INDY User's Manual (v1.8)

Elvira Albert *María Alpuente* *Germán Vidal*

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
{ealbert,alpuente,gvidal}@dsic.upv.es

Moreno Falaschi

Dipartimento di Matematica e Informatica Università di Udine
Via delle Scienze 206, 33100 Udine, Italy
falaschi@dimi.uniud.it

November 28, 1998

1 Introduction

Partial evaluation is a semantics-preserving optimization technique for computer programs which consists of the specialization of the program w.r.t. parts of its input [CD93, JGS93]. This report introduces the INDY system, which is the first automatic partial evaluator for functional logic programs. INDY (Integrated Narrowing-Driven specialization system) is a rather concise implementation of the generic narrowing-driven partial evaluator originally introduced in [AFV96]. It implements the different instances of the algorithm which have been presented in [AFV98b] (based on innermost narrowing), [AFJV97] (based on lazy narrowing), and [AHLV98] (based on needed narrowing), as well as the control improvements of [AAF+98]. In contrast to the approach usually taken with pure functional languages, INDY uses the unification-based computation mechanism of *narrowing* for specialization. This coincides with the usual execution mechanism of functional logic languages, which allows us to deal with expressions containing partial information by means of logical variables and unification in a natural way, as well as to consider efficient evaluation strategies and include a deterministic simplification phase (see, e.g., [Han94b]). This provides better opportunities for optimization and improves both the overall specialization and the efficiency of the method.

INDY is parametric w.r.t. the narrowing strategy which is used for the automatic construction of (finite) narrowing trees. The system follows a structure which is similar to the framework developed by Martens and Gallagher [MG95] for partial deduction of logic programs, where a clear distinction between *local* and *global* control is done. Roughly speaking, local control concerns the construction of partial narrowing trees for single terms, while global control is dedicated to ensuring the *closedness* of the partially evaluated program, producing a sufficient number of specialized versions for each function definition in the original program without risking nontermination. The algorithm starts by partially evaluating the initial set of terms, and then it recursively specializes the terms which are introduced dynamically during this process. Appropriate *unfolding* and *abstraction* (generalization) operators are needed to ensure that infinite unfolding is not attempted (local termination) and to ensure that the set of (to be) partially evaluated terms is kept finite throughout the specialization process (global termination). The inclusion of primitive function symbols (and, in particular, the conjunctive operator) in input calls poses specific control problems that require an special treatment in order to achieve a good specialization (see [AAF+98]). At the local level of control, the order in which elements are chosen during the construction of the local narrowing trees is crucial to achieving good specialization. Some kind of dynamic selection strategy which keeps track of the ancestors reduced in the same derivation is necessary to significantly speed-up execution. At the global level of control, the NPE algorithm requires particular techniques for carefully splitting complex terms, while still being able to keep the terms which may communicate data structures as close together as possible. This is not always possible because ensuring termination sometimes forces splitting at undesirable positions. A sophisticated unfolding rule and a novel abstraction operator are implemented to care for the appropriate handling of primitive function symbols. Using the terminology of [GS96], our control strategy allows us to produce both *polyvariant* and *polygenetic* specializations, i.e., it can produce different specializations for the same function definition, and can also combine distinct original function definitions into a comprehensive specialized function.

The INDY system is written in SICStus Prolog and is publicly available. The complete implementation consists of about 800 clauses (4000 lines of code). The partial evaluator is expressed by 95 clauses, the metainterpreter by 415 clauses (including the code needed to handle the *ground representation*), the parser and other utilities by 90 clauses, the user interface by 100 clauses, and the postprocessing renaming by 95 clauses. The implementation only considers unconditional programs. Conditional programs are treated by using the predefined functions `and`, `if_then_else`, and `case_of`, which are reduced by standard defining rules (see, e.g., [MR92]). The current implementation allows the user to select either a *call-by-name* (lazy or needed narrowing) or a *call-by-value* (innermost narrowing) evaluation strategy. Normalization of goals between narrowing steps is allowed (using only a terminating subset of program rules).

The specializer also performs a post-processing renaming transformation which is useful for automatically fulfilling the *independence* condition as well as for ensuring that the left-hand sides of the specialized rules are patterns [AFJV97]. This ensures that the different specialization for the same function definition are correctly distinguished, which is crucial for polyvariant specialization.

