Towards Just-In-Time Partial Evaluation of Prolog

Carl Friedrich Bolz, Michael Leuschel, Armin Rigo

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
cfbolz@gmx.de, leuschel@cs.uni-duesseldorf.de, arigo@tunes.org

Abstract. We introduce a just-in-time specializer for Prolog. Just-in-time specialization attempts to unify of the concepts and benefits of partial evaluation (PE) and just-in-time (JIT) compilation. It is a variant of PE that occurs purely at runtime, which lazily generates residual code and is constantly driven by runtime feedback.
Our prototype is an on-line just-in-time partial evaluator. A major focus of our work is to remove the overhead incurred when executing an interpreter written in Prolog. It improves over classical offline PE by requiring almost no heuristics nor hints from the author of the interpreter; it also avoids most termination issues due to interleaving execution and specialization. We evaluate the performance of our prototype on a small number of benchmarks.

1 Introduction

Just-in-time compilers have been hugely successful in recent years, often providing significant benefits over traditional (ahead-of-time) compilers.[1] Indeed, much more information is available at runtime, some of which can be very expensive or impossible to obtain ahead-of-time by traditional static analysis. The biggest success story is possibly the Java HotSpot [22] just-in-time compiler, which now often matches or beats classical C++ compilers in terms of speed.

Dynamic languages have seen a recent surge in activity and industrial applications. Dynamic languages, due to their very nature, make traditional static analysis and compilation nigh impossible. Hence, a lot of hope is put into just-in-time compilation. Many techniques have been proposed; one of the main recent successes is the Psyco just-in-time specializer [24] for Python. In the best cases it can remove all the overhead incurred by the dynamic nature of the language. Its successor, the JIT compiler generator developed in the PyPy framework [26], is one of the bases for the present work, where we are interested in applying similar techniques to Prolog in general and partial evaluation of Prolog programs in particular.

[1] Even though there is of course room for both. Some applications do require static compilation techniques and validation, in the form of static analysis or type checking, which provides benefits over runtime validation.

Partial evaluation [16] is a technology that has been very popular for improving the performance of Prolog programs. Indeed, for Prolog, partial evaluation is more tractable than for imperative or object-oriented languages, such as C or Python. Especially for interpreters (one of the typical Prolog applications), speedups of several orders of magnitude are possible [2]. However, while some isolated successul applications exist, there is no widespread usage of partial evaluation technology. One problem is that the static input needs to be known ahead of time, whereas quite often the input that enables optimisations is only available at runtime. Also, one faces problems such as code explosion, as the specialized program sometimes needs to anticipate all possible runtime combinations in order not to loose static information. We argue that these problems can be solved by incorporating and adapting ideas from just-in-time compilation.

In this paper we present the technique of just-in-time partial evaluation along with an first prototype implementation for Prolog. The key contributions of our work are:

1. Just-in-time specialization allows us to decide which information is relevant for good optimisation; we can decide at runtime what is static and dynamic.
2. The specializer can inspect a runtime value at any point in time, and use it as a static value in order to partially evaluate the code that follows. We call this concept *promotion*.
3. Partial evaluation is done lazily; only parts really required are specialized, and compilation and execution are tightly interleaved.

Our paper is structured as follows. We discuss the problems that trouble classical partial evaluation in more detail in Sect. 2. The main mechanism of just-in-time partial evaluation is explained in Sect. 3. These goals are achieved with the use of "lazy choice points", which are the basic concept of this work. The control of our partial evaluator is discussed in Sect. 4. In Sect. 5 we examine the behaviour of our specializer for some examples. Related work and conclusion are presented in Sect. 6 and 7 respectively.

2 Problems of Classical Partial Evaluation

Partial evaluation [16] is a well-known source-to-source program transformation technique. It specialises programs by pre-computing those parts of the program which depend only on statically known input. The so-obtained transformed programs are less general than the original but can be much more efficient. In the context of logic programming, partial evaluation proceeds mostly by unfolding [19, 17] and is sometimes referred to as *partial deduction*.

