An Interprocedural Constant Propagation Package for the `D System`

Nathaniel McIntosh

February 16, 1996

Contents

Chapter 1

Description

1.1 Abstract

This document describes the implementation of an interprocedural constant propagation package recently added to the `D System`. We give some background on the IP constants problem and outline the challenges it presents, then present an overview of the `D System` implementation.

1.2 Introduction

Interprocedural constant propagation seeks to determine the set of variables (global variables and formal parameters) for a given procedure that will always have the same value on every invocation of that procedure. Constants obtained in this fashion are used primarily to sharpen other forms of program analysis (dependence analysis, array section analysis, etc).

```
program root
   integer a(100)
   call leaf(a, 1, 10)
   call leaf(a, 2, 10)
end

subroutine leaf(a, lx, ly)
   integer a(100), lx, ly, i
   do i = 3, ly
      a(i) = a(lx) + 1
   enddo
end
```

Figure 1.1 A simple example

For example, consider the program in Figure 1.1. On every invocation of the procedure `leaf`, a value of 10 is passed to the third parameter. Interprocedural constant propagation seeks to discover facts such as these.

1.3 Outline

In Section 1.4 we give an overview of IP constant propagation. We briefly discuss previous research on the problem, we outline the various components of the analysis, and we discuss two of the main obstacles to efficient computation of constants (side effects and the so-called "invisible globals" problem). In Chapter 2, we describe the actual implementation in the `D System`.[1].

1.4 Analysis overview

1.4.1 Previous research

Interprocedural constant propagation has been studied by a number of researchers, many of them here at Rice [CCKT86, CKT86, MS93, GT93]. Our approach is based primarily on the work of Torczon et al [CCKT86, GT93]. The analysis is split into a local phase and an interprocedural phase. During the local phase, *jump functions* are computed for every call site in the program. During the interprocedural phase, a sweep (or sweeps, if the program is recursive) is made over the call graph, evaluating the jump functions and propagating constants downward from caller to callee.

1.4.2 General goals and requirements

As is traditional for interprocedural analysis, providing a system that supports separate compilation is very important. We address this problem by using a split-phase design, composed of a local phase that is run once for every procedure, and an interprocedural phase that is run for the entire program [CKT86].

1.4.3 Local phase

The local phase for constant propagation examines all of the call sites in the program and computes jump functions for the formal parameters and globals passed at the call. Local information is computed *prior* to any IP scalar mod/ref information.

1.4.4 Interprocedural phase

The interprocedural phase can be thought of in the abstract as operating in a fashion similar to standard intra-procedural constant propagation algorithms. The solver visits each call graph node, and at each call site within the node, the set of constants leaving that call site is calculated by applying the call site's jump functions to the current set of constants reaching the node.

These operations can be described in more details as follows:

1. **Edge translation**: for each edge $e = N \rightarrow M$ entering M, we take the set of constants S reaching N and apply the jump function $F_{N \rightarrow M}$ to produce a new set S_e.
2. **Meet edge summaries**: All of the partially computed constant set S_e for all edges e entering M are "meet-ed" together.

[1]For hints on the order in which to look at source files, see section 2.6

3. **Node test**: The result of the meet is compared with the previous set of constants reaching M to determine if there was any change.

This process is repeated at each node until there is no further change.

1.4.5 Side effects

One of the chief problems that comes up when implementing IP constants is dealing with side effects of procedures. Unless one chooses to compute the local info for a procedure after interprocedural scalar MOD/REF analysis is already completed, one has to assume that a procedure call can modify any global scalar value (or any reference actual). This means that jump functions have to encapsulate all possible side effects from calls. Consider the example program shown in Figure 1.2.

```
program root
   common /foo/ i1
   integer i1
   i1 = 9
   call clobber
   call leaf(10)
end
```

```
subroutine clobber
   common /foo/ i1
   integer i1
   read *, i1
end

subroutine leaf(x)
   ...
end
```

Figure 1.2 Mod/ref example

Suppose we want to compute the local information (i.e. the jump functions) for the call sites in `root`. The jump functions for the call to `clobber` are easily computed. The jump functions for the call to `leaf` are more complicated. In particular, the problem is that one cannot tell just by looking at `root` whether the call to `clobber` overwrites the value given to `i1` prior to the call.

There are two ways to solve this problem. The first is to only compute jump functions *after* scalar IP mod is available. This simplifies the computation of the jump functions, but slows down the analysis (i.e. more passes over the program are required). It also interferes with the goal of providing separate compilation: if a change is made to a single function, it means that the local info for all the functions has to be potentially recomputed.

The second way to deal with this problem is to build a "conditional" jump function that incorporates the MOD decision (i.e. "if `i1` is modified at the call to `clobber` then `bottom` else `9`"). This increases the complexity of the jump functions, but allows separate compilation to be preserved, and reduces the number of passes to compute local info.

