N_i. The produced value is stored in $V[i]$ and N_i is returned to the propagator engine so that its def-use edges can be added to the SSA work list.

   If the statement is a conditional jump and it is possible to compute which edge E will be taken, E is returned so that its destination basic block can be added to the CFG work list. Otherwise, all outgoing edges are added to the list.

3. ϕ node simulation (`ssa_prop_visit_phi`). Similar to `ssa_prop_visit_`

`stmt` but the evaluation is a user-defined merge operation of all the values coming in through executable edges.

Once an implementation for `ssa_prop_visit_stmt` and `ssa_prop_visit_phi` exists, propagation is done with a call to `ssa_propagate`.

3 Copy Propagation

Copy propagation in SSA form is, in principle, very simple. Given the assignment $x_5 = y_4$, all we need to do is traverse all the immediate uses of x_5 and replace them with y_4. However, such approach will not be able to propagate copies past ϕ nodes, particularly those involved in loops. Note that it may be debatable whether aggressive copy-propagation is desirable, as this may have negative effects on passes like register allocation (due to increased register pressure), but the current implementation sticks to the simplistic metric of maximizing the number of propagated copies.

3.1 Lattice for copy propagation

Copy propagation can be described as the problem of propagating the *copy-of* value of SSA names. Given

y_4 = z_6;
x_5 = y_4;

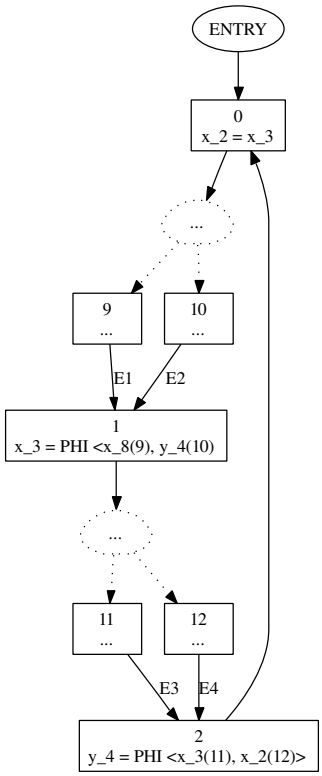
We say that y_4 is a *copy-of* z_6 and x_5 is a *copy-of* y_4. The problem with this representation is that there is no apparent link from x_5 to z_6. So, when visiting assignments in `copy_prop_visit_stmt`, we assign copy-of values instead of the direct copy. If a variable is not found to be a copy of anything else, its copy-of value is itself. So, in this case we would have y_4 copy-of z_6 and x_5 copy-of z_6. At the end of propagation, uses of x_5 and y_4 will be replaced with z_6.

Propagation must also be able to propagate copies exposed by ϕ nodes. For instance,

y_4 = z_6;
x_5 = y_4;
...
z_9 = PHI <x_5, y_4>

Should result in z_9 being a copy of z_6. The implementation of `ssa_prop_visit_phi` only needs to check the copy-of values of every executable ϕ-argument. If they all match, then the LHS of the ϕ node (z_9 in this case) can have its copy-of value set to the common copy-of value. Otherwise, the value of the ϕ node is considered varying and the copy-of value of the name on the LHS is itself. So, when visiting the ϕ node for z_9, the propagator finds x_5 copy-of z_6 and y_4 copy-of z_6, which means that z_9 is copy-of z_6.

The following example shows a more complex situation where copy relations may be obfuscated by loops. Note that the actual visit ordering depends on the shape of the CFG and immediate uses, the ordering used here is meant for illustration only:



1. The first time we visit block 1, edge E1 is marked executable, but edge E2 is not. Therefore, the visit to $x_3 = \phi<x_8(9), y_4(10)>$ results in x_4 copy-of x_8. Since x_3 has changed to a new value, the SSA edges for x_3 are added to the work list ($1 \rightarrow 2$ and $1 \rightarrow 0$).

2. Visit SSA edge $1 \rightarrow 2$: $y_4 = \phi <x_3(11), x_2(12)>$. Assume that edge E3 is marked executable, and edge E4 is marked not executable. This yields y_4 copy-of x_8, because x_3 is copy-of x_8. The SSA edge $2 \rightarrow 1$ for y_4 is added to the work list.

3. Visit SSA edge $1 \rightarrow 0$: $x_2 = x_3$. This yields x_2 copy-of x_8. The SSA edge $0 \rightarrow 2$ for x_2 is added to the work list.

4. Visit SSA edge $2 \rightarrow 1$: $x_3 = \phi<x_8(9), y_4(10)>$. This time both edges E1 and E2 are marked executable. Since x_3 has not changed its copy-of value, no edges are added to the work list.

5. Visit SSA edge $1 \rightarrow 0$: $x_2 = x_3$. The value of x_2 changes to copy-of x_8. Therefore, SSA edge $0 \rightarrow 2$ for x_2 is added to the work list.

6. Visit SSA edge $0 \rightarrow 2$. This time both edges E3 and E4 are marked executable. Since both arguments are copy-of x_8, the value of y_4 doesn't change.