2 Starting INDY

The INDY system needs no previous installation. It runs under SICStus Prolog v3.x and is is entered by simply typing:

```
?- consult('indyv1.8').
```

Once the system is loaded, you can type `help` to display the complete range of commands of the system:

```
>> help.

------------------------------------------------------------
load   -> Read in a file containing the source program
read   -> Read a source program from the standard input
list   -> Display the source program
nwing  -> Set the narrowing strategy
normal -> Switch normalization on/off
unfold -> Set the unfolding rule
split  -> Set the splitting option
mix    -> Start the specialization process
flags  -> Consult the current settings
help   -> See this info
cd     -> Change the working directory
sh     -> Start a new interactive default shell process
quit   -> Leave the system
------------------------------------------------------------
```

Commands must be followed by a period. These commands are described below.

2.1 Loading Programs

Language syntax is very simple. Programs consist of a sequence of rules `Term -> Term` for narrowing and a (possibly empty) sequence of rules of the form `Term --> Term` used for rewriting. This distinction is useful when non-terminating programs are considered, since the (possibly empty) terminating subset of the program rules can be safely used for simplification [Fri85, Han94a].

The following grammar formalizes our language. We use the following notations in grammar rules. Terminals are character sequences enclosed within apostrophes, `{...}` denotes a (possibly empty) repetition, `[...]` denotes an optional part, and ...|... denotes a choice.

```
Program     := Rule ';' { Rule ';' } { RwRule ';' }
Rule        := Pattern '->' Term
RwRule      := Pattern '-->' Term
TermList    := Term { ',' Term }
ConstrList  := ConstrTerm { ',' ConstrTerm }
Term        := Variable | FuncTerm
FuncTerm    := FuncName [ '(' TermList ')' ]
Pattern     := Defined [ '(' ConstrList ')' ]
ConstrTerm  := Constructor [ '(' ConstrList ')' ]
FuncName    := Defined | Constructor
Defined     := Identifier
Constructor := ':' Identifier
Identifier  := Lowercase { Letter | Digit }
Variable    := Uppercase { Letter | Digit }
Letter      := Uppercase | Lowercase
Uppercase   := 'A' | 'B' | ... | 'Z'
Lowercase   := 'a' | 'b' | ... | 'z'
Digit       := '0' | '1' | ... | '9'
```

We note that a Prolog-like notation for lists is also allowed in programs. In order to load a source program, you can use one of these two commands:

- `load`: it is used to read a file which contains the source program and to put the program into the internal database. Filenames comply with the Prolog notation for terms, i.e., simple quotes are required when special characters appear in the filename (e.g., you should type `'append.fl'` rather than `append.fl`). The (terminating) subset of program rules to be used for normalization must be duplicated, with the symbol `->` replaced by `-->`.

- `read`: this option reads a source program from the standard input on-line. Let us illustrate the use of `read` on the well-known program `append`:

```
>> read.

Introduce program rules (quit to finish):

"lhs -> rhs;" : "append([],Ys) -> Ys;".
"lhs -> rhs;" : "append([X|Xs],Ys) -> [X|append(Xs,Ys)];".
"lhs -> rhs;" : quit.

Same rules for normalization? (yes/no): no.
```

```
Introduce rules for normalization (quit to finish):

"lhs --> rhs;" : "append([],Ys) --> Ys;".
"lhs --> rhs;" : "append([X|Xs],Ys) --> [X|append(Xs,Ys)];".
"lhs --> rhs;" : quit.

Program loaded!
```

Of course, the user can avoid re-typing the rules in this example by answering `yes` to the question asked on-line.

The program which is currently loaded into the system can be displayed by using the INDY command `list`.

2.2 Narrowing Strategy

The narrowing strategy is used by the partial evaluator to construct local narrowing trees. It can be fixed by typing the command `nwing`. Currently, the user can select one of these options:

- `innermost`: it implements the innermost narrowing strategy of [Fri85]; namely, only the (leftmost) innermost position of the goal is considered for the next narrowing step. This strategy requires programs to be constructor-based, completely-defined.
- `lazy`: it implements the lazy narrowing strategy of [MR92]. At each step, only the outermost position is narrowed, unless further evaluation of inner subterms are *demanded* by some program rule. The completeness of this strategy is ensured, e.g., for constructor-based (weakly) orthogonal programs. Note that, since programs are not required to terminate, a terminating subset of the program rules has to be selected for normalization in order to guarantee the finiteness of the partial evaluation process.
- `needed`: it implements the needed narrowing strategy of [AEH94]. This strategy only performs narrowing steps which are necessary for computing solutions. The completeness and optimality of this strategy is ensured for *inductively sequential programs* [Ant92].

Innermost as well as lazy narrowing strategies can be speeded up by the use of normalization between narrowing steps. This setting can be fixed by using the command `normal`. The default setting for the narrowing strategy is: needed narrowing.

