Domination-Based Scoping and Static Single Assignment Languages

M. Anton Ertl*, TU Wien

Abstract. For local variables, domination-based scoping is an alternative to block scoping: the scope of a variable extends from its initializing definition to every place that can be reached only via its definition, ensuring that the variable is initialized everywhere in its scope. The difference between domination-based scoping and block scoping depends on the control structures in the language; domination-based scoping is preferable if the language allows branching into the middle of blocks. Another programming language feature explored in this paper is how to enable static single assignment (SSA) programming. ϕ-functions as in SSA form are confusing to programmers, so we propose an alternative based on passing tuples of values across control flow edges.

1 Introduction

We attack two programming language design problems in this paper:

The first one is the question of what the scope of a local variable should be. Is the traditional block scope always the right thing? What about programming languages that do not fit the block-structured model? Is there an alternative scoping regime, how does it compare to block scoping, and when is it preferable? We answer these questions in Section 2.

The other problem is how to support static single assignment (SSA) programming in an imperative programming language. SSA form has advantages as intermediate representation, and SSA programming would also have advantages when used by humans, but what should the notation for SSA programming be? Should it be a variant of SSA form, or is a different notation preferable? We discuss these issues in Section 3.

2 Domination-based scoping

Many languages encourage or require local variables to be initialized on definition. But the variable is only guaranteed to be initialized at all possible use points if it is not visible at points that can be reached without executing the definition first; in the terminology of control-flow theory, the variable should only be visible at points dominated by the definition [CFR+91]. Conversely, letting

* Correspondence to: M. Anton Ertl, Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; Email: anton@mips.complang.tuwien.ac.at

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: 14. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS '07)

the variable be visiable at all points dominated by the definition is a natural scoping rule.

However, many languages use block scoping. Depending on the control structure, this may lead to the same or different scoping than domination-based scoping.

2.1 Same scope

For several of the most frequently used control structures, the domination-based scope is the same as the block-structured scope, in particular, for the usual `if` and `while` control structures. Variables defined inside the blocks controlled by these control structures are visible until the end of the control structure.

2.2 Domination-based scope larger

A typical example of this case is the C `do`...`while` loop, which is executed at least once.[1] With block-structured scoping, a variable defined inside the loop will be visible until the end of the loop, but the domination-based visibility would extend beyond the end of the loop, until, e.g., the end of an enclosing `if` block.

Another example is a definition in the first clause of the `for` loop in C++ and C99; the block-structured rules of these languages make the variable visible only inside the `for` statement, whereas its domination-based visibility extends beyond the end of the loop. A typical example where domination-based visibility would be useful is a search loop:

```
for (int i=0; i<n; i++)
  if (a[i]==key)
    break;
do something with i /* but i is invisible here in C99 */
```

However, apart from occasional inconveniences such as having to work around this, having the language-defined scope smaller than the domination-based scope poses no problem.

2.3 Domination-based scope smaller

Typical cases where the domination-based scope is smaller than the block scope is when there is a jump from outside of the block to inside the block, behind the definition of the variable. E.g., consider the following C fragment:

```
if (flag)
  goto entry;
```

[1] This style of loop is also known as `REPEAT`...`UNTIL` loop after the well-known Pascal construct.

```
{
  int i=5;
 entry:
  printf("%d",i);
}
```

If `flag` is true, `i` will be uninitialized in the `printf` statement.[2] With domination-based scoping `i` would not be visible there and trying to compile this code would produce an error.

In addition to `goto`, in C the `switch` statement jumps into a block; actually using a block with `switch` is required in practice, and the additional function of the block to control scoping is particularly perverse in this context.

2.4 Experiences

We have implemented (a single-pass approximation of) domination-based scoping in Gforth [Ert94], and our experiences are positive: No problems with this feature were reported from users, and occasionally it has turned out to be useful.

One factor contributing to the decision to use domination-based scoping is that standard Forth supports low-level control-flow constructs that do not fit into a block structure (although they are higher level than `goto`) [Bad90].

2.5 Lessons for future languages

Domination-based scoping is an interesting alternative to the traditional block scoping. The following points should be taken into consideration when deciding which scoping regime to use:

- Block-structured scoping is well-known by programmers, and any alternative requires additional learning effort.
- As long as the programming language only has mostly-structured control-flow constructs, with any jumps only allowed to leave blocks, the block scope will be the same or smaller than the domination-based scope, and uninitialized variables cannot happen. In this case block scoping is probably good enough.
- If the language allows jumping into a block or within a block, then block scoping can lead to uninitialized variables, and domination-based scoping becomes more attractive.

3 Static single assignment programming languages

Initialization of local variables on definition leads to a programming style where many variables are never overwritten, only initialized and later used. One advantage of this style is that, with only one assignment, and the variable guaranteed

[2] Placing the `if`-statement inside the block (before the definition of `i`) would have the same problem.

to be initialized, it is trivial to see where the value in a variable is coming from. This is the advantage that makes static single assignment (SSA) form so popular as intermediate representation in compilers, but it helps humans to understand the program just as much as it helps the machine.