1.4.6 The "invisible" global problem

A second problem that causes major headaches is the so-called "invisible global" problem. This refers to the fact that within a given procedure, not all global variables may be declared. Consider the example program shown in Figure 1.3.

```
program root
   common /foo/ x, y
   integer x, y
   y = 9
   call func1
   call func2
end

subroutine func1
   call func2
   call leaf
end

subroutine func2
   common /foo/ q, r
   integer q, r
   q = 10
   read *, r
   call leaf
end

subroutine leaf
   ...
end
```

Figure 1.3 Invisible global example

Suppose we want to compute the jump functions for the call to `leaf` in the subroutine `func1`. There are no global variables declared at all in this procedure, however this doesn't mean that there can't be global constants that are discovered to flow into `func1` during the interprocedural phase. We can't simply produce *no* jump functions for this call site – instead we need a sort of an "anonymous" jump function that models the flow of some unknown global. Here the anonymous jump function would have to take into account that a constant entering `func1` might be clobbered by the call to `func2` before reaching the call to `leaf`.

Chapter 2

Implementation

In this chapter we describe the actual implementation of the IP constants package within the `D System`.

2.1 Jump functions

In our implementation, a jump function is simply a `ValNumber`. The value numbering package is well-suited for this purpose, since a `ValNumber` provides a good way to encapsulate an arbitrary integer expression of formals and globals. In addition, the value numbering package provides mechanisms for capturing possible side effects of procedure calls (via the `VAL_RETURN` value).

2.1.1 Invisible globals

Invisible globals are handled in our implementation using a mechanism in the SSA package that we call the "`DUMMY_GLOBAL`" variable. In addition to tracking all of the variables that are actually used within a procedure, the SSA package also builds phantom `IP_MOD` and `IP_REF` nodes at call sites for a nonexistent variable called `DUMMY_GLOBAL`. This variable can then be used to compute values for invisible globals during the IP phase.

For example, returning to the program in Figure 1.3, consider the call to `leaf` within the function `func1`. During the local phase, we ask for a `ValNumber` for the variable `DUMMY_GLOBAL` at the call site, and we store this value as part of the local info.

During the interprocedural phase, we can then use this `DUMMY_GLOBAL` value to compute real values for globals that are discovered during the interprocedural phase. For example, suppose that while visiting `func1`, we discover that the global /foo/0/4/ has a constant value upon entry to the function. We can then combine the `DUMMY_GLOBAL` value with the available scalar IP mod information to determine whether this particular global will reach the call to `leaf`.

2.2 Local phase implementation

The class `ConstantsModAttr` (see section 2.4.9) encapsulates the local phase for IP constant propagation. It contains the method `ConstantsModAttr::ComputeUpCall`, which computes a `ProcConstantsInfo` (see section 2.4.8) object for each procedure in the given Fortran source file. `ComputeUpCall` in turn calls the helper routine `ComputeProcConstantsInfo`, which then calls the helper routine `CollectCallsiteJumpFunctions` on every call site.

2.2.1 ComputeProcConstantsInfo

This routine is responsible for building the local info for a single function.

2.2.2 CollectCallsiteJumpFunctions

This routine is the heart of the IP constants local phase. It contains code for dealing with actuals and globals. For actuals, it simply demands the value number for the expression in each actual position. Globals are a little more tricky: here we walk the SSA graph and find all of the `IP_REF` nodes hanging off the call site– these nodes correspond to possible modifications of global variables. For each suitable `IP_REF` node, we demand a value from the value numbering package. It should also be added that we look only for integer scalars; other variables are screened out. The value numbers produced are "stored" by installing them into a `ValIPMgr` object.

The two main abstractions used to hold jump function values are `ValVector` (used for the values passed at actuals) and `GlobalValSet` (used for the values passed in global variables). As its name suggests, `ValVector` is simply a vector of value numbers, whereas `GlobalValSet` is a set of global variables (in offset/length representation), each with an associated value.

The final result for each call site is a `ConstantsCallSiteContext`, which is basically just a container for a `ValVector` and a `GlobalValSet`.

2.3 Interprocedural phase implementation

The most important class in the interprocedural phase is `ConstantsDFProblem` (see section 2.4.2). It contains the guts of the interprocedural phase, and it orchestrates all of the various tasks that have to be done. `ConstantsDFProblem` inherits from the class `CallGraphFlowInsensitiveDFP`, so in fact much of the code for walking the call graph, storing intermediate results, etc., resides in the base class. The interesting routines in `ConstantsDFProblem` are all methods of `CallGraphFlowInsensitiveDFP` that have been overridden. The most important methods in `ConstantsDFProblem` are described in the following sections.

