Sparse Bidirectional Data Flow Analysis as a Basis for Type Inference

Jeremy Singer*

February 20, 2004

Abstract

Type inference can be expressed as a bidirectional data flow problem. In this paper we show how to perform type inference using sparse data flow analysis on the static single information form. We infer types for two example programs from previous work, to show that our sparse technique is as effective as, and more efficient than, classical data flow analysis.

1 Introduction

Data flow analysis is the traditional form of program analysis [ASU86]. It is a compile time technique that predicts a safe approximation of program behaviour at run time. Program behaviour is specified in terms of data flow information at program points. A system of simultaneous data flow equations is derived from the program. These equations are solved, generally by an iterative fixed point calculation.

A program is represented as a control flow graph. In this paper we restrict attention to intraprocedural data flow analysis. The scope of data flow analysis is a single procedure, hence program/procedure are interchangeable terms. Nodes in the control flow graph represent basic blocks of consecutive instructions. There is an edge (n_1, n_2) from node n_1 to node n_2 if the flow of execution may proceed directly from n_1 to n_2. There is a unique node `Entry` which has no predecessors, and a unique node `Exit` which has no successors.

Each data flow analysis has an associated direction. A data flow analysis is said to be forwards if the data flow information at each node depends on the data flow information at that node's predecessors. Forwards data flow analysis propagates information in the same direction as the flow of control. Reaching definitions and available expressions are both forwards analyses. A data flow analysis is said to be backwards if the data flow information at each node depends on the data flow information at that node's successors. Backwards data flow analysis propagates information in the opposite direction from the flow of

*University of Cambridge Computer Lab, UK, jeremy.singer@cl.cam.ac.uk

control. Live variables and very busy expressions are both backwards analyses. A data flow analysis is said to be bidirectional if the data flow information at each node depends on the data flow information at that node's predecessors and successors. Bidirectional data flow analysis propagates information both with and against the flow of control.

Partial redundancy elimination (PRE) was originally formulated as a bidirectional data flow problem [MR79]. PRE attempts to remove equivalent computations that occur more than once along a path in the control flow graph. PRE has since been decomposed to a fixed sequence of forwards and backwards (unidirectional) data flow problems [DRZ92, KRS94]. Type inference can be expressed as a data flow analysis [KDM03]. Unlike PRE, type inference cannot be decomposed into a fixed sequence of unidirectional flows, it is a genuine bidirectional data flow problem [KD99].

Classical data flow analysis stores many units of data flow information at each node in the control flow graph. The major feature of sparse data flow analysis is that it stores less data flow information at fewer program points. In this paper, we show how bidirectional data flow problems (exemplified by type inference) can be solved by means of sparse data flow analysis using the static single information form (SSI). The rest of this paper is structured as follows: in section 2 we describe in detail the problem of type inference as a data flow analysis; in section 3 we review SSI and give its formal definition; in section 4 we survey the field of sparse data flow analysis; in section 5 we highlight our contribution, which is applying sparse data flow analysis to bidirectional problems; in sections 6 and 7 we work through two examples of type inference using our sparse technique; we review related work in section 8; finally we make some concluding remarks in section 9.

2 Type Inference

Type inference [Car97] is the process of discovering the derivation of types for terms in a program, within a given type system. Type inference determines the type of a term from the contexts in which it is mentioned in the program. It is also known as type reconstruction. The standard type inference algorithm [Mil78] generates constraints on types which must be solved by unification.

However type inference can also be implemented as data flow analysis. Aho et al [ASU86] treat type inference as a bidirectional data flow problem. They explicitly state that it requires both forwards and backwards propagation of information to obtain precise estimates of possible types. We consider one of their examples in section 6. Aho et al's treatment of the subject is based on the work on Tennenbaum [Ten74] for the SETL programming language. Kaplan and Ullman [KU80] present another early example of type inference using data flow analysis. The most authoritative recent work on the subject is by Khedker et al [KDM03]. We consider one of their examples in section 7. They define the first ever formal data flow framework for type inference, based on bidirectional data flow analysis. They distinguish between statically typed languages,

which require a variable to have the same type throughout the program, and dynamically typed languages, which allow the same variable to hold values of different types in different parts of the program. Mycroft [Myc99] has shown, in the context of decompilation, that dynamic types can often be reduced to static types by transforming to static single assignment form (SSA). Our type inference works on SSI, thus we reap the same benefits as Mycroft—dynamic types are reduced to static types in SSI. Indeed, it appears that a larger class of dynamically typed programs are reducible to statically typed programs by SSI transformation than by SSA transformation.