Partial evaluation has a number of problems that have prevented it from being widely used, despite its considerable promise. One of the hardest problems of partial evaluation is the balance between under- and over-specialization. Over-specialization occurs when the partial evaluator generates code that is too specialized. This usually leads to too much code being generated and can lead to "code explosion", where a huge amount of code is generated, without significantly improving the speed of the code.

The opposite effect is that of under-specialization. When it occurs, the residual code is too general. This happens either if the partial evaluator does not have enough static information to make better code, or if the partial evaluator erroneously decides that some of the information it has is actually not useful and it then discards it.

The partial evaluator has to face difficult choices between over- and under-specialization. To prevent under-specialization it must keep as much information as possible, since once some information is lost, it cannot regained. However, keeping too much information is also not desirable, since it can lead to too much residual code being produced, without producing any real benefit.

Figure 2 shows an example where ECCE (a partial evaluator for pure Prolog [18]) produces bad code when doing partial evaluation. The code in the figure is a simple Prolog meta-interpreter which stores the outstanding goals in a list (the point of the `jit_merge_point` predicate is explained in Sect. 4.2. ECCE just ignores it). The interpreter works on object-level representations of append, naive reverse and a predicate replacing the leaves of a tree. When ECCE is asked to residualize a call to the meta-interpreter interpreting the `replaceleaves` predicate, it loses the information that the list of goals can only consist of `replaceleaves` terms. Thus eventually the residual code must be able to deal with arbitrary goals in the list of goals, which causes the full original program to be included in the residual code that ECCE produces (see predicates `solve__5`, `my_clause__6` and `append__7` in the residual code). This is a case of under-specialization (the code could be more specific and thus faster) and also of code explosion (the full interpreter is contained again, not only the parts that are needed for `replaceleaves`). We will come back to this example in Section 5.

A related problem are Prolog builtins. Many Prolog partial evaluators do not handle Prolog builtins very well. For example ECCE only supports purely logical builtins (which are builtins which could in theory be implemented by writing down a potentially infinite set of facts). Some builtins are just hard to support in principle, e.g., a partial evaluator cannot assume anything about the result of `read(X)`.

The fact that many classical Prolog partial evaluators do not support builtins, means that quite often user programs have to be rewritten in non-trivial ways – a time-consuming task.

3 Basics of Just-in-time Specialization

3.1 Basic Setting

We propose to solve the problems described in the previous section by *just-in-time partial evaluation.* The basic idea is that the partial evaluator is executed at runtime rather than ahead of time, interleaved with the execution of the

Original code:

```
solve([]).
solve([A|T]) :-
    jit_merge_point,
    my_clause(A,B), append(B,T,C), solve(C).

append([], T, T).
append([H|T1], T2, [H|T3]) :-
    append(T1, T2, T3).

my_clause(app([],L,L),[]).
my_clause(app([H|X],Y,[H|Z]),[app(X,Y,Z)]).
my_clause(replaceleaves(leaf, NewLeaf, NewLeaf),[]).
my_clause(replaceleaves(node(Left, Right), NewLeaf,
                        node(NewLeft, NewRight)),
          [replaceleaves(Left, NewLeaf, NewLeft),
           replaceleaves(Right, NewLeaf, NewRight)]).
my_clause(nrev([],[]), []).
my_clause(nrev([H|T], Z), [nrev(T, T1), app(T1, [H], Z)]).
```

