INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT
DÜSSELDORF

Automatic JIT Compiler Generation with Runtime Partial Evaluation

Carl Friedrich Bolz

Masterarbeit

Beginn der Arbeit:	26. März 2008
Abgabe der Arbeit:	26. September 2008
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Volker Aurich

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 26. September 2008

Carl Friedrich Bolz

Acknowledgments

I would like to express my deepest gratitude to Armin Rigo for his help with the implementation of the partial evaluation prototype. This thesis owes a lot to his input. I would also like to thank Michael Leuschel for many fruitful and invigorating discussions and for his constant support.

Abstract

Dynamic languages are enjoying increasing popularity recently, due to their power, simplicity and flexibility. However, since many popular dynamic languages are implemented using straightforward interpreters, their performance is not up to par with more static languages. This situation can be improved with just-in-time compilers, but those are hard to implement, hard to maintain and hard to change.

Partial evaluation is a technique to make the construction of compilers easier. It can automatically generate a compiler for a certain language, given an interpreter for the same language. Despite considerable apparent promise it has so far failed to make a large impact due to various problems. In addition, it can only be used for ahead-of-time compilation, which does not work well for dynamic languages.

In this thesis we explore the use of partial evaluation at runtime for interpreters written in Prolog. For this we wrote a Prolog prototype that performs partial evaluation interleaved with actual program execution. Doing so solves some problems of classical partial evaluation and makes it at the same time applicable to dynamic languages.

Contents

1 Introduction and Motivation

One of the oldest high-level programming language that is still in use today is Lisp [McC60][1]. Lisp pioneered language features that are typical for what are today called *dynamic programming languages*. The great flexibility of these languages, their versatility and the ease of development has always made them very attractive. Due to various reasons, dynamic languages have become even more popular in the last years. One reason for that is the increasing power of computers, making up for the relative slowness of dynamic language implementations and thus making dynamic languages suitable for many problem domains. Another reason is the invention of the web, on which dynamic languages have been used from quite early on, both on the client-side (directly in the browser with JavaScript, but also more recent developments like Flash, AIR or Silverlight) and on the server-side with Perl, Python, Ruby etc.

Since the beginning it has been of great concern and a frequent research topic as to how dynamic languages can be developed that work at the highest possible speeds. This has turned out not to be easy, as dynamic languages do not really benefit much from the application of traditional ahead-of-time compiler techniques. This is due to the fact that the programs themselves typically do not contain any type information. Thus a compiler cannot remove the method dispatch overhead or the overhead of garbage-collection, which are the real speed problems a dynamic language incurs. Therefore research has turned to runtime techniques such as just-in-time compilation to lower execution overhead. This endeavour has led to some impressive results, however at the price of enormous complexity. A good JIT compiler is usually a large and intricate piece of code that is hard to change and to maintain.

Compilers are generally complex pieces of software, whether classical ahead-of-time compilers or their runtime equivalents. Various techniques exist to ease their implementation, from ways to describe syntax such as YACC or JavaCC to table-driven, processor independent assembler generation. However, the fact remains that writing a compiler to another language is a fairly poor way to encode the semantics of a language. One much simpler way to do this encoding is to write a simple interpreter for a language. This fact has been extensively used for dynamic languages; most of the currently popular dynamic languages are implemented using simple bytecode interpreters.

In the 70's, starting from this observation, Futamura conjectured [Fut71] that it should be possible and even easy to turn an interpreter into a compiler in an automated manner using a process that he called *partial evaluation*. This initiated a lot of research on the subject, particularly in the functional and logic programming communities. One language on which much research has been focused is Prolog, which is a very good language in which to do partial evaluation. Despite many early successes, partial evaluation has failed to gain widespread usage in practice for various reasons, the main problem being that while the techniques work in general, it can be very hard to control them in exactly the right way. This leads to code explosion (i.e., compilers that generate too much code) and over- and under-specialization (which lead to too much code and bad performance, respectively).