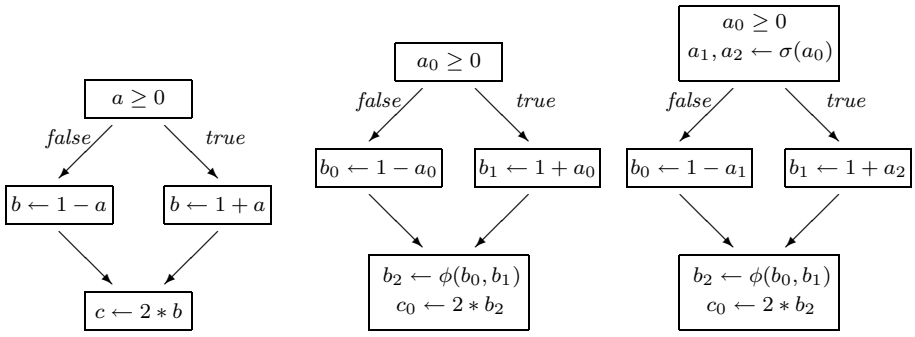
All of these type inference algorithms which are based on data flow analysis are implemented in the classical, non-sparse manner. This paper shows how type inference can be performed using a sparse data flow analysis technique.

3 Static Single Information Form

Static single information form (SSI) is an extension of the well known static single assignment form (SSA) [CFR+91]. The beauty of SSA is that program variables are renamed such that each variable has only one definition site in the program. The most notable feature of SSA is the ϕ-function, a pseudo-assignment which is used to combine multiple incoming variable definitions at control flow merge points, thus (albeit rather artificially) preserving the property that each variable has a unique definition site.

SSI builds upon SSA by adding another pseudo-assignment, the σ-function, which is used to separate variables at control flow split points. In this way it is possible to differentiate between uses of a variable in separate arms of a conditional branch. SSI was originally described by Ananian [Ana99]. He states that "the principal benefits of using SSI form are the ability to do predicated and backwards dataflow analyses efficiently." He gives several examples including very busy expressions analysis and sparse predicated typed constant propagation. Indeed, SSI has been applied to a wide range of problems [RR00, GSR03, AR03]. However to the best of our knowledge, SSI has never previously been applied to a bidirectional data flow problem. In this paper we show that bidirectional data flow analysis can be performed with great efficiency when combined with SSI.

Figure 1(a) shows an example program in standard control flow graph notation. In the SSA notation of figure 1(b), variables are renamed so that each variable has a unique definition site, but the semantics of the program are unchanged. Note that the ϕ-function for b_2 is needed to merge the definitions of b_0 and b_1 in the two arms of the conditional branch. In the SSI notation of figure 1(c), variables are renamed so that each variable has a unique definition site, and that each variable is only mentioned in at most one arm of a multi-way branch. Note that the σ-function for a_1 and a_2 is needed to split a_0 across the conditional branch. In general σ-functions may have to be inserted for any variable, not necessarily the variable mentioned in the conditional branch test as is the case here.



(a) Control flow graph *(b) SSA* *(c) SSI*

Figure 1: One program, three representations

In order to define SSI formally, some auxiliary definitions are required. Node n_1 *dominates* n_2 if every path in the control flow graph from `Entry` to n_2 passes through n_1. Node n_1 *postdominates* n_2 if every path in the control flow graph from n_2 to `Exit` passes through n_1. The *dominance frontier* of a node x is the set of all nodes n_i such that x dominates a predecessor of n_i but does not dominate n_i. The *reverse dominance frontier* of a node x is the set of all nodes n_i such that x postdominates a successor of n_i but does not postdominate n_i. The property of SSI may now be stated as follows:

1. Each variable has a unique definition point in the program text;
2. Every definition of variable v dominates all non-ϕ-function uses of v and all ϕ-function uses of v are on the dominance frontier of that definition;
3. Every use of variable v postdominates all non-σ-function reaching definitions of v and all σ-function definitions of v are on the reverse dominance frontier of that use.

Ananian [Ana99] gives an alternative (more verbose) definition of SSI. Construction of SSI can be performed in $O(EV)$ time, where E is the number of edges in the control flow graph and V is the number of variables in the original program (before SSI renaming). This is worst case complexity—typical time complexity is linear in the program size.