2.3.1 ConstantsDFProblem::InitializeNode

This routine initializes a `CallGraphNode`; it receives the `ProcConstantsInfo` object computed during the local phase for the function in question and "unpacks" it in preparation for the interprocedural propagation. It creates a new `ValIPMgr` object for the call graph node, and it translates value numbers obtained from the local phase into this new `ValIPMgr`. It then creates a `ConstantsCallSiteContextTable` containing a `ConstantsCallSiteContext` for each call site within the procedure (i.e. the jump functions for each call site).

2.3.2 ConstantsDFProblem::NodeToEdge

This routine is passed a `CallGraphEdge` ($E = X \rightarrow Y$) and a pointer to the completed `ConstantsAnnot` for the node at the source of the edge (X). It looks up the `ValVector` and `GlobalValSet` for the edge in X's `ConstantsCallSiteContextTable`, and then computes a new `ValVector` and `GlobalValSet` that correspond to the constants arriving at Y, and stores then in a new `ConstantsAnnot` temporary annotation.

2.3.3 ConstantsDFProblem::Meet

Given a call graph node N with in-edges $X \rightarrow N$, $Y \rightarrow N$, ..., $Z \rightarrow N$, this routine "meets" together the constants at each edge to produce the final set of constants at the node N. The solver implements the multi-way meet by calling `ConstantsDFProblem::Meet` once for each in-edge, passing it the edge annotation (computed previously by `ConstantsDFProblem::NodeToEdge`) and an "accumulator" annotation to store the result of the meet.

2.3.4 ConstantsDFProblem::FinalizeNodeAnnotation

Once the solver has determined that the analysis complete, this routine installs the final annotation for each node.

2.4 Important C++ Classes

The following is a list of the important classes that are used during array section analysis. Most of these classes are primarily container classes; they hold the results from some particular pass of the analysis.
or interprocedural).

2.4.1 ConstantsAnnot

This class inherits from `DataFlowSet`; it is an annotation that is placed on `CallGraphNode`s. It serves mainly as a container for a `FormalValVector` (corresponding to the constants reaching each formal) and a `GlobalValSet` (corresponding to the global variables that have constant values on entry to the procedure).

2.4.2 ConstantsDFProblem

This class inherits from `CallGraphFlowInsensitiveDFP`. Its job is to take a `CallGraph` and the local info for the constants problem and run the interprocedural phase for IP constants, and then annotate the call graph with `ConstantsAnnot` annotations. It contains most of the driver code for the interprocedural phase of array sections.

2.4.3 ValVector

This class is simply a container for a fixed-length vector of `ValNumber`s. It is used to model the values passed in the actuals at a call site, or to store the values reaching the formals of a particular routine.

2.4.4 FormalValVector

This class contains information on the constants reaching a particular function. It contains a `ValVector` to hold the values for each formal, and a `StringVector` to hold the formal names (primarily for debugging).

2.4.5 GlobalValSet

This class models a set of global variables in [name/offset/length] representation, with each global having an associated `ValNumber`.

2.4.6 ConstantsCallSiteContext

This class stores information about a particular call site. It inherits from `WordObjectIO`. It contains four items: the node number of the call site in question (this is stored implicitly as `WordObjectIO::id`), a `ValVector` corresponding to the values passed at each actual, a `GlobalValSet` corresponding to the values of globals passed at the call site, and finally a single `ValNumber` corresponding to the value for the `DUMMY_GLOBAL` variable reaching the call site (see section 2.1.1 for more on `DUMMY_GLOBAL`).

During local analysis, this object is stored inside an `ConstantsCallSiteLocalInfo` object. When local analysis is complete and the interprocedural phase is taking place, it is subsequently kept within a `ConstantsCallSiteContextTable` that is part of `ConstantsDFProblem`.

2.4.7 ConstantsCallSiteLocalInfo

This class inherits from `CallSiteLocalInfo`. Its job is simply to contain an `ConstantsCallSiteContext` object.

2.4.8 ProcConstantsInfo

This abstraction stores all of the local information for a given function. It contains a `ValIPMgr` and (implicitly) a table of `ConstantsCallSiteLocalInfo` objects, one for each call site.

2.4.9 ConstantsModAttr

This class is the module attribute for the array sections problem. Its primary job is to provide demand-driven computation of `ProcConstantsInfo` objects for a given Fortran source file (i.e. the local phase). It inherits from the class `ModuleLocalInfo`.