Residual code for `solve([replaceleaves(A, B, C)])` by Ecce :

```
solve([replaceleaves(A, B, C)]) :- solve__2(A, B, C).
solve__2(leaf,A,A).
solve__2(node(A,B),C,node(D,E)) :- solve__3(A,C,D,B,E,[]).
solve__3(leaf,A,A,B,C,D) :- solve__4(B,A,C,D).
solve__3(node(A,B),C,node(D,E),F,G,H) :-
    solve__3(A,C,D,B,E,[replaceleaves(F,C,G)|H]).
solve__4(leaf,A,A,B) :- solve__5(B).
solve__4(node(A,B),C,node(D,E),F) :- solve__3(A,C,D,B,E,F).

solve__5([]).
solve__5([A|B]) :-
    my_clause__6(A,C),
    append__7(C,B,D),
    solve__5(D).
my_clause__6(app([],A,A),[]).
my_clause__6(app([A|B],C,[A|D]),[app(B,C,D)]).
my_clause__6(replaceleaves(leaf,A,A),[]).
my_clause__6(replaceleaves(node(A,B),C,node(D,E)),
             [replaceleaves(A,C,D),replaceleaves(B,C,E)]).
my_clause__6(nrev([],[]),[]).
my_clause__6(nrev([A|B],C),[nrev(B,D),app(D,[A],C)]).
append__7([],A,A).
append__7([A|B],C,[A|D]) :-
    append__7(B,C,D).
```

Fig. 1. Under-Specialization in Ecce for a Meta-Interpreter

specialized code (see Fig. 2). This allows it to observe the runtime behaviour of the program, giving it more information than a static specializer to base its decisions on. The approach we take is that the specializer produces some residual code upon demand, uses `assert` to put it into the Prolog database and then immediately runs the asserted code.[2] More residual code is produced later, if that becomes necessary. The details of when this process is started and stopped are described below.

The specialization process itself proceeds by interpretation of the Prolog source code. If a deterministic call to a user-predicate is interpreted, it is unfolded; otherwise specialization stops as described in the following section. If a call to a built-in is encountered, in the general case the call is skipped, i.e. put in the residual code; but a number of common built-ins have corresponding custom specialization rules and produce specialized residual code (or no code at all).

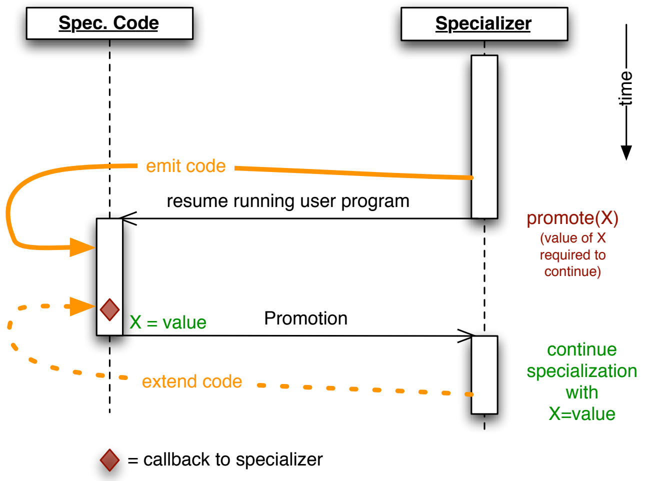
Spec. Code
Specializer
time
emit code
resume running user program
promote(X)
(value of X required to continue)
X = value
Promotion
extend code
continue specialization with X=value
= callback to specializer

Fig. 2. Interleaving of partial evaluation and execution of residual code

3.2 Promotion: Lazy Choice Points

The fundamental building block for the partial evaluator to make use of the just-in-time setting are *lazy choice points*. When reaching a choice point in the original program, the partial evaluator does not know which choice would be taken at runtime. Compiling all cases is undesirable, since that can lead to code

[2] On some Prolog systems, dynamically asserted code runs slower than static code. We can sometimes use workarounds, like `compile_predicates` in SWI-Prolog.

explosion. Therefore it inserts a *callback* to the specializer into the residual code and stops the partial evaluation to let the residual code run. When the callback is reached, the specializer is invoked again and specializes exactly the switch case that is needed by the running code. After specialization has finished, this new code is generated. See Figure 2 for a diagram of the interactions involved.