4 Sparse Data Flow Analysis

The goal of sparse data flow analysis is to avoid storing and propagating irrelevant data flow information. Because of this, sparse data flow analysis has the potential to be more efficient and more effective than classical data flow

analysis. Admittedly this is a rather imprecise definition, but there is no clearer consensus on the meaning of sparseness in this context.

General sparse representations completely describe a program's behaviour, that is, it is possible to reconstruct the entire original control flow graph for the program. On the other hand, analysis-specific sparse representations only retain those program statements needed to solve a particular data flow problem [Ruf95]. SSA [CFR+91] and SSI [Ana99] are examples of general sparse representations. The sparse evaluation graph [CCF91, Ram97] is an example of an analysis-specific sparse representation.

In this paper, we devote our attention to general sparse representations. We assume the following model of data flow analysis. There is a vector E, in which units of data flow information (generally elements of some data flow lattice) are stored. Classical data flow analysis associates a distinct unit of data flow information with each variable[1] at each node in the control flow graph. $E_{\text{classical}}$ is a two-dimensional vector indexed by $\langle \text{variable}, \text{node} \rangle$ pairs. Many entries in $E_{\text{classical}}$ will not contain relevant data flow information. It is a waste of space to store such entries, and it is a waste of time to calculate them. In contrast, sparse data flow analysis associates a distinct unit of data flow information only with each variable. Thus sparse data flow information must hold over the entire program. This is referred to as control flow insensitive analysis [MRB95]. E_{sparse} is a one-dimensional vector indexed by $\langle \text{variable} \rangle$. This reduces data flow analysis costs, in terms of both computational time and space. In order to increase the precision of sparse analysis, extra variables are introduced by renaming variables from the original program using a scheme such as SSI. The hope is that E_{sparse} remains smaller than $E_{\text{classical}}$, even after the increase of variable names. Constant propagation [WZ91] is a popular data flow analysis that fits this sparse model.

5 Our Contribution

Our contribution is to demonstrate that SSI can be used for sparse bidirectional data flow analysis. We apply SSI to the problem of type inference, simply because this is the most outstanding bidirectional problem that cannot be decomposed into unidirectional data flow. However it should be possible to devise a sparse data flow analysis using SSI for any bidirectional problem.

SSA enables sparse forwards data flow analysis, such as reaching definitions and constant propagation [WZ91]. However it is well known that SSA cannot support sparse backwards data flow analysis, such as live variables [CCF91]. SSI enables both forwards and backwards sparse analysis [Ana99, Sin03]. However only ϕ-functions are required for forwards analysis, and only σ-functions are required for backwards analysis. In this paper we argue that both ϕ- and σ-functions are essential in bidirectional analysis.

[1] This can be generalised to something other than variables, such as expressions. We only consider relating data flow information to variables in order to keep the presentation simple.

We compare classical and sparse bidirectional data flow analysis for both statically typed programs (section 6) and dynamically typed programs (section 7). In both cases our sparse analysis is as precise and more efficient than previous classical analysis.

6 Simple Example

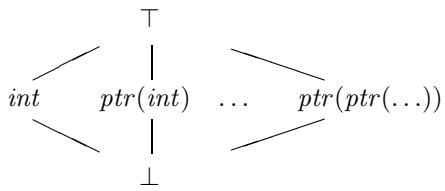
The simple program given below is taken from Aho et al [ASU86], example 10.50 in their book. This straightline program already satisfies the SSI property, so it does not need to be transformed at all. The array indexing operator is denoted by `[ ]` which requires an integer argument. The assignment operator is denoted by `:=`.

```
i := a[j]
k := a[i]
```

We adopt the following type algebra, where *int* is the integer type and *ptr* is the pointer type:

$$t ::= int \mid ptr(t)$$

It is impossible to infer the correct types by reasoning only forwards or only backwards. We do sparse data flow analysis on this example, in the style of sparse constant propagation [WZ91]. Our data flow lattice[2] is:



This lattice has the same shape as the constant propagation lattice, in that it is only three levels deep, but infinitely wide. The maximal element $\top$ represents "type not yet inferred" and the minimal element $\bot$ represents "overdefined type". The lattice meet operator $\sqcap$ is defined as follows:

$$\begin{aligned}
\top \sqcap x &= x \\
x \sqcap \top &= x \\
x \sqcap x &= x \\
x \sqcap \bot &= \bot \\
\bot \sqcap x &= \bot \\
x \sqcap y &= \bot \quad (x \neq \top, y \neq \top, x \neq y)
\end{aligned}$$

[2]We adopt the classical data flow convention for lattices, which is rather at odds with a semantic view of the world.