This leads to the question how to enable using this no-overwrite (aka SSA) style in a programming language intended to be used by humans. Functional programming languages require (static and dynamic) single assignment programming, but in this paper we are interested in adding an SSA feature to imperative programming languages, as an option.

3.1 SSA form

An obvious idea is to use a syntax inspired by the papers on SSA form [CFR+91], in particular the ϕ "function". An iterative Fibonacci function definition in a version of C extended in this way might look like this:

```
unsigned fib(unsigned n)
{
  for (;;) {
    unsigned a=phi(0,b);
    unsigned b=phi(1,a+b);
    unsigned i=phi(n,i-1);
    if (i==0)
      break;
  }
  /* a is visible due to domination-based scoping */
  return a;
}
```

In this example, the value of each `phi` is the value of its first argument if the loop was just entered, and the value of its second argument if the start of the loop was reached (again) by looping back from the bottom of the loop. However, the arguments of the `phi` functions are not evaluated there; instead, the first argument of each `phi` is evaluated just before entering the loop, and the second argument is evaluated just before looping back to the start.

The evaluation of the `phi` arguments at a different place than their source code position is probably quite confusing to programmers, so we should instead strive for a syntax that places the source code for these arguments in those positions where they are evaluated.

3.2 Stack-based languages

This happens to be very easy in stack-based languages like Forth and Postscript. There one can evaluate the arguments and put them on the stack across control flow. The Forth implementation Gforth has the appropriate local variable implementation with an approximation of domination-based scoping [Ert94]. Our Fibonacci example looks like this:

```
: fib { n -- n2 }
    0 1 n BEGIN { a b i }
        i 0 <> WHILE
        b  a b +  i 1-
    REPEAT
    a ;
```

Here, at the start the initial values $0, 1, n$ for the variables a, b, i are pushed on the stack, and the variable definition `{ a b i }` at the start of the loop takes them off the stack. At the end of the iteration, the new values $b, a+b, i-1$ (written in stack-based postfix syntax) are computed and pushed on the stack, and the control flow follows the loop back edge (from `REPEAT` to `BEGIN`) to the variable definition, which again consumes the values from the stack.

While this approach is quite satisfactory as far as the single-assignment part and the code positioning is concerned, stack-based languages will probably never become mainstream. So, how might an SSA variant of a mainstream (i.e., Algol-family) language look like?

3.3 Algol-family experiment

To answer this question (and have a new language for our compiler course), we designed a tiny language for our compiler course along these lines.[3] In particular, we introduced `->` as a syntactical element that allows passing a tuple of values across control flow edges as well as for defining variables. Our example in (a slightly extended version of) this language:

```
func fib(n)
  0,1,n -> start -> a,b,i;
     if i=0 then
       a -> exit;
     b, a+b, i-1 ->
  repeat ->
end;
```

We have a `start`...`repeat` loop, and at the start of the loop the variables a, b, i are defined from the value tuples that are evaluated right before the start or right before the end of the loop and passed with `->`.

The loop is left through the `exit`, and the value of a is passed across this control flow edge to be passed out after the `repeat` and used as return value by passing it on to the end of the function. If the language had domination-based scoping[4] and a more flexible syntax for the `exit`, it would probably be clearer to write the a directly in front of the `end` instead of passing it through `exit`.

[3] http://www.complang.tuwien.ac.at/ublu/skriptum/skriptum06.html#tth_sEc6.4

[4] Domination-based scoping would have required too much implementation effort for our compiler course.

One unusual feature of the language that does not come out in the example is that plain assignment-style definitions have the data-flow left-to-right: *expr* `->` *var*; in contrast to the Algol *var* `:=` *expr*. To accommodate the usual conventions, one might also add the Algol assignment syntax.

The grammar for this language proved to be surprisingly hard to design, mainly because the introduction of `->` requires a rethinking of the well-established statement/statement-sequence/block arrangement of Algol-family grammars. We eventually did not have statement-sequences as non-terminal in our grammar. However, a few bugs in the grammar turned up, in particular programs that are syntax errors but were not intended to be, so our grammar may not be a good model to copy, and we recommend further experimentation. However, we don't think that this is a fundamental problem, just one of getting the grammar right.

3.4 Compound variables

For variables containing arrays, structures, and objects, it is sometimes useful to treat them as a single unit, and then one can use the SSA style as for simple atomic variables; but sometimes it is also useful to modify individual components, which can be cumbersome in SSA style. Therefore it is probably a good idea to make SSA style an option rather than forcing it to be used everywhere.

4 Conclusion

Domination-based scoping ensures that the variable always has a defined value, and that a variable is visible wherever it has a defined value. Whether domination-based scoping is preferable to traditional block scoping depends on the control structures that the language features.

Static single assignment programming makes the program more readable by making it obvious where a variable got its value. Its syntactic integration into Algol-family programming languages is not yet completely solved, though.

References

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Ert94] M. Anton Ertl. Automatic scoping of local variables. In *EuroForth '94 Conference Proceedings*, pages 31–37, Winchester, UK, 1994.