Another usage of lazy choice points by the partial evaluator is to get information about terms (X in the figure) which are required to obtain good specialization but are not available statically. When the actual runtime value (or some partial info about the value, like the functor and arity) of an unknown term is needed by the partial evaluator during specialization, specialization stops and a callback is inserted. Then the residual code generated so far is executed until the callback point is reached. When this happens, the value of the formerly unknown term *is* available (there are no unknown terms at runtime of course). At this point the specializer is invoked with the now known term and more code can be produced. We call this process *promotion*: it promotes a dynamic, unknown value to a static value available to the specializer.

Our approach is best illustrated by an example. Assume we have the following predicate:

```
negation(true(X), false(X)).
negation(false(X), true(X)).
```

First, our specializer rewrites this predicate in a pre-processing phase into the following form, which makes the choice point and first-argument indexing visible:

```
negation(X, Y) :- switch_functor(X, [
                    case(true/1,  (X = true(Z), Y = false(Z))),
                    case(false/1, (X = false(Z), Y = true(Z)))]).
```

The predicate `switch_functor` performs a switch on the functor of its first argument, the possible cases are described by the second argument. It could be implemented as a Prolog-predicate like this:

```
switch_functor(X, [case(F/Arity, Body)|_]) :-
    functor(X, F, Arity), call(Body).
switch_functor(X, [_|MoreCases]) :-
    switch_functor(X, MoreCases).
```

If the specializer encounters the call `negation(X)` it cannot know whether the functor of `X` will be `true` or `false` (if it would know the functor of `X` it could continue unfolding with the correct case immediately). Therefore the specialization process stops. At this point the following code has been generated and put into the clause database:

```
'$negation1'(X, Y) :-
    '$case1(X),  '$promotion1'(X, Y).
'$case1(true(_)).
```

```
'$case1(false(_)).
'$promotion1'(X, Y) :-
    functor(X, F, N),
    callback(F/N, '$promotion1', ...),
    '$promotion1'(X, Y).
```

The predicate `'$negation1'` is the entry-point of the specialized version of `negation`. The `'$case1'` predicate ensures that `X` is bound when `'$promotion1'` is called. The `'$promotion1'` predicate is the lazy choice point. At this point this predicate has only one clause, which is for invoking the specializer again. More clauses will be added later. If it is executed, partial evaluation will be resumed by calling `callback`, passing in the functor and the arity of the argument as information for specializing more code. Thus, one concrete clause of the choice point will be generated. After this is done, the promotion predicate is called again, which will execute the newly generated case.

The `callback` gets the functor and arity as its first argument. The second argument is the name of the predicate that should get a new clause added. The further arguments (shown only as `...` in the code above) contain the `Cases` in the `switch_functor` call, the continuation of what the partial evaluator still has to evaluate after the choice point. When `callback` is called, it will use its first argument to decide which of the cases it should partially evaluate further.

Let us assume that `'$negation1'` is first called with `false(X)` as an argument. Then `'$promotion1'` will be executed, calling `callback(false/1, '$promotion1', ...)`. This will resume the partial evaluator which then generates residual code only for the case where `X` is of the form `false(_)`. The residual code looks as follows:

```
'$promotion1'(false(Z), Y) :-
    !, Y = true(Z).
```

This code will be asserted using `asserta`, which means that it will be tried before the clause of `'$promotion1'` shown above. This has the effect that the next time `'$negation1'` is called with `false(X)` as an argument, this code will be used and no specialization will be performed. The cut is necessary to prevent the backtracking into the clause calling back into the specializer.

If the `'$negation1'` predicate is never actually called with an argument of the form `true(X)`, then the other case of the switch will never be specialized, saving time and memory. This might not matter for such a trivial case as the one above, but it strongly reduces specialization time and size of the residual code for more realistic cases (e.g. consider what happens if the body of `negation` contains calls to many predicates). If the other case will be specialized eventually, the residual code would look like this:

```
'$promotion1'(true(Z), Y) :-
    !, Y = false(Z).
```

This code will again be inserted into the database using `asserta` so that it too will be tried before the specialization case.