# complete iterations	$\langle E^i, E^j, E^k, E^a \rangle$
0	$\langle \top, \top, \top, \top \rangle$
1	$\langle int, int, \top, \top \rangle$
2	$\langle int, int, int, ptr(int) \rangle$
3	$\langle int, int, int, ptr(int) \rangle$

Figure 2: Trace of state of E after each sparse analysis iteration

Auxiliary functions are required to handle pointer referencing (ptr^+) and dereferencing (ptr^-):

$$\begin{aligned} \text{ptr}^+(\top) &= \top \\ \text{ptr}^+(\bot) &= \bot \\ \text{ptr}^+(t) &= ptr(t) \end{aligned} \qquad \begin{aligned} \text{ptr}^-(\top) &= \top \\ \text{ptr}^-(\bot) &= \bot \\ \text{ptr}^-(int) &= \bot \\ \text{ptr}^-(ptr(t)) &= t \end{aligned}$$

One type inference rule is needed for each kind of statement:

(*array deref*)	`x := y[z]`	$E^z := E^z \sqcap int$ $E^y := E^y \sqcap \text{ptr}^+(E^x)$ $E^x := E^x \sqcap \text{ptr}^-(E^y)$
(*var assign*)	`x := y`	$E^x := E^x \sqcap E^y$ $E^y := E^x \sqcap E^y$

A single global vector E of data flow information is maintained. (This is the characteristic feature of sparse data flow analysis.) E^v represents the lattice element associated with variable v. Initially, all entries in E are set to $\top$. A single iteration of the data flow analysis processes the program statement-by-statement and applies the appropriate inference rules. E is updated as specified by the inference rules. This iterative pass is repeated until a fixed point is reached, that is, E does not change any more.

We implemented this analysis in ML. E was four elements in size: $\langle E^i, E^j, E^k, E^a \rangle$. The fixed point was detected after three passes through the program. Figure 2 shows an execution trace of our analysis. This is an improvement over the approach given in [ASU86]. Their data flow analysis takes a forwards pass through the whole program, then a backwards pass, then another forwards pass, then another backwards pass which happens to detect the fixed point. They store two lattice values for each variable—one lattice value per variable per statement. So our sparse approach saves both computational space and time. (Extra space also requires extra time to manage and update.)

7 Complicated Example

In this section we apply our sparse data flow analysis to type inference for languages with dynamic type constraints. The example program shown in figure 3 is taken from figure 6 in [KDM03]. Our version of this program has been transformed into SSI. The original version can be recovered simply by eliding ϕ- and σ-functions and numerical variable subscripts. The only statements which are relevant to type inference for variable a are "use" statements and ϕ- and σ-functions. No other program statements are shown in figure 3, for the sake of simplicity.

Khedker et al qualify type information with a degree of certainty and with its origin. In this analysis we only qualify type information with its origin, in order to simplify the presentation. We achieve the same results as [KDM03] on this example. It should be straightforward to extend our analysis to handle degrees of certainty.

A control flow *ancestor* of node n is a node a that is passed through on at least one path from `Entry` to n. A control flow *descendant* of node n is a node d that is passed through on at least one path from n to `Exit`.

It is necessary to remember the origin of type information, since the type information propagated to a node from one point in the program may conflict with the type information propagated from another point. Type information generated at the current node is the most reliable. If this is not available then type information from ancestors should take precedence over type information from descendants because at run time, control flows from ancestors to descendants. Type information propagated from descendants should only be used where useful information is not available from ancestors. Type information propagated from a node other than an ancestor or a descendant should only be used as a last resort. The Γ operator below selects the most appropriate source of type information available.

The types used in this type system are integer i, real r and string s. We define $\mathcal{T}$ to be the set of all types $\{\mathsf{i}, \mathsf{r}, \mathsf{s}\}$. Our component data flow lattice $\hat{\mathcal{L}}$ is the power set of $\mathcal{T}$. The maximal element $\hat{\top}$ is the empty set. The minimal element $\hat{\bot}$ is the set $\mathcal{T}$. The partial order $\hat{\sqsubseteq}$ is the standard set inclusion operator $\supseteq$. The meet operator $\hat{\sqcap}$ on lattice elements is the standard set union operator $\cup$.