7. Work lists are drained. Iteration stops.

The straightforward implementation of copy propagation, would have needed multiple passes to discover that $x_3 \rightarrow x_8$. But the iterative nature of the propagation engine prevents that. Moreover, this kind of propagation will only iterate over the subset of statements affected, not the whole CFG.

4 Value Range Propagation (VRP)

This transformation is similar to constant propagation but instead of propagating single constant values, it propagates known value ranges. GCC's implementation is based on Patterson's range propagation algorithm [2]. In contrast to Patterson's algorithm, this implementation does not propagate branch probabilities nor it uses more than a single range per SSA name. This means that the current implementation cannot be used for branch prediction (though adapting it would not be difficult).

The current implementation is used to remove NULL pointer checks and redundant conditional branches. For instance, the code in Figure 1 is extracted from a typical expansion of bound checking code in languages like Java. Notice how the bound checking done at line 3 is not really necessary as variable *i* is guaranteed to take values in the range [0, a->len].

```
struct array
{
  const int len;
  int *data;
};

void
doit (array *a)
{
  1  for (int i = 0; i < a->len; ++i)
  2    {
  3      if (i < 0 || (i) >= (a->len))
  4        throw 5;
  5      call (a->data[i]);
  6    }
}
```

Figure 1: Bound checking code generated by the compiler.

Value range propagation works in two main phases:

1. Range Assertions. Some expressions like predicates in conditional jumps, pointer dereferences or taking the absolute value of a variable imply something about the range of values that their result may take. For instance, the expression `if (a_5 > 10) ...` implies that every use of a_5 inside the if will be guaranteed to use values in the range [11, +INF].

   For every expression in this category, the compiler generates a new expression code (`ASSERT_EXPR`) that describes the guaranteed range of values taken by the associated name.

2. Range Propagation. Once `ASSERT_EXPR` instructions have been inserted, the SSA propagation engine is used to evaluate the program. After propagation, every SSA name created by the program will have a range of values associated with it. Those ranges are then used to eliminate conditional jumps made superfluous by the new range information.

4.1 Inserting range assertions

Certain expressions found in the code give us information about the range of values that may be taken by the operands involved in the expression. For instance, consider the code fragment in Figure 2(a).

Since pointer p_4 is dereferenced at line 6, we know that the NULL test at line 8 must always fail. Similarly, the use of a_5 at line 12 is guaranteed to always use the constant value 10. However, we cannot guarantee that **all** uses of p_4 and a_5 will always have a known value. For instance, we have no way of knowing at compile time whether the NULL test for p_4 at line 3 will succeed or not. Similarly, the use of a_5 at line 14 does not use a known value.

The technique used by VRP to overcome this problem is to create new SSA names to which we can pin the range information that we want to propagate. GCC generates a new expression called `ASSERT_EXPR` that captures this information and stores it into a new SSA name. When the compiler finds an expression that contains interesting range information for name N_i, it builds a predicate P describing that range and generates the assignment `N_j = ASSERT_EXPR <N_i, P>`. This expression means that variable N_j has the same value as N_i **and** that value is guaranteed to make predicate P evaluate to *true*.

Therefore, for the code in Figure 2(a), the compiler inserts the assertions found in Figure 2(b). The pointer dereference in line 6 produces the assertion p_5 = `ASSERT_EXPR <`p_4, p_4 != 0`>`. With this conversion, all uses of p_5 are guaranteed to be uses of a non-NULL pointer. Simi-

```
1  p4 = p3 + 1
2
3  if (p4 == 0)
4    return 0
5
6  x10 = *p4
7
8  if (p4 == 0)
9    return 0
10
11 if (a5 == 10)
12   return a5 + x10
13
14 return a5 − x10
```

(a) Before inserting assertions.

```
1  p4 = p3 + 1
2
3  if (p4 == 0)
4    return 0
5
6  x10 = *p4
7  p5 = ASSERT_EXPR <p4, p4 != 0>
8
9  if (p5 == 0)
10   return 0
11
12 if (a5 == 10)
13   a6 = ASSERT_EXPR <a5, a5 == 10>
14   return a6 + x10
15
16 return a5 − x10
```

(b) After inserting assertions.

Figure 2: Preparing the program for Value Range Propagation.

larly, uses of a_6 are guaranteed to use the constant value 10[2].

4.2 Incremental updates of the SSA form

Since range assertion expressions are inserted once the program is in SSA form, it must be updated before ranges are propagated. Each expression N_i = ASSERT_EXPR <N_j, P> creates a mapping from the existing name N_j to the new name N_i.

As assertions are inserted in the IL, a replacement mapping is built. In the example code of Figure 2(b), the compiler will build two mappings, namely $p_5 \rightarrow p_4$ and $a_6 \rightarrow a_5$. Once all the assertions have been inserted, a call to `update_ssa` replaces all the uses of every existing name dominated by the new name.

[2]For equality expressions, GCC generates straight assignments instead of ASSERT_EXPR, but the effect is the same.