3.3 Other uses of lazy switches

The `switch_functor` primitive has some other uses apart from the obvious ones that it was designed for. These other uses also exploit the laziness of `switch_functor`, less so the switching part. One of them is to implement a lazy version of disjunction (the "`;`" builtin).

Another use of `switch_functor` is to support the `call(X)` builtin (which very few partial evaluators for Prolog do efficiently). This can be considered to be a switch of `X` over all the predicates in the program. Since `switch_functor` is lazy, only those predicates that are actually called at runtime need to be specialized. An example for this can be found in Sect. 4.3.

4 Control and Ensuring Termination

4.1 Code Generation and Local Control

So far we have not explained exactly how we generate the specialized code (apart from the lazy switches). Basically, we use the well-known partial evaluation framework as presented in [17] (which builds upon the original work in [19]). The control of partial evaluation for logic programs is often separated into local and global control [21], where the global control decides which calls are specialized and the local control performs the unfolding of those calls. In the simple setting described so far, we can simply view the local control of our just-in-time specializer as performing unfolding until a choice point is reached. At this point, the specializer stops and generates a resultant clause with a callback into the specializer (as explained in the last section). More precisely, the unfolding rule will recursively process the leftmost literal in a goal that has not yet been examined, with the following options:

1. If it is a `switch_functor` which is sufficiently instantiated, the proper case will be chosen.
2. If it is a `switch_functor` which is *not* sufficiently instantiated, unfolding stops and a call back into the specializer is inserted into the resultant, using a lazy switch, as explained in the previous section.
3. If it is a built-in, then the built-in is specialized, yielding a single computed answer along with a specialized version of the built-in to be put into the residual code. For non-deterministic built-ins, the computed answer is general enough to cover all solutions. Failure can also be detected, in which case the branch is pruned.
4. If the leftmost literal is a user-predicate, it will be simply unfolded. Observe that this is deterministic, as all choice points are encoded via the `switch_functor` primitive.

To ensure that the semantics are preserved in the presence of impure built-ins or predicates, we do not always left-propagate bindings (in case we do not select the leftmost literal). Bindings are left-propagated only until impure built-ins are met, using techniques from [23].

As our just-in-time specializer interleaves ordinary execution with code generation, the overall procedure cannot always terminate (namely when the user query under consideration does not terminate). However, we would like to ensure that if the unspecialized program itself terminates (existentially or universally respectively) then the just-in-time specializer process should also terminate (existentially or universally respectively). The above process does not fully guarantee this, as our just-in-time specializer may not detect that a call to a built-in in point 3 actually fails. This means that the just-in-time specializer would proceed specialization on a computation path which does not occur at runtime, which is a problem if this path is infinite.

One pragmatic solution is to ensure that the just-in-time specializer will maximally perform N specialization steps before executing residual code again. Every time the residual code is executed, the computation progresses. Therefore the presence of the just-in-time specializer specializer can only lead to a linear slowdown, which means in particular that it preserves termination behaviour.

4.2 Global Control

In some cases the specialization technique described so far can be sufficient. However, it does not reuse any of the generated residual code (i.e., the specializer produces a tree of predicate); what we want is to eventually obtain a jump to an already-specialized predicate, typically closing a loop. Instead of a tree, the final result should be an arbitrary graph of residual predicates.

In the current prototype, the specializer never tries to reuse existing residual code on its own. To trigger global control, the specialized program needs to request the attempt to reuse existing residual code by inserting a call to a special predicate called `jit_merge_point`. This predicate does nothing if executed normally, but is dealt with by the partial evaluator in a special way. For an example usage, see Figure 2.

The need for this sort of explicit hint is clearly not ideal, but we felt that it simplified implementation enough to still be a good choice, given that most programs with an interpretative nature need to contain only one call or a small number of calls to this predicate. We plan to find ways of automatically placing this call in the future.

