Simple closure analysis*

Fritz Henglein
DIKU, University of Copenhagen
Universitetsparken 1
2100 Copenhagen Ø, Denmark
Internet: henglein@diku.dk

March 12, 1992

1 Introduction

Closure analysis determines statically which function definitions reach which program points. This information is used for many different purposes; e.g., type inference for object-oriented programming languages [PS91], globalization analysis of functional programs [Ses89], partial evaluation [Bon90], type recovery in Scheme [Shi90], and others. The reason why closure analysis is such a fundamental analysis in different applications is that it is (sort of) the *universal* data flow analysis problem for monomorphic (data flow oriented) analyses for higher-order (functional) languages. As such it may be viewed as the analogue of path analysis, which is universal for (continuous) first-order (classical) data flow analysis problems [Tar81].

General closure analysis is, unlike its first-order counterpart, expensive in the worst case: the best known algorithms take time $\Theta(n^3)$ in the worst case [Hen91a,PS92]. Many times it is sufficient, however, to get somewhat coarser information than the exact closure information. Coarser here means that the results may indicate that more function definitions reach a point than a precise closure analysis would really yield; nonetheless the information should in practice be only marginally different from the exact analysis.

In this note we exhibit a simple, but very efficient closure analysis based on the binding-time analysis of [Gom90] and the algorithm [Hen91b] for it. (In fact the algorithm here is substantially simpler than the one in [Hen91b].) The computed abstract value flow information[1] is coarser than normal closure analysis exactly in the following sense: exact closure analysis keeps track of uni-directional flow of (abstract) values from one program point to another whereas our *simple* closure analysis works with the assumption that flows are reversible; that is, that any abstract value that flows from point p to point p' can also flow (backwards!) from p' to p.

In the practice of partial evaluation this loss of information appears to be insubstantial. Since the simple closure analysis algorithm runs in almost-linear time [Hen91b] and is very

*DIKU Semantics Report D-193

[1]We prefer to call the abstractions of values corresponding to expressions in a program abstract values rather than (abstract) closures, since these values represent values other than (run-time) closures, including pairs, integers, etc.

efficient in practice [Hen91c] this appears to be an attractive alternative to computing complete closure information.

2 Basic idea of simple closure analysis

In the binding-time analysis of [Gom89], on which [Hen91b] is based, we begin by associating a unique *abstract value* (also called a *token*, *label* or a *type variable* depending on the intention of their use) with every (sub)expression in a program, and constraints are extracted that capture the flow of actual values represented by these abstract values in the program. The flow is, of course, a conservative approximation of the actual flow of data. It not only makes the standard assumptions that all expressions inside a function definition are actually evaluated and that abstract values can flow forwards *and* backwards, as mentioned above, but also that the flows of different arguments of a function merge inside the function — the latter is why we refer to this analysis as *monomorphic.*

In the second step, the critical one, these constraints are *normalized* by a (very small) set of rewriting rules. In the process different abstract values may be identified, which is tantamount to saying that one abstract value could “flow” to the other, in any direction. At the end of this process a single abstract value represents a whole set of abstract values that have been identified during normalization. Such a set may contain (abstract values labeling) closures as well as other program points, notably application points such as the actual parameters of function applications. This information can be interpreted by saying that the closures in the set may reach any of the application points in the same set. The analysis is conservative w.r.t. to exact closure analysis in the sense that no other closures reach the application points, but that some of the reported closures may actually be shown *not* to reach some application point in the same set by exact closure analysis. (Of course, “exact” closure analysis is itself a conservative approximation of the actual dynamic flow of run-time values, including concrete closures.)

3 Simple closure analysis exemplified

In this section we exemplify the steps above by considering a simple example.

Consider the following code (fragment), representing Turner’s tautology checker and two calls to it:

```
taut = fn f =¿ fn n. if n = 0 then f else taut (f true) (n-1) and taut (f false) (n-1)
g = fn x =¿ fn y =¿ (x and not y) or (not x or y)
h = fn z =¿ z
taut g 2 taut h 1
```

3.1 Constraint extraction

In the first step we associate a distinct abstract value with *every* occurrence of a subexpression occurring in this code; for simplicity’s sake all occurrences of a variable have the same abstract value. We shall refer to the abstract value of an (occurrence of an) expression by the special variable α indexed by the expression; e.g., the abstract values of the three function definitions are α_{taut}, α_g and α_h. On top of these abstract values representing (abstract) closures we have the abstract values $\alpha_{\lambda f}$ and $\alpha_{\lambda n}$ corresponding to the partial applications of *taut* to one, resp. two arguments. Finally, we have $\alpha_{\lambda y}$ corresponding to the partial application of g to one argument. These are all the abstract values representing *(abstract) closures*; of course, as mentioned before, *every* subexpression has a corresponding (unique) abstract value.