2.3 Unfolding Rule

The unfolding rule determines how to build local narrowing trees and, what is of even greater significance, how to stop constructing them. The unfolding rule can be set up by typing the command `unfold`. Currently, there are three options:

- `emb_goal`: the nonembedding unfolding rule, which uses the *homeomorphic embedding ordering* (see, e.g., [AFV98b, SG95]) to stop the construction of the narrowing trees;
- `emb_redex`: this unfolding rule uses a dynamic narrowing selection strategy to determine the set of positions to be unfolded by a don't-care selection within a conjunction. It implements the *dependency_clash* test of [AAF+98] using homeomorphic embedding on comparable ancestors of selected redexes to ensure finiteness (note that it differs from `emb_goal` in that `emb_redex` implements dynamic scheduling on conjunctions and that homeomorphic embedding is checked on simple redexes rather than on whole goals);
- `comp_redex`: it differs from `emb_redex` in that `comp_redex` uses a simpler definition of *dependency_clash* based on comparable ancestors of selected redexes as a whistle;
- `depthk`: a depth-k bound strategy, which expands derivations down to the depth-k frontier of the tree.

The default unfolding rule is `emb_goal`. For instance, in order to select an `emb_redex` strategy, you should proceed as follows:

```
>> unfold.

Unfolding rule? (emb_goal/emb_redex/comp_redex/depthk): emb_redex.
```

2.4 Splitting strategy

The abstraction operator carefully considers primitive functions to provide for accurate polygenetic specialization. By setting the splitting strategy on, the partition of expressions containing primitive function symbols is allowed (in order to avoid unnecessary generalization).

The splitting strategy is not fixed by default. This option may be fixed by using the command `split`. For example, in order to allow the splitting of terms containing the primitive functions `and` (conjunction) and `eq` (equality), you can proceed as follows:

```
>> split.

Splitting of primitive terms? (yes/no): yes.

List of primitive symbols? ["and","eq"].
```

2.5 Specialization

The partial evaluation process is started by the command `mix`. The input data is a list containing the initial calls (terms) to specialize. For example, if we have loaded the program `append`, it can be specialized (using needed narrowing and the `emb_redex` unfolding rule) w.r.t. the term `append(append(X,Y),Z)` as follows:

```
>> mix.

List of expressions? ["append(append(X,Y),Z)"].

Final Set of Calls:
  append(RLN,YLN)
  append(append(X,Y),Z)

Independent Renaming:
  append(RLN,YLN) => append2_1(RLN,YLN)
  append(append(X,Y),Z) => append3_1(X,Y,Z)

Specialized program:

append2_1([],A) -> A;
append2_1([A|B],C) -> [A|append2_1(B,C)];
append3_1([],[],A) -> A;
append3_1([],[A|B],C) -> [A|append2_1(B,C)];
append3_1([A|B],C,D) -> [A|append3_1(B,C,D)];

Mix Time: 70 ms.

Save to a file? (yes/no) no.
```

The partial evaluator returns the final set of specialized calls[1], the renamed functions for these calls, and the partially evaluated program (which can be saved into a file).

This specialized program is able to concatenate three lists by traversing the first input list only once, whereas the source program passes twice over this list.

2.6 Utilities

The INDY system also has some utility commands:

- `cd`: it is used to change the working directory;
- `flags`: it displays the current settings (narrowing strategy and unfolding rule) of the system;
- `sh`: it starts an external shell.

The command `quit` is used to leave the system.

3 An Example of Specialization

The following example is a kind of standard test in partial evaluation and similar techniques: the specialization of a semi-naïve pattern matcher for a fixed pattern into an efficient algorithm (sometimes called "the KMP-test" [SGJ96]). This example is particularly interesting because it provides a kind of optimization that neither (conventional) partial evaluation nor deforestation can achieve. We consider the specialization of the program `kmp` to the pattern `[a,a,b]` using needed narrowing and the `comp_redex` unfolding rule:

```
>> load.

Filename? 'Examples/kmp'.

File loaded!

>> list.
```