At the places where a call to `jit_merge_point` is seen, the partial evaluator tries to reuse an already existing residual predicate. It does this by comparing the continuations that the partial evaluator has currently with those it had at earlier calls to `jit_merge_point`. If those continuations are similar enough the partial evaluator inserts a call to the residual predicate produced earlier and stops the partial evaluation process. The exact conditions when this is possible are outside the scope of this paper and are fully explained in [3]. In summary, the procedure remembers which parts of the term have been used to resolve choice points; parts which did not contribute in any way to improve the specialisation are thrown away.[3]

[3] In some sense this can be seen as an evolution of the generalisation operator from [12] to a just-in-time specialisation setting.

In the next subsection we present a simple example which illustrates this aspect of our system, and also highlights the potential of our just-in-time specialization compared to traditional partial evaluation.

4.3 A Worked Out Example: Read-Eval-Print Loop

As an showcase example we wrote a minimal read-eval-print loop for Prolog, which can be seen in Fig. 3. Most classical partial evaluators have a hard-time producing good code for `read_eval_print_loop`, because after `read(X)` the value of `X` is unknown, which makes it impossible to figure out which predicate `call(X)` will ultimately call.

For our prototype this represents no real problem. The functor of `X` can be promoted, thus observing at runtime which predicate is to be called. Subsequently, this predicate can be specialized. Fig. 3 also shows an example session as well as the residual code that our prototype generated for this session (note that the clauses for `'$callpromotion1'` are shown in the order in which they are in the database, which is the reverse order in which they have been generated).

5 Experimental Results

To get some impression for the performance of our dynamic partial evaluation system, we ran a number of benchmarks. We compared the results with those of Ecce [18], an automatic online program specializer for pure Prolog. The experiments were run on a machine with a 1.4 GHz Pentium M processor and 1GiB RAM, using Linux 2.6.24. For running our prototype and the original and specialized programs we used SWI-Prolog Version 5.6.47 (Multi-threaded, 32 bits). Ecce was used both in "classic mode" which uses normal partial evaluation and in "conjunctive mode" (which uses conjunctive partial deduction with characteristic trees and homeomorphic embedding; see [9]). Conjunctive partial evaluation is considerably more powerful, but also much more complex.

Figure 5 presents five benchmarks. The first three are examples for a typical logic programming interpreter with one and also with two levels of interpretation. The fourth example is a higher-order example, using the meta-predicates `=..` and `call`. Finally, the fifth is a small interpreter for a dynamic language. Note that "spec" refers to the specialization time and "run" to the runtime of the specialized code. For Ecce the specialization time was not measured.

Our prototype is in all cases faster than the original code, but also in all cases slower (by a factor between 2 and 8) than Ecce in conjunctive mode. On the other hand, our prototype is faster than Ecce in classical mode in two cases. These are not bad results, considering the relative complexity and maturity of the two projects. Our prototype is rather straightforward. It was written from scratch over the course of some months and consists of about 1500 lines of Prolog code. On the other hand, Ecce is a mature system that employs serious theoretical results and consists of about 25000 lines of Prolog code.

Repl code and some example predicates:

```
read_eval_print_loop :-
    jit_merge_point,
    read(X),
    call(X),
    print(X),
    nl, read_eval_print_loop.

% example predicates
f(a). f(b). f(c).
g(X) :- h(Y, X), f(Y).
h(c, d).
k(_, _, _) :- g(X), g(X).
```

Example session:

```
|: f(c).
f(c)
|: g(X).
g(d)
|: fail.

No
```

Produced residual code (promotion specialization cases not shown):

```
'$entrypoint1' :-
        read(A),
        '$callpromotion1'(A).

'$callpromotion1'(fail) :- !,
        fail.
'$callpromotion1'(g(A)) :- !,
        A=d,
        print(g(d)),
        nl,
        '$entrypoint1'.
'$callpromotion1'(f(A)) :- !,
        '$case1'(A), '$promotion1'(A).

'$case1'(a). '$case1'(b). '$case1'(c).
'$promotion1'(c) :- !,
        print(f(c)),
        nl,
        '$entrypoint1'.
```