In the constraint extraction phase of the binding-time analysis algorithm the constraints in Figure 1 are generated for the code above. Two kinds of constraints are generated.

- $\alpha = \alpha'$: Such an equation between abstract values expresses that one flows to the other, and vice versa (Remember our basic assumption of bidirectionality of flow!); e.g., α could be the abstract value of an actual argument to a function, and α' the abstract value of the function's corresponding formal parameter.

- $\alpha \to \alpha' \leq \alpha''$ or $integer \leq \alpha''$, *etc.*: An inequation has an *abstract value constructor* applied to $n \geq 0$ abstract values on the left-hand side and an abstract value on the right; such an inequality expresses that the right-hand side abstract value "can be" the abstract value on the left (there may be more than one, but not with the same constructor; see below).

3.2 Constraint normalization

In the constraint normalization phase we combine definitional information (constraints of the second form above) and form equivalence classes of abstract values (for the constraints of the first form).

For simple closure analysis we need only 2 of the 11 rewriting rules in [Hen91b]. In particular, there is no need for an occurs check rule since we don't have to interpret the result as finite type expressions; and there is no need for a special type Dynamic (or Λ) representing either possible type errors or run-time computable expressions or both, since we are interested neither in the former nor in the latter. In particular, the rewriting rules manipulating Dynamic can be omitted. We end up with the rewriting rules in Figure 2.

As proved in [Hen91b] a rewriting system normalizing constraints with additional rewriting rules can be implemented in time $O(n\alpha(n, n)$ where $\alpha(n, n) < 5$ for any value of n smaller than the number of atoms in the universe. It is quite easy to see that this more rudimentary rewriting system can be implemented in the same time. In fact, a slightly modified unification closure algorithm will do the job.

3.3 Interpretation as closure sets

The normalization of the constraints results in a transition $C \Rightarrow C'$ labeled with a substitution S. For closure analysis it is this substitution we are interested, not the normal form constraints left over. Since the substitution maps abstract values to abstract values the pre-images of every abstract value in its range form a partition of the original abstract values in the program. For our example these sets are displayed in Figure 3.

Looking at the set E_f we can see that simple closure analysis reports that the functions f, g, all their partial applications, and the abstract values corresponding to their bodies may reach the formal parameter f of *taut*. In this case this happens to be precisely correct, but in general there will be some overreporting of one abstract value reaching (the point of) another.

$$
\begin{array}{lcl}
\alpha_f \rightarrow \alpha_{\lambda n} & \leq & \alpha_{taut} \\
\alpha_n \rightarrow \alpha_{if} & \leq & \alpha_{\lambda n} \\
\alpha_f & = & \alpha_{if} \\
\alpha_{and} & = & \alpha_{if} \\
integer & \leq & \alpha_n \\
integer & \leq & \alpha_0 \\
bool & \leq & \alpha_{n=0} \\
integer & \leq & \alpha_1 \\
integer & \leq & \alpha_n \\
integer & \leq & \alpha_{n-1} \\
bool & \leq & \alpha_{true} \\
\alpha_{true} \rightarrow \alpha_{ftrue} & \leq & \alpha_f \\
\alpha_{ftrue} \rightarrow \alpha_{taut(ftrue)} & \leq & \alpha_{taut} \\
\alpha_{n-1} \rightarrow \alpha_{taut(ftrue)(n-1)} & \leq & \alpha_{taut(ftrue)} \\
bool & \leq & \alpha_{false} \\
\alpha_{false} \rightarrow \alpha_{ffalse} & \leq & \alpha_f \\
\alpha_{ffalse} \rightarrow \alpha_{taut(ffalse)} & \leq & \alpha_{taut} \\
\alpha_{n-1} \rightarrow \alpha_{taut(ffalse)(n-1)} & \leq & \alpha_{taut(ffalse)} \\
bool & \leq & \alpha_{t(ftrue)(n-1)} \\
bool & \leq & \alpha_{t(ffalse)(n-1)} \\
bool & \leq & \alpha_{and} \\
\alpha_x \rightarrow \alpha_{\lambda y} & \leq & \alpha_g \\
\alpha_y \rightarrow \alpha_{or} & \leq & \alpha_{\lambda y} \\
bool & \leq & \alpha_{or} \\
 & \cdots & \\
\alpha_z \rightarrow \alpha_z & \leq & \alpha_h \\
integer & \leq & \alpha_2 \\
\alpha_g \rightarrow \alpha_{tautg} & \leq & \alpha_{taut} \\
\alpha_2 \rightarrow \alpha_{tautg2} & \leq & \alpha_{tautg} \\
integer & \leq & \alpha'_1 \\
\alpha_h \rightarrow \alpha_{tauth} & \leq & \alpha_{taut} \\
\alpha'_1 \rightarrow \alpha_{tauth1} & \leq & \alpha_{tauth}
\end{array}
$$