[1]Note that the final set of calls can be different from the original one, since the abstraction operator can add or delete terms from the current set at each iteration of the algorithm.

```
--Program Rules--

  match(P,S) -> loop(P,S,P,S);
  loop([],SS,OP,OS) -> :true;
  loop([P|PP],[],OP,OS) -> :false;
  loop([P|PP],[S|SS],OP,OS) -> if(eq(P,S),loop(PP,SS,OP,OS),next(OP,OS));
  next(OP,[]) -> :false;
  next(OP,[S|SS]) -> loop(OP,SS,OP,SS);
  if(:true,A,B) -> A;
  if(:false,A,B) -> B;
  eq(:a,:a) -> :true;
  eq(:b,:b) -> :true;
  eq(:a,:b) -> :false;
  eq(:b,:a) -> :false;

--Normalization Rules--

  match(P,S) --> loop(P,S,P,S);
  loop([],SS,OP,OS) --> :true;
  loop([P|PP],[],OP,OS) --> :false;
  loop([P|PP],[S|SS],OP,OS) --> if(eq(P,S),loop(PP,SS,OP,OS),next(OP,OS));
  next(OP,[]) --> :false;
  next(OP,[S|SS]) --> loop(OP,SS,OP,SS);
  if(:true,A,B) --> A;
  if(:false,A,B) --> B;
  eq(:a,:a) --> :true;
  eq(:b,:b) --> :true;
  eq(:a,:b) --> :false;
  eq(:b,:a) --> :false;


>> nwing.

Narrowing strategy? (lazy/needed/innermost): lazy.


>> normal.

Normalization? (yes/no): yes.


>> unfold.

Unfolding rule? (emb_goal/emb_redex/comp_redex/depthk): comp_redex.
```

```
>> mix.

List of expressions? ["match([:a,:a,:b],S)"].


Final Set of Calls:
  loop([:b],SSZT,[:a,:a,:b],[:a,:a|SSZT])
  loop([:a,:a,:b],SSF,[:a,:a,:b],SSF)
  loop([:a,:b],SSF,[:a,:a,:b],[:a|SSF])
  match([:a,:a,:b],S)


Independent Renaming:
  loop([:b],SSZT,[:a,:a,:b],[:a,:a|SSZT]) => loop1_1(SSZT)
  loop([:a,:a,:b],SSF,[:a,:a,:b],SSF) => loop1_2(SSF)
  loop([:a,:b],SSF,[:a,:a,:b],[:a|SSF]) => loop1_3(SSF)
  match([:a,:a,:b],S) => match1_1(S)


Specialized program:

loop1_1([]) -> :false;
loop1_1([:b|A]) -> :true;
loop1_1([:a|A]) -> loop1_1(A);
loop1_2([]) -> :false;
loop1_2([:a|A]) -> loop1_3(A);
loop1_2([:b|A]) -> loop1_2(A);
loop1_3([]) -> :false;
loop1_3([:a|A]) -> loop1_1(A);
loop1_3([:b|A]) -> loop1_2(A);
match1_1([]) -> :false;
match1_1([:a|A]) -> loop1_3(A);
match1_1([:b|A]) -> loop1_2(A);

Mix Time: 1100 ms.
```

This program never backs up on the subject string when a mismatch is detected: whenever three consecutive `a`'s are found, the algorithm looks for one `b`, instead of attempting to match the whole pattern `[a,a,b]` from scratch. This is essentially a KMP-style pattern matcher [KMP77].

4 About the System

The distribution of the partial evaluator INDY v1.8 contains the following files:

- `indyv1.8`: This is the source code of the partial evaluator.
- `indy_user_v1.8.tex`: The latex source for the User's Manual (this document).
- `Examples`: This folder contains several functional logic programs which can be specialized by the system.

The current implementation only requires SICStus Prolog `3.x` and is independent of the hardware platform.

The system is publicly available at:

`http://www.dsic.upv.es/users/elp/indy/indy.html`

We refer to [AFJV97, AFV98a, AFV98b, AHLV98] for more information about narrowing-driven partial evaluation and the INDY system.

Acknowledgements

We wish to thank Santiago Escobar, Michael Hanus, and Frank Steiner for their helpful suggestions, which have allowed us to substantially improve the INDY system.

This work has been partially supported by CICYT TIC 98-0445-C03-01 and by Acción Integrada hispano–alemana HA1997-0073.

References

[AAF+98] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In G. Levi, editor, *Proc. of Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503, 1998.

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.

[AFJV97] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.

[AFV96] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.

[AFV98a] M. Alpuente, M. Falaschi, and G. Vidal. Experiments with the Call-by-Value Partial Evaluator. Technical Report DSIC-II/13/98, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[AFV98b] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[AHLV98] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing, 1998. In preparation.

[Ant92] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.

[CD93] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[Fri85] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York, 1985.

[GS96] R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, February 1996.

[Han94a] M. Hanus. Combining Lazy Narrowing with Simplification. In *Proc. of 6th Int'l Symp. on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

[Han94b] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal of Computation*, 6(2):323–350, 1977.

[MG95] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.

[MR92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[SG95] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In J.W. Lloyd, editor, *Proc. of ILPS'95*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[SGJ96] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.