Fig. 3. A Simple read-eval-print-loop for Prolog

	Experiment	Inferences	CPU Time	Speedup
A vanilla meta-interpreter [14, 20] running append with a list of 100000 elements. The interpreter can be seen in Figure 2.	Vanilla - Append			
	original	500008	0.35 s	1.0
	JIT PE, spec+run	281842	0.13 s	2.69
	JIT PE, run	200016	0.11 s	3.18
	ecce classic	100003	0.03 s	11.67
	ecce conjunctive	100003	0.03 s	11.67
The vanilla interpreter running itself running append with a list of 100000 elements.	Vanilla - Vanilla - Append			
	original	2000023	1.42 s	1.0
	JIT PE, spec+run	1577228	0.66 s	2.15
	JIT PE, run	700020	0.32 s	4.44
	ecce classic	100003	0.04 s	35.5
	ecce conjunctive	100003	0.04 s	35.5
The vanilla interpreter running `replaceleaves`, see Figure 2. Input was a full tree of depth 18.	Vanilla - Replace Leaves			
	original	2621438	2.76 s	1.0
	JIT PE, spec+run	2493636	1.77 s	1.56
	JIT PE, run	2097162	1.58 s	1.75
	ecce classic	2097074	2.64 s	1.05
	ecce conjunctive	589825	0.78 s	3.54
A higher order example: `reduce` in Prolog using `=..` and `call`. This is summing a list of 100000 integers, knowing statically the functor that is used for the summation.	Reduce - Add			
	original	1492586	16.73 s	1.0
	JIT PE, spec+run	5082861	3.53 s	4.74
	JIT PE, run	5000014	3.24 s	5.16
	ecce classic	1134504	8.5 s	1.97
	ecce conjunctive	2000001	1.85 s	9.04
An interpreter (~100 lines of Prolog) for a small stack-based dynamic language. The benchmark is running an empty loop of 100000 iterations.	Stack Interpreter			
	original	2100010	3.13 s	1.0
	JIT PE, spec+run	5699992	1.46 s	2.14
	JIT PE, run	200019	0.08 s	39.13
	ecce classic	100003	0.05 s	62.6
	ecce conjunctive	100003	0.04 s	78.25

Fig. 4. Experimental Results

As we have also seen in Section 2 the third benchmark is one where Ecce in classical mode produces rather bad code. This can be seen in the benchmark results as well, there is nearly no speedup when compared to the original code. Our prototype has the same problem, it also loses the information that all the goals in the goal list are `replaceleaves` calls. However, in our case this is not a problem, since that information can be regained with a promotion, thus preventing code explosion and under-specialization.

6 More Related Work

Promotion is a concept that we have already explored in other contexts. Psyco is a run-time specializer for Python that uses promotion (called "unlift" in [24]).

Similarly, the PyPy project [25, 4], in which all three authors are also involved, contains a just-in-time specialization system built on promotion [26].

Greg Sullivan describes a runtime partial evaluator for a small dynamic language based on lambda calculus [27]. Sullivan [27] further distinguishes two cases (quoting): *"Runtime partial evaluation [...] defers some of the partial evaluation process until actual data is available at runtime. However the scope and actions related to partial evaluation are largely decided at compile time. Dynamic partial evaluation goes further, deferring all partial evaluation activity to runtime."* Using this terminology, our system does dynamic partial evaluation.

One of the earliest works on runtime specialization is Tempo for C [8, 7]. However, it is essentially an offline specializer "packaged as a library"; decisions about what can be specialized and how are pre-determined.