Now we define the compound lattice $\mathcal{L}$. Each compound lattice element X has four component elements $\langle X_c, X_a, X_d, X_o \rangle$, each of which is a member of $\hat{\mathcal{L}}$.

X_c represents type information generated at current nodes, that is, at "use" statements. X_a represents type information generated at ancestor nodes, that is, propagated forwards through ϕ- and/or σ-functions. X_d represents type information generated at descendant nodes, that is, propagated backwards through ϕ- and/or σ-functions. X_o represents type information generated at some other node, that is, indirectly in terms of control flow.

The maximal element $\top$ is $\langle \hat{\top}, \hat{\top}, \hat{\top}, \hat{\top} \rangle$. The minimal element $\bot$ is $\langle \hat{\bot}, \hat{\bot}, \hat{\bot}, \hat{\bot} \rangle$. The partial order $\sqsubseteq$ is defined as:

$$X \sqsubseteq Y = (X_c \hat{\sqsubseteq} Y_c) \wedge (X_a \hat{\sqsubseteq} Y_a) \wedge (X_d \hat{\sqsubseteq} Y_d) \wedge (X_o \hat{\sqsubseteq} Y_o)$$

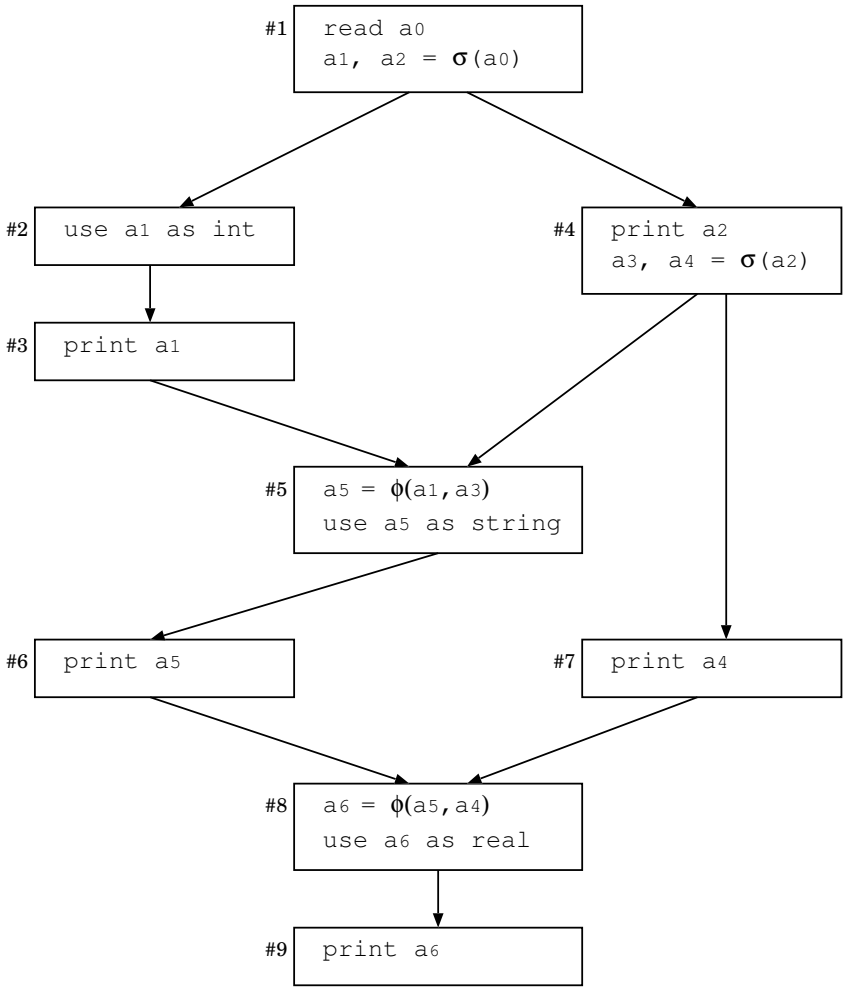


Figure 3: Example program from [KDM03]