The mechanics of the updating process are a little more elaborate than this, but in essence all it does is search and replace inside the sub-regions of the CFG affected by the existing names and their replacements. More details about the replacement process and its API are available in the GCC internal documentation (`http://gcc.gnu.org/onlinedocs/gccint/SSA.html`).

4.3 Propagating ranges

The current VRP implementation uses two range representations:

RANGE $[MIN, MAX]$ to denote all the values that are between MIN and MAX (i.e., N such that MIN <= N <= MAX), and,

ANTI-RANGE $\sim[MIN, MAX]$ to denote all the values that are **not** between MIN and MAX (i.e., N such that N < MIN or N > MAX).

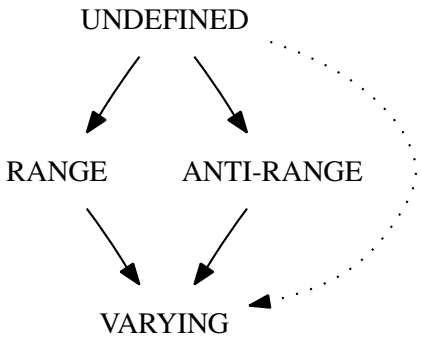


Figure 3: Lattice values used for range propagation.

As opposed to Patterson's formulation, SSA names cannot take multiple disjoint ranges. This was done mainly for simplicity of implementation and compile-time performance[3]. But it would be perfectly feasible to allow names to take disjoint ranges in the future.

The range propagation lattice has 4 values as shown in Figure 3. As is the case with other propagation problems, the only valid transitions are those that move downward in the lattice. If we were to allow transitions in different directions, we would risk infinite loops during propagation.

Lattice values RANGE and ANTI-RANGE are exactly the same in terms of propagation, they both represent known range values for the associated SSA names. The key difference is in the semantics of the actual value when evaluating expressions.

Statements are evaluated by `vrp_visit_stmt`. Two types of statements are considered interesting by the propagator:

1. Assignments of the form N_i = EXPR, where EXPR is of an integral or pointer type. The expression is evaluated and if it results in a useful range, its value is associated to N_i.

   Naturally, the more common sources of useful range information are `ASSERT_EXPR`s, but other expressions may also provide useful ranges. For instance, if EXPR is 42, then we can set the range of N_i to $[42, 42]$. Similarly, expressions involving names with known ranges may yield useful information.

   If scalar evolution information is available for N_i, the computed range is augmented with the bounds computed by N_i's chain of recurrences [1].

2. Conditional branches are also evaluated. If the controlling predicate includes names with known ranges, only the taken edges are added to the CFG work list.

Evaluation of ϕ nodes uses the usual shortcut of ignoring arguments coming through non-executable edges. Given two arguments with ranges VR0 and VR1:

1. If VR0 and VR1 have an empty intersection the resulting range is set to VARYING. Note that if VR0 and VR1 were adjacent, the result could actually be the VR0 $\bigcup$ VR1, but this has not been implemented at the time of this writing.

2. Otherwise, the resulting range is VR0 $\bigcup$ VR1.

Propagation continues while names change from one state to the other. Once all the basic blocks have been simulated and no state transitions occur, simulation stops. The resulting ranges are recorded in the `SSA_NAME_VALUE_RANGE` field of each SSA name and the affected conditional expressions are folded.

[3] In general, I have found the current VRP implementation to be about 4x slower than the CCP pass.

5 Conclusion

This paper describes an abstraction of one of the main components of a commonly used constant propagation algorithm [3]. The basic propagation and simulation done to propagate constants in SSA-CCP can be factored out and re-used for several other transformations that need to propagate values globally.

We have also described two transformations that are based on this generic propagation engine. Several other applications are possible: attributes like string lengths, variable types, bit values, etc. may be propagated using this technique. Some of these applications are either planned or in the process of being implemented.

When implementing a propagation pass using this engine, three basic elements must be defined:

1. A lattice value to control state transitions for SSA names. It is important to only allow transitions in one direction and to limit the depth of the lattice. State transitions that go in different directions may throw the propagator into an infinite loop. Also, deep lattices take longer to converge.

   In most cases, the majority of the values will tend to be varying, so providing a fast path to the varying state speeds up the simulation.

2. An implementation for `ssa_prop_visit_stmt`. This function will receive a statement taken from either the SSA or the CFG work list. If evaluation produces a new value, the name N_i for which that value is produced must be returned so that the propagator can add the SSA edges for N_i to the work list.

   If the statement is a conditional branch and the controlling predicate can be computed to a known value, the corresponding outgoing edge E should be returned. In that case, only E will be added to the CFG work list.

   If a statement is considered varying, the simulator will not schedule any more visits to it.

3. An implementation for `ssa_prop_visit_phi`. Think of this function as a *merge* operation. If all the ϕ arguments that flow through executable edges have compatible values according to the lattice then the result will be an interesting value. Otherwise, the result should be marked varying, in which case this ϕ node will not be visited again.

References

[1] D. Berlin, D. Edelsohn, and S. Pop. High-Level Loop Optimizations for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[2] Jason R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, 1995.

[3] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.