2.5 Source files

2.5.1 Shared abstractions/classes

```
libs/ipAnalysis/ipInfo/ConstantsCallSite.C
libs/ipAnalysis/ipInfo/ConstantsCallSite.h
libs/ipAnalysis/ipInfo/ProcConstantsInfo.C
libs/ipAnalysis/ipInfo/ProcConstantsInfo.h
libs/ipAnalysis/ipInfo/ValVector.C
libs/ipAnalysis/ipInfo/ValVector.h
libs/ipAnalysis/ipInfo/VarValSet.C
libs/ipAnalysis/ipInfo/VarValSet.h
```

2.5.2 Local phase code

```
libs/fileAttrMgmt/analysisAttributes/ConstantsModAttr.C
libs/fileAttrMgmt/analysisAttributes/ConstantsModAttr.h
libs/fileAttrMgmt/analysisAttributes/ComputeConstants.C
libs/fileAttrMgmt/analysisAttributes/ComputeConstants.i
```

2.5.3 Interprocedural phase code

```
libs/ipAnalysis/problems/sections/ConstantsAnnot.C
libs/ipAnalysis/problems/sections/ConstantsAnnot.h
libs/ipAnalysis/problems/sections/ManageConstantsAnnots.C
libs/ipAnalysis/problems/sections/ConstantsDFProblem.C
libs/ipAnalysis/problems/sections/ConstantsDFProblem.i
libs/ipAnalysis/problems/sections/ConstantsDebug.C
libs/ipAnalysis/problems/sections/ConstantsDebug.h
```

2.6 Familiarizing yourself with this code

The best order in which to look at the source files in this package is probably as follows:

1. take a look at `libs/fileAttrMgmt/analysisAttributes/ComputeConstants.C`, particularly the function `CollectCallsiteJumpFunctions`, since it has most of the meat of the local phase.
2. then take a quick look at the interfaces for classes `ValVector` and `GlobalValSet`, since they encapsulate the information computed during the local phase.
3. move on to `libs/ipAnalysis/problems/constants/ConstantsDFPProblem.C`, starting with `ConstantsDFProblem::InitializeNode`, then `ConstantsDFProblem::NodeToEdge`, and finally `ConstantsDFProblem::Meet`.

2.7 Debugging aids

There is a fair amount of code in this package that produces debugging output. This debugging code can be enabled/disabled from the command line, using an environment variable:

`DSYSTEM_IPCONSTANTS_DEBUG`

The value of this variable controls the level of debugging output produced. The base debugging level is 1 – this will produce `printf`'s showing what portions of the code are being invoked and what procedures are being analyzed. See the file `libs/ipAnalysis/problems/constants/ConstantsDebug.h` for more on the various settings. Note that the value of this variable is interpreted in hex, not in decimal.

2.8 Bugs and limitations

This section describes some of the outstanding bugs, limitations, and restrictions of the current implementation.

1. constants from block data subprograms are not currently incorporated. This is mainly since I haven't had the chance to dig through the F77 standard and figure out the semantics of block data subprograms.

2. although the algorithm used in this package handles recursive programs, the implementation relies on the use of the `ValTransAnnot` IP package to translate value numbers interprocedurally, and as a result it can't handle recursive programs (since `ValTransAnnot` can't handle recursion).

3. functions with multiple entry points are not handled (this is similar to the restriction above – the problem lies with the `ValTransAnnot` code).

4. the precision of the analysis when dealing with call sites within loops is somewhat poor, due primarily to the local symbolic analysis.

5. our implementation only provides a single level of "return" jump functions. In other words, suppose we have a routine X that sets a bunch of globals (or formals) to constant values when you call it. In our implementation, if function A calls function B calls function X, then the constants created by the call to X will be "visible" to the code following the call to X in B, but will not be visible to the code in A following the call to B. This limitation is due to the the way we build return jump functions.

2.9 Future improvements

One strategy that could be used to improve the precision of this analysis would be to develop a framework in which SSA graphs for the procedures being analyzed could be built in the presence of IP scalar mod/ref information. This would be a reasonably large undertaking, however, since it would have to be integrated into the demand-driven annotation/attribute framework fairly carefully.

The current shortcomings related to call sites within loops could probably be ameliorated by adding additional information to VAL_VAR_MU values. One way to handle most common cases would be to post-process VAR_MU's by checking to see whether the cycle in the SSA graph that causes them is composed entirely of IP_MOD, IP_REF, and PHI nodes– if this is the case, then (provided that the variable is unmodified at all the call sites in the loop) the recusive value could be replaced by value of the initial edge into the cycle. I've written a preliminary version of some code to do this, but I haven't hooked it in.

The precision of the return jump functions in our implementation could probably be improved. It would require adding an additional upward pass over the call graph to refine the return jump functions (the details are described in [GT93]).

2.10 Acknowledgements

As usual, much of the credit goes to Paul Havlak for the symbolic analysis package he wrote for the `D System`. The value numbering and SSA packages provide an ideal starting point for IP constant propagation.

Credit also is due for the interprocedural analysis framework developed by Mary Hall et al and later refined by John Mellor-Crummey.

Bibliography

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^{\mathrm{n}}$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[GT93] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[MS93] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(1–4):213–232, March–December 1993.