Another work in this direction is DyC [13], another runtime specializer for C. Specialization decisions are also pre-determined, i.e. dynamic partial evaluation is not attempted, but "polyvariant program-point specialization" gives a coarse-grained equivalent of our promotion. Targeting the C language makes higher-level specialization difficult, though (e.g. `malloc` is not optimized).

Polymorphic inline caches (PIC) [15] are very closely related to promotion. They are used by JIT compilers of object-oriented language and also insert a growable switch directly into the generated machine code. This switch examines the receiver types for a message for a particular call site. From that angle, promotion is an extension of PICs, since promotions can be used to switch on arbitrary values, not just receiver types.

The recent work on trace-based JITs [11] (originating from Dynamo [1]) shares many characteristics of our work. Trace-based JITs concentrate on generating good code for loops, and generate code by observing the runtime behaviour of the user program. They also only generate code for code paths that are actually followed by the program at runtime. The generated code typically contains guards; in recent research [10] on Java, these guards' behavior is extended to be similar to our promotion. This has been used by several implementations to implement a dynamic language (JavaScript) [5, 6].

7 Conclusion and Future Work

In this paper we drew explicit parallels between partial evaluation and just-in-time compilers. We showed with a Prolog prototype of a just-in-time partial evaluator that these two domains might benefit a lot from a synergy. In particular, inspired by Polymorphic Inline Caches, we have developed the notion of *promotion* for partial evaluation. We hope that our approach can help address several fundamental issues that so far prevent classical partial evaluation to reach its fullest potential: code explosion, termination, full Prolog support, and scalability to large programs.

Due to the use of promotion our just-in-time partial evaluator works reasonably well for interpreters of dynamic languages and generally in situations where information that the partial evaluator needs is only available at runtime. This is an advantage that a classical partial evaluator can never possess for fundamental reasons. We have not tried our prototype on really large programs yet, so it remains to be seen whether it works well for these.

There are some downsides to our approach. In particular promotion needs a Prolog system that supports `assert` well, since the whole approach depends on that in a crucial manner. We have not yet evaluated our work on any Prolog system other than SWI-Prolog (which supports `assert` rather well). In the future we would like to support other Prolog platforms like Ciao Prolog or Sicstus Prolog as well.

Global control is another area that still needs further work. We plan to explore ways of inserting the `jit_merge_points` automatically. Furthermore, the global control strategy needs further evaluation and possible refinement.

Finally we need to take a look at the speed of the partial evaluator itself, which we so far disregarded completely. Since partial evaluation happens at runtime it is necessary for the partial evaluator to not have too bad performance.

References

1. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35:1–12, 2000.
2. S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *Proceedings PEPM'04*, pages 190–199. ACM Press, 2004.
3. C. F. Bolz. *Automatic JIT Compiler Generation with Runtime Partial Evaluation*. Master thesis, Heinrich-Heine-Universität Düsseldorf, 2008. http://www.stups.uni-duesseldorf.de/thesis_detail.php?id=14.
4. C. F. Bolz and A. Rigo. How to *not* write a virtual machine. In *Proceedings of 3rd Workshop on Dynamic Languages and Applications (DYLA 2007)*, 2007.
5. M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.
6. M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual ExecutionEnvironments*, pages 71–80, Washington, DC, USA, 2009. ACM.
7. C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volansche. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 54–72. Springer, 1996.
8. C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *POPL*, pages 145–156, 1996.
9. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.
10. A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

11. A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.
12. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
13. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248:147–199, 2000.
14. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
15. U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.
16. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
17. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
18. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
19. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
20. B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.
21. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
22. M. Paleczny, C. Vick, and C. Click. The java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, Monterey, California, 2001. USENIX Association.
23. S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, pages 199–213, University of Manchester, 1992. Springer-Verlag.
24. A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In N. Heintze and P. Sestoft, editors, *PEPM*, pages 15–26. ACM, 2004.
25. A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.
26. A. Rigo and S. Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
27. G. T. Sullivan. Dynamic partial evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 238–256. Springer-Verlag, 2001.