[1]Fortran is even older. Of course both have changed significantly since their inception

Partial evaluation has been particularly unsuccessful for dynamic languages, for the same reasons why classical ahead-of-time compilers fail to work for them. The PyPy project [PyP] explored ways to generate just-in-time compilers automatically from interpreters by performing partial evaluation at runtime. Moving partial evaluation to runtime turns out to solve many problems that trouble classical partial evaluation while at the same time making it more applicable to dynamic languages.

This thesis was directly inspired by the PyPy project. We try to explore partial evaluation at runtime further by building a Prolog prototype. Due to its simplicity Prolog is particularly conducive to experiments with partial evaluation. The contributions of this work are:

- The first implementation of a dynamic partial evaluator for a large subset of Prolog, including many builtins
- An efficient implementation of *promotion* in Prolog
- Techniques that allow the prevention of under-specialization without resorting to heuristics

This thesis is structured as follows: Section 2 gives an overview of dynamic programming languages and their implementation, describes classical partial evaluation and shows some of its problems. In Section 3 we describe the basics of dynamic partial evaluation and introduce our Prolog runtime partial evaluation prototype in detail. Section 4 explains how lazy choice point handling works. Section 5 describes how we support large parts of the Prolog language in our partial evaluator. Section 6 describes how the partial evaluator reuses older generated code. Section 7 describes the partial evaluation algorithm in some detail. In Section 8 we describe benchmarks comparing our system to a classical partial evaluator, ECCE . In Section 9 we compare our work to other approaches and in Section 10 we conclude and provide a view of possible future developments.

2 Background

2.1 Dynamic Languages and their Implementation

While not everyone agrees about what constitutes a dynamic language , commonly agreed on elements include dynamic (but strong) typing, garbage collection and reflection. Many dynamic languages try to go to extremes and allow the changing of as many aspects of a program's behaviour at runtime as possible. Notable examples of dynamic languages are Smalltalk, Lisp, Python, Ruby, JavaScript.

Another, less typical, dynamic language is Prolog. Prolog is a declarative logic programming language invented by Alain Colmerauer, Philippe Roussel and Robert Kowalski in 1972 [CR93]. The execution of Prolog programs is based on first order predicate calculus. The Prolog language is formally defined in the ISO Prolog standard [DCED96].

While Prolog differs from the typical dynamic languages popular today (being neither imperative nor object-oriented), it still shows many of the characteristic traits of a dy-

Figure 1: Python Example: Summing the Elements of an Iterable

```python
def sumlist(l, initial):
    result = initial
    for element in l:
        result += element
    return result
```

namic language. It is dynamically typed since there is no way to know to which sort of value a logic variable will be bound at runtime and it has automatic memory management. Prolog also contains a number of reflective facilities. Obvious ones include `call`, which allows the insertion of dynamically constructed goals into the goal stack. Furthermore there are builtins which allow the inspection of the current database (i.e., the program being run) like `listing` and also the changing of it by adding new facts, like `assert` and `retract`.

Most of today's Prolog implementations use a more or less heavily modified version of Warren's abstract machine (WAM) [War83] to interpret Prolog programs. The WAM is a virtual machine designed for running Prolog programs by compiling them first to the bytecode instruction set of the WAM and then executing those by running them on an emulator. Many dynamic languages use similar implementation strategies. Interpreters (whether they are bytecode-based or interpret some sort of tree) are simple to write, simple to maintain and simple to enhance. On the other hand they are said not to reach optimum speeds.

If it actually becomes necessary to get speed higher than what a good interpreter provides, things become more complicated. Attempts to apply classical compiler-techniques to dynamic languages have not yielded significant performance improvements. The problem with this approach is that the source code of dynamic languages does not really contain enough information to produce efficient machine code. See, for example, the Python code in Figure 1 Just by looking at the code, the compiler cannot know what the types of the elements in `l` are and therefore has no idea how to perform the addition on them. Since addition in Python works on any number of types due to operator overloading, the compiled code must perform the same costly lookup that the interpreter would have performed too. Thus the only benefit of a compiler when applied to a dynamic language is mostly only that it can remove the overhead of bytecode-dispatching, which gives speedups between 20% and two times faster. See [Sal04] for an overview of approaches to compiling Python. While this is not a bad result, it is not enough to close the gap in performance to statically typed compiled languages, which are typically two orders of magnitude faster that Python.

To get really interesting results, a just-in-time compilation approach is needed. A JIT compiler has the benefit that it can do runtime profiling to concentrate its efforts on functions or methods that are executed often. This is the general approach of a JIT to get more information than what the static compiler has available. It observes the running program and thus gets to know how the program is likely to behave. The information obtained in this way is not necessarily correct, so the generated code needs to work in other cases too.

However, those cases are unlikely, so their performance can be slow without downsides[2].

For the Python code shown in Figure 1, a (hypothetical) JIT-compiler could for example observe that the elements of `l` are typically normal integers. Then it could produce special code for this case and general code which works for other cases. The special code would be very efficient, since the integer addition could be implemented by directly using the processor.

Writing a JIT-compiler, just like writing a normal ahead-of-time compiler, has many downsides. It is more complicated to write a program that produces code that does the things the user program intends than writing an interpreter, which is a program that directly does the thing the user program intends. The compiler needs to contain the language semantics of the implemented language and the indirect encoding of these semantics that a compiler has are usually much more verbose than the direct encodings of these semantics in an interpreter. This makes a compiler harder to write and also harder to understand. If the language evolves, it is much harder to adapt a compiler than to adapt an interpreter.

A good example for this lack of maintainability of a compiler is the Psyco project [Rig, Rig04]. Psyco is a hand-written just-in-time compiler for the Python language. It was written when Python 2.2 was current and Python 2.1 still widely used. It has not been changed to keep up with some of Python's more recent features due to the complex nature of the changes needed.

2.2 Classical Partial Evaluation

In 1971 Yoshihiko Futamura published a paper [Fut71] that proposed a technique to automatically transform an interpreter of a programming language into a compiler for the same language. This would solve the problem of having to write a compiler instead of a much simpler interpreter. He proposed to use partial evaluation to achieve this goal. He defined partial evaluation along the following lines:

Given a program P with $m + n$ input variables $s_1, ..., s_m$ and $d_1, ..., d_m$, the partial evaluation of P with respect to concrete values $s'_1, ..., s'_m$ for the first m variables is a program P'. The program P' takes only the input variables $d_1, ..., d_n$ but behaves exactly like P with the concrete values (but is hopefully more efficient). This transformation is done by a program S, the partial evaluator, which takes P and $s_1, ..., s_m$ as input:

$$S(P, (s'_1, ..., s'_m)) = P'$$

The variables $s_1, ..., s_m$ are called the *static* variables, the variables $d_1, ..., d_n$ are called the *dynamic* variables; P' is the *residual code*. Partial evaluation creates a version of P that works only for a fixed set of inputs for the first m arguments. This effect is called *specialization* (the terms "specialization" and "partial evaluation" will be used interchangeably in this thesis).

[2]Many JIT compilers don't even generate code for the unlikely general case, but put in a check as to whether the general case is needed and fall back to an interpreter.

Figure 2: Python Partial Evaluation Example: Exponentiation

Original function:

```
def power(d, s):
    if s < 0:
        s = -s
        invert = True
    else:
        invert = False
    result = 1
    while s > 0:
        result = result * d
        s -= 1
    if invert:
        return 1 / result
    return result
```

Residual code for `s = 4`:

```
def power_4(d):
    result = 1
    result = result * d
    result = result * d
    result = result * d
    result = result * d
    return result
```

When P is an interpreter for a programming language, then the $s_1, ..., s_m$ are chosen such that they represent the program that the interpreter is interpreting and the $d_1, ..., d_n$ represent the input of this program. Then P' can be regarded as a compiled version of the program that the chosen $s'_1, ..., s'_m$ represent, since it is a version of the interpreter that can only interpret this program. Now once the partial evaluator S is implemented, it is actually enough to implement an interpreter for a new language and use S together with this interpreter to compile programs in that new language.

A valid implementation for S would be to just put the concrete values into P to get P', which would not actually produce any performance benefits compared with directly using P. A good implementation for S should instead make use of the information it has and evaluate all the parts of the program that actually depend only on the $s_1, ..., s_m$ and to remove parts of P that cannot be reached given the concrete values.