The meet operator $\sqcap$ is defined as:

$$X \sqcap Y = \langle X_c \hat{\sqcap} Y_c, X_a \hat{\sqcap} Y_a, X_d \hat{\sqcap} Y_d, X_o \hat{\sqcap} Y_o \rangle$$

It is necessary to define specialised meet operators $\sqcap_a$ and $\sqcap_d$, which are used to ensure that type information flows to the correct components of the compound lattice elements. These correspond to the forwards ($\sqcap_a$) and backwards ($\sqcap_d$) edge flow functions of [KDM03].

$$X \sqcap_a Y = \langle X_c, X_a \hat{\sqcap} (Y_c \hat{\sqcap} Y_a), X_d, X_o \hat{\sqcap} (Y_d \hat{\sqcap} Y_o) \rangle$$

$$X \sqcap_d Y = \langle X_c, X_a, X_d \hat{\sqcap} (Y_c \hat{\sqcap} Y_d), X_o \hat{\sqcap} (Y_a \hat{\sqcap} Y_o) \rangle$$

The Γ operator below selects a precise estimate of the type from a compound lattice element X by giving precedence to information generated at the current node, then preferring ancestors over descendants.

$$\Gamma(X) = \begin{cases} X_c, & X_c \neq \hat{\top} \\ X_a, & X_c = \hat{\top} \wedge X_a \neq \hat{\top} \\ X_d, & X_c = \hat{\top} \wedge X_a = \hat{\top} \wedge X_d \neq \hat{\top} \\ X_o & \text{in all other cases} \end{cases}$$

E represents the single global vector of data flow information that characterises sparse data flow analysis. E is indexed by SSI variable names. E^v represents the compound lattice element associated with variable v.

The actual inference rules for this type system are given below:

(*use*)	use x as t	$E^x_c := E^x_c \hat{\sqcap} \{\mathsf{t}\}$
(*sigma*)	$x_1, x_2 \leftarrow \sigma(x_0)$	$E^{x_0} := E^{x_0} \sqcap_d (E^{x_1} \sqcap E^{x_2})$ $E^{x_1} := E^{x_1} \sqcap_a E^{x_0}$ $E^{x_2} := E^{x_2} \sqcap_a E^{x_0}$
(*phi*)	$x_0 \leftarrow \phi(x_1, x_2)$	$E^{x_0} := E^{x_0} \sqcap_a (E^{x_1} \sqcap E^{x_2})$ $E^{x_1} := E^{x_1} \sqcap_d E^{x_0}$ $E^{x_2} := E^{x_2} \sqcap_d E^{x_0}$

Initially each element of E is set to $\top$. A single iteration of the data flow analysis processes the program statement-by-statement and applies the appropriate inference rules. The vector E is updated as specified by the inference rules. This iterative pass is repeated until a fixed point is reached, that is to say, E does not change at all after an entire iterative pass. (The order in which the statements are processed in the iterative pass is irrelevant to the final state of E, although it may have some effect on the number of iterations required to reach a fixed point.)

We have implemented this type inference algorithm in ML using the sparse data flow analysis framework given above. We achieve the same precision of results as [KDM03], using a single lattice element for each variable, and in six passes through the program. In summary, our results are:

variable v	a_0	a_1	a_2	a_3	a_4	a_5	a_6
$\Gamma(E^v)$	$\{\mathsf{i},\mathsf{s},\mathsf{r}\}$	$\{\mathsf{i}\}$	$\{\mathsf{s},\mathsf{r}\}$	$\{\mathsf{s},\mathsf{r}\}$	$\{\mathsf{r}\}$	$\{\mathsf{s}\}$	$\{\mathsf{r}\}$

8 Related Work

The previous work on type inference as bidirectional data flow analysis has been mentioned already in section 2. This previous work [Ten74, KU80, ASU86, KDM03] uses classical, non-sparse data flow analysis. In this paper we have applied sparse data flow analysis to the same problems. Our results are as precise and our analysis is more efficient.

A restricted form of type inference is used for object-oriented languages in order to compute sets of possible concrete classes for variables. This type information enables the replacement of virtual method calls by direct calls, and the elimination of runtime type checks. Chambers et al [CDG96] describe intraprocedural class analysis, which is a standard forwards data flow analysis. Diwan et al [DMM01] describe intraprocedural type propagation, which is similar to reaching definitions analysis. Bidirectional data flow analysis is not necessary to solve the problem of concrete type inference in object-oriented languages.