Figure 1: Abstract value constraints for Turner's tautology checker

$$
\begin{array}{l}
C \cup \{\alpha \to \alpha' \leq \gamma, \beta \to \beta' \leq \gamma\} \\
\Rightarrow \\
C \cup \{\alpha \to \alpha' \leq \gamma, \alpha = \beta, \alpha' = \beta'\} \\
\\
C \cup \{\alpha = \alpha'\} \\
\overset{S}{\Rightarrow} \\
S(C) \\
\text{where } S = \{\alpha \mapsto \alpha'\}
\end{array}
$$

Figure 2: Constraint rewritings for simple closure analysis

$$
\begin{array}{lcl}
E_{taut} & = & \{\alpha_{taut}\} \\
E_f & = & \{\alpha_f, \alpha_g, \alpha_h, \alpha_{if}, \alpha_{and}, \alpha_{taut(ftrue)(n-1)}, \alpha_{taut(ffalse)(n-1)}, \\
 & & \alpha_{ftrue}, \alpha_{ffalse}, \alpha_{lambday}, \alpha_{or}, \alpha_z\} \\
E_{\lambda n} & = & \{\alpha_{\lambda n}, \alpha_{taut(f\,true)}, \alpha_{taut(ffalse)}\} \\
E_n & = & \{\alpha_n, \alpha_{n-1}\}
\end{array}
$$

Figure 3: Equivalence classes of abstract values for Turner's tautology checker

4 Damas-Milner polymorphism

Simple closure analysis is easily and naturally extended to a polymorphic analysis since the normalized constraints of an expression can be interpreted as an "abstract value scheme": e.g, the normalized constraints of $\lambda x.xx$ are $\{\alpha \to \beta \leq \alpha, \alpha \to \beta \leq \gamma\}$ where γ is the abstract value of the whole expression. This then corresponds to the polymorphic "type scheme" $\forall \alpha\beta\gamma : \alpha \to \beta \leq \alpha, \alpha \to \beta \leq \gamma.\gamma$. Such an abstract value scheme can be instantiated by substituting new abstract values for the scheme variables at every occurrence of the identifier let-bound to $\lambda x.xx$.

5 Milner-Mycroft polymorphism

ML-style polymorphic (also called polyvariant) simple closure analysis as above does not find abstract value schemes for mutually recursive definitions. One approach is to topsort the strong components of a mutually recursive definition and treat them as a nonrecursive serious of groups of genuinely mutually recursive definitions. The groups themselves are analyzed by a monomorphic closure analysis. This solution has well-known disadvantages, especially in a setting where closure information has to be relayed back to a user/programmer or where attributes of code generated from such analyses has to satisfy certain robustness criteria, such as: if the definition of function f is changed, and this has nothing to do with the definitions of functions g and h, then after these changes the code for g and h should not be negatively affected.

A properly more general (and computationally principally more expensive) solution is to pattern the analysis after Mycroft's type system with polymorphic recursion.

For this it is necessary to add an additional form of constraints, $\alpha \mapsto^i \alpha'$ and additional rewriting rules. This results essentially in the problem of regular semi-unification, which is undecidable [DR90], but behaves no worse than ML type inference in practice [Hen92]. Since it utilizes a form of "lazy" instantiation of (type) schemes it may even lead to more efficient implementations than the ML type inference algorithms, most of which use "eager" type instantiation.

References

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Dec. 1990.

[DR90] J. Dörre and W. Rounds. On subsumption and semiunification in feature algebras. In *Proc. 1990 IEEE Symp. on Logic in Computer Science (LICS)*, pages 300–311. IEEE Computer Society Press, July 1990.

[Gom89] C. Gomard. Higher order partial evaluation – hope for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, September 1989.

[Gom90] C. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proc. LISP and Functional Programming (LFP), Nice, France*, July 1990.

[Hen91a] F. Henglein. An $o(n^2)$ single point online closure analysis algorithm. Research notes, May 1991.

[Hen91b] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 448–472. Springer, Aug. 1991. Lecture Notes in Computer Science, Vol. 523.

[Hen91c] F. Henglein. Global tagging optimization by type inference. Semantique Note 102. Submitted to LISP and Functional Programming '92, Nov. 1991.

[Hen92] F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems (TOPLAS)*, 1992. To appear.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Phoenix, Arizona*, pages 146–161. ACM Press, Oct. 1991.

[PS92] J. Palsberg and M. Schwartzbach. Polyvariant flow analysis of λ-calculus. Submitted for publication, Feb. 1992.

[Ses89] P. Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 39–53. ACM Press, Sept. 1989.

[Shi90] O. Shivers. Data-flow analysis and type recovery in Scheme. Technical Report CMU-CS-90-115, Carnegie Mellon University, March 1990.

[Tar81] R. Tarjan. A unified approach to path problems. *J. of the ACM*, 28(3):577–593, July 1981.