2.3 Partial Evaluation of Prolog Programs

Partial Evaluation is quite natural in the context of a logic programming language and very similar to normal evaluation. See [LS91] for an overview. A partial evaluator needs to deal with known (static) and unknown (dynamic) information in the program, and Prolog already has this distinction (bound versus unbound logic variables) built into the language. Normal Prolog programs already deal with some amount of unknown in-

Figure 3: Partial Evaluation of Prolog Predicates Using the Prolog Interpreter
Original predicates:

```
f(X, Y) :- p(X), q(X, Y).

p(a). p(b).

q(a, Y) :- r(Y).
q(b, Y) :- t(Y).
q(c, Y) :- s(Y).

r(ra). r(rb).
t(ta). t(tb).
s(sa). s(sb).
```

Running `findall(f_a_Y(a, Y), f(a, Y), L)` yields `L = [f_a_Y(a, ra), f_a_Y(a, rb)]`, therefore the residual code is:

```
f_a_Y(a, ra).
f_a_Y(a, rb).
```

Even not giving any static information at all can produce better code. Running `findall(f_X_Y(X, Y), f(X, Y), L)` yields `L = [f_X_Y(a, ra), f_X_Y(a, rb), f_X_Y(b, ta), f_X_Y(b, tb)]`

```
f_X_Y(a, ra).
f_X_Y(a, rb).
f_X_Y(b, ta).
f_X_Y(b, tb).
```

formation. Therefore the normal evaluation process can sometimes be used to directly perform partial evaluation. See figure 3 for an example. Of course this breaks down as soon as builtins are used or as soon as the predicate to be evaluated gives infinitely many solutions for the partially known goal.

One of the most important mechanisms of partial evaluation in Prolog is *unfolding*. It is similar to inlining in normal compilers. When a goal in the body of a clause is unfolded, the goal is replaced by the body of the called function. If the called function has more than one clause, all the clauses of the called function have to be considered.

Let us look at an example. Given the following Prolog clauses:

```
f(a, X) :- g(X).
f(a, X) :- h(X, Y), i(Y).
f(b, X) :- j(X).

func(X) :- f(a, X), k(X).
```

If we unfold `f` in the body of `func`, we get:

```
func(X) :- g(X), k(X).
func(X) :- h(X, Y), i(Y), k(X).
```

A partial evaluator in Prolog needs to control the unfolding properly. Usually the distinction between local and global control is made. Local control decides which goals should be unfolded. Global control needs to ensure that the goals that are not unfolded have suitably specialized implementations. See [LB02] for an overview of various control strategies that are used to achieve this.

2.4 Problems of Classical Partial Evaluation

Classical partial evaluation has a number of problems that have prevented it from being widely used, despite its considerable apparent 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. This happens when the partial evaluator tries to keep all the static information that it was given, even though not all of it is actually relevant to producing good 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 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.

See Figure 4 for an example where ECCE (a classical partial evaluator for pure Prolog [LMDS98]) produces bad code. The code in the figure is a simple Prolog meta-interpreter which keeps the list of goals that are left to do in a list. In additions there are the programs for append, naive reverse and a predicate replacing the leaves of a tree by something else. 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 leads to the fact that the full original program is contained 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 8.

A related problem are Prolog builtins. Many Prolog partial evaluators do not handle Prolog builtins very well. For example ECCE [LMDS98] 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

Figure 4: Under-Specialization in ECCE for the Meta-Interpreter

```
% 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).
```

partial evaluator cannot assume anything about the result of `read(X)`.

The fact that many classical Prolog partial evaluators do not support builtins well makes applying them to a large existing interpreter a time-consuming task, since often the interpreter needs to be rewritten to no longer use unsupported builtins.

When using a partial evaluator on an interpreter for a dynamic languages, essentially the same problems as with static compilers for such languages occur. The partial evaluator does not have more information than a static compiler, i.e., it only knows the program itself. Therefore it cannot produce good residual code for dynamic languages either.

3 Dynamic Partial Evaluation

In this work we try to approach a solution to some of the problems of partial evaluation by performing partial evaluation at runtime. This has been attempted several times before. Examples are Tempo [CN96, CHN$^+$96] and DyC [GMP$^+$00], both doing partial evaluation of the C language at runtime. Sullivan [Sul01] differentiates between two flavours of partial evaluation at runtime: *"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."*

Our work explores specifically dynamic partial evaluation. Dynamic partial evaluation has many benefits over classical partial evaluation (and also over runtime partial evaluation). The partial evaluator can decide at runtime which parts of the input arguments should be static and dynamic. Since the partial evaluator is running interleaved with the program being partially evaluated, the partial evaluator can always obtain more information (i.e., make more things static) by observing the running program. This is particularly helpful, since it allows the partial evaluator to forget information when it is too expensive to keep it, since this information can be re-gained later if this becomes necessary. This makes the implementation of the partial evaluator (but also the theory behind it) quite straightforward.

Another advantage is that the partial evaluator can be *lazy*. It only compiles those code paths that are really used at runtime, preventing the generation of a lot of code that would never actually be executed. Thus the effort of the partial evaluator can be concentrated on the places where they are actually needed. A classical partial evaluator of course cannot do this and must compile all possible code paths, which is one of the reasons for code explosion.

Of course doing all the partial evaluation work at runtime slows things down compared to ahead-of-time partial evaluation. However, due to the increasing power of computers, doing dynamic partial evaluation is more feasible than twenty or even ten years ago. The trade-offs between classical partial evaluation, runtime partial evaluation and dynamic evaluation are evolving.

3.1 Informal Overview

The general approach of our prototype is to perform partial evaluation at runtime, interleaved with actual execution of the compiled code which makes it possible to feed back information from the runtime behaviour of the program to the compiler. This feeding back is done by a primitive operation we call *promotion*. Promotion is one of the central concepts of this work, it is also used to control the interleaving of compilation and execution.

In this section we first give an informal overview of the partial evaluation process as performed by our prototype. More detailed and formal accounts of the steps of the process will be given in later sections.

The whole compilation process is started by the user of the partial evaluator by calling a special predicate named `compile_and_call(Term)` which behaves exactly like the builtin `call(Term)` but first produces residual code for the predicate that is called in this manner and then executes the newly generated residual code. When starting to produce code in this way, there is no actual specialization of the builtin happening. The functor of `Term` is the only information given to the partial evaluator about what it should produce code for. In particular, no information about the arguments is passed to the compiler. This is not a problem, indeed it ensures that the most general possible code is generated – and due to the runtime-nature of the process, information about the arguments can always be obtained, if needed. Whenever the compiler needs more information for proceeding, it observes the runtime-behaviour of the program to gain that information. Only then are non-general paths created.

If the called user-predicate consists of more than one clause, a preprocessing step transforms these several clauses into one clause, making all choice points explicit (one can think about this as using the `';'` builtin to join the separate clauses into one, although in practice it is more complicated than that, see Sect. 5). This frees the specializer from having to support several clauses per predicate.

After this pre-processing step, the actual specialization process begins. The specializer partially evaluates the code from left to right, in the same order that a normal Prolog interpreter would evaluate the code. During this process the specializer can have varying degrees of information available about the logic variables that occur. They can be completely known to the specializer, they can be completely unknown or the specializer knows that a variable is definitely unbound at a certain point. Also combinations are possible, e.g., a term with functor `f` whose first argument is known to be the atom `a` while the second argument is unknown and the third argument is known to be a free variable.

A very important concept of this work is how choice points in the program are handled. When a choice point of any form is hit, two cases are possible:

1. If the specializer has enough information available to decide that at runtime only one of the choices applies it will continue specialization with this choice.

2. If not enough information is available to make this decision, the specializer stops. It does not specialize all the paths of the choice point, because that would potentially generate code for paths that are never actually taken. Instead, it generates a stub

that will call back to the compiler, once execution actually reaches the stub. Then only the path that will be executed is compiled.

Choice points are also the way in which execution and compilation are interleaved. After it has been started, compilation will eventually hit a choice point and thus has to stop. At this point the code that was generated so far will be executed, until execution hits a choice point in the residual code and the case to choose has not been compiled.[3] Then the compiler is called again to compile the code for that particular case. After that is done, the new code is executed, and so on.

There is also a very different view on lazy choice point handling. Which choice is taken (and thus compiled) at runtime gives the partial evaluator information about the runtime behaviour of the program. This is particularly the case when the various cases of a choice point deconstruct the same term in different ways. Under this view on lazy choice points they become very similar to what is called *promotion* in the context of PyPy's dynamic partial evaluation work [RP07]. Therefore will use this term as well.

When the partial evaluator reaches a call to a builtin there are various possibilities. When the builtin is side-effect free, and the arguments are sufficiently known, the builtin can actually be evaluated at compile-time and no residual code will be produced. When the builtin has side-effects or the arguments are not sufficiently known, a call to that builtin will be put into the residual code.

The process as described so far would never actually re-use existing code and existing specialized predicates. However, this is of course desirable, so there is another mechanism to do that. At certain points (so called *merge points*) the specializer tries to stop generating new code and instead insert a call to an existing specialized predicate. If it manages to do this, we call it a successful *merge*. A merge will insert a call to an earlier predicate and then stop generating more code, which in Prolog is essentially equivalent to creating a loop, due to tail recursion.

A merge is allowed to happen when two conditions are met. The first (and obvious) condition is that the merge is *sound*: The behaviour of the predicate that we insert a call to must be exactly the same as what would happen if we just continued to specialize. The second, less obvious but essential condition, is that the merge must be *efficient*, which informally means that the loop that will be created through the merge must not contain any "obviously inefficient" operations. More on this later in Sect. 6.

If a merge is not possible, because none of the predicates generated earlier are suitable, specialization cannot stop and more code needs to be generated. This new code is inserted into a new predicate to make sure that if a similar merge is attempted later in the specialization process, it will succeed.

The actual partial evaluation is performed by a special interpreter which executes the to-be-specialized program and produces residual code as a side-effect. The main predicate of this interpreter looks like this[4]:

[3]This can happen when the choice point has never been reached yet or when it has been reached but other cases have been compiled. These two possibilities are effectively identical.

[4]The code examples of the partial evaluator are slightly simplified.

```
interpret(Continuation, ResidualCode, ResidualCodeHead, History)
```

The `Continuation` contains the term that is currently interpreted, the `ResidualCode` contains the residual code which was generated so far and `History` is a high-level description of the residual code, which is used for merging (see Section 6).

The `Continuation` term is a ground representation [HG98] for representing the currently interpreted term. This means that it is represented using terms of the following forms:

- `varbox(RtVar)` represents a term about which nothing at all is known. `RtVar` is a variable which is used to represent this unknown term in the residual code.
- `freebox(RtVar)` represents a variable which is known to be free. `RtVar` is used as for `varbox`.
- `termbox(Functor, Args, Id)` represents a term that is fully known. `Functor` is an atom describing the functor of the term, `Args` is a list of the argument-terms, again in ground representation and `Id` is a unique atomic identifier that is used to figure out which free variables are sharing.

For brevity, we introduce a short notation for terms in ground representation that is close to the standard Prolog syntax for terms. A `varbox` is represented by a capital letter (for example X), a `freebox` by a capital letter with a line above it (for example $\bar{X}$), a `termbox` is written in standard Prolog term notation using a lower-case letter and parenthesis, but in addition has an index with the identity, if that is relevant in an example (for example $f_1(a_2, b_3)$).

The whole partial evaluation process is started from Prolog using the predicate `compile_and_call(Term)` which behaves exactly like the builtin `call(Term)`, except that it partially evaluates the to-be called predicate first and then calls the residual code. This includes a caching mechanism so that only one residual predicate per normal predicate exists. This means that if `compile_and_call(Term)` is used twice with the same predicate, compilation is only performed once.

When using `compile_and_call(Term)` to call a predicate, the arguments of the predicate are generalized to `varbox`. This means that the partial evaluator does not know anything about the predicate's arguments. If the partial evaluator needs to know anything, this information has to be re-gained using promotion (Sect. 4).

3.2 The Control Strategy

The term *control strategy* describes in which order the program that is partially evaluated is looked at. The control strategy for our prototype is very simple. It directly follows that of a Prolog interpreter, i.e., partially evaluates from left to write, unfolding all calls to non-builtins. If a call to a builtin is encountered, a single answer is produced, together with a bit of specialized residual code. If the builtin is non-deterministic, the answer still needs to be general enough to cover all cases. If failure is detected by the partial evaluator, the current branch can be pruned.

A simple example to demonstrate how this works. Assume we have a Prolog database looking like this:

```
p(X) :- q(a, X).
q(X, Y) :- X = a, r(Y, b).
r(X, Y) :- X = Y.
```

And we specialize the call `p(X)` (e.g., by calling `compile_and_call(p(b)))`. Then specialization proceeds as follows (the goal that will be handled next is underlined):

Step	Current Goal	Residual Code
1.	$\underline{p(X)}$	$true$
2.	$\underline{q(a, X)}$	$true$
3.	$\underline{a = a}, r(X, b)$	$true$
4.	$\underline{r(X, b)}$	$true$
5.	$\underline{X = b}$	$true$
6.	$true$	$X = b$

The above example was simple in that it did not contain any choice points, i.e., predicates with more than one clause. How those are handled is described in Section 4.

3.3 Producing Residual Code

This section describes how the residual code is produced and what structure it has. The residual code is split up into a set of clauses, that call each other. At any time, the specializer is building the body of one such clause by prepending calls that should go into the body to `ResidualCode` argument of the `interpret` functor and passing on the larger list. When the predicate that is currently being built up is done, its body is constructed by reversing the `ResidualCode` code list and joining the calls in there with a conjunction. Then the predicate is asserted using `asserta` so that the new predicate is tried before any older code. Then a new predicate is started, by using an empty list for `ResidualCode` and putting the head of the new predicate into `ResidualCodeHead`.

All of the generated clause bodies consist only of calls to builtins (and to some special predicates), except for the last call of the body, which is usually a tail call to another residual predicate (if not it is either a call to `true` or to `fail`). Therefore the residual predicates behave like basic blocks since they are pieces of linear code ending in a tail call (which is essentially a jump).

When a new predicate is started, there is a need to have as many arguments at the head of this predicate as there are distinct `varboxes` in the continuation. Since there is nothing known about a `varbox` at specialization time, there needs to be an argument to the residual predicate that will receive the corresponding value at runtime. Contrarily, a `termbox` does not need any variable to represent it at runtime, since its functor and arguments are known anyway. Same for a `freebox`, which is known to be free and not sharing with any other variable.

Figure 5: Turning a Term in Ground Representation into a Prolog Term

```
get_runtime_term(varbox(RtVar), RtVar, Cont, Cont).
get_runtime_term(freebox(RtVar), RtVar, ContIn, ContOut) :-
    replace_box(freebox(RtVar), varbox(RtVar), ContIn, ContOut).
get_runtime_term(termbox(Functor, Args, Id), RtTerm, ContIn, ContOut)
    :-
    get_runtime_term_list(Args, RtArgs, ContIn, ContOut),
    RtTerm =.. [Functor | RtArgs].

get_runtime_term_list([], [], Cont, Cont).
get_runtime_term_list([Box|Boxes], [RtTerm|RtTerms], ContIn, ContOut)
    :-
    get_runtime_term(Box, RtTerm, ContIn, Cont),
    get_runtime_term_list(Boxes, RtTerms, Cont, ContOut).
```

When the specializer tries to add a new call to the predicate body it is currently building, it usually knows the arguments in their ground representation only. The body consists of calls in non-ground representation, i.e., just normal Prolog terms. A `termbox` can just directly be turned into a Prolog term. A `varbox(RtVar)` carries the variable `RtVar` that represents it in the residual code inside it.

The complicated case is the `freebox`. About a `freebox` the specializer knows that it is free. However, once it appears in the residual code in a call (to a builtin or to something else), it is possible that it is bound by this call, so the specializer can no longer assume its freeness. Therefore requesting a term to represent the `freebox` at runtime turns it automatically into a `varbox`. This is achieved by replacing the `freebox` by the new `varbox` everywhere in the `Continuation`. Figure 5 shows the predicate that turns terms in ground representation into normal Prolog terms.