Although bidirectional type inference has not previously been done using sparse data flow analysis, there has been some work on solving other bidirectional problems using sparse techniques. Dhamdhere et al [DRZ92] show how to perform partial redundancy elimination using sparse data flow analysis. However they reduce the bidirectional analysis to a simpler unidirectional analysis. Kennedy et al [KCL+99] present the SSAPRE algorithm for partial redundancy elimination on SSA. They do a series of forwards and backwards data flow analysis passes on the factored redundancy graph, which is derived from SSA.

There has been little previous work on type inference for SSA-like intermediate representations. Mycroft [Myc99] shows how to perform type inference on register transfer language (RTL) code in SSA. However he uses constraint-based type inference. It would be interesting to translate Mycroft's system into our sparse data flow analysis framework.

9 Concluding Remarks

A key observation made by Khedker et al [KDM03] is that

> "Data flow analysis refers to forms of program analysis with no auxiliary store; each node in the program has an attribute. The space required by these attributes is usually tightly bounded by the program whereas the auxiliary store in constraint-based analyses is not tightly bounded."

The sparse data flow analysis presented in this paper also requires no auxiliary store. Each SSI variable in the program (rather than each node) has an attribute. The space required by these attributes is generally smaller for sparse

data flow analysis, since it is only necessary to store a single lattice element for each variable, rather than one lattice element per variable at each node.

Data flow information is never killed in sparse data flow analysis, unlike classical data flow analysis. Sparse data flow information has to hold for the entire program, so information must only be generated if it holds globally, rather than just locally. Classically, data flow information is killed when it is true for some part of the program, but not another part. These local, control flow sensitive effects cannot be modelled by sparse data flow analysis. However they do not need to be, since SSI variable renaming effectively factors control flow sensitivity into the variable naming scheme [HH98].

Nielson et al [NNH99] state that there is a very strong connection between the data flow equational approach and the constraint-based approach to program analysis. It would be interesting to make a proper comparison of our sparse data flow analysis technique for type inference with traditional constraint-based analysis.

Plans for future work include the development of a real-world implementation of this type inference mechanism. At the moment, we have merely produced simple ML proof-of-concept implementations. As mentioned in section 8, we hope to construct a Mycroft-style type inference system for RTL code [Myc99].

To conclude, in this paper we have shown that SSI can be used to perform sparse bidirectional data flow analysis. SSA only supports sparse forwards data flow analysis. The extra analysis potential of SSI is required to handle bidirectional information flow.

References

[Ana99] C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Sep 1999.

[AR03] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems*, pages 59–68, 2003.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[Car97] Luca Cardelli. *Type Systems*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report 96-06-

02, Department of Computer Science and Engineering, University of Washington, Jun 1996.

[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, Oct 1991.

[DMM01] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, Jan 2001.

[DRZ92] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 212–223, 1992.

[GSR03] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 273–284, 2003.

[HH98] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 97–105, 1998.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.

[KD99] Uday P. Khedker and Dhananjay M. Dhamdhere. Bidirectional data flow analysis: myths and reality. *SIGPLAN Notices*, 34(6):47–57, 1999.

[KDM03] Uday P. Khedker, Dhananjay M. Dhamdhere, and Alan Mycroft. Bidirectional data flow analysis for type inferencing. *Computer Languages, Systems and Structures*, 29(1–2):15–44, 2003.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, Jul 1994.

[KU80] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, Jan 1980.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, Dec 1978.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb 1979.

[MRB95] T. J. Marlowe, B. G. Ryder, and M. Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Laboratory of Computer Science, Rutgers University, Jul 1995.

[Myc99] Alan Mycroft. Type-based decompilation. In *Proceedings of the European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer-Verlag, 1999.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Ram97] G. Ramalingam. On sparse evaluation representations. In *Proceedings of the 4th International Symposium on Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[RR00] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[Ruf95] Erik Ruf. Optimizing sparse representations for dataflow analysis. *ACM SIGPLAN Notices*, 30(3):50–61, Mar 1995.

[Sin03] Jeremy Singer. SSI extends SSA. In *Work in Progress Session Proceedings of the Twelth International Conference on Parallel Architectures and Compilation Techniques*, Sep 2003.

[Ten74] A. Tennenbaum. *Type determination for very high level languages*. PhD thesis, Courant Institute, New York University, Oct 1974. As cited by [ASU86].

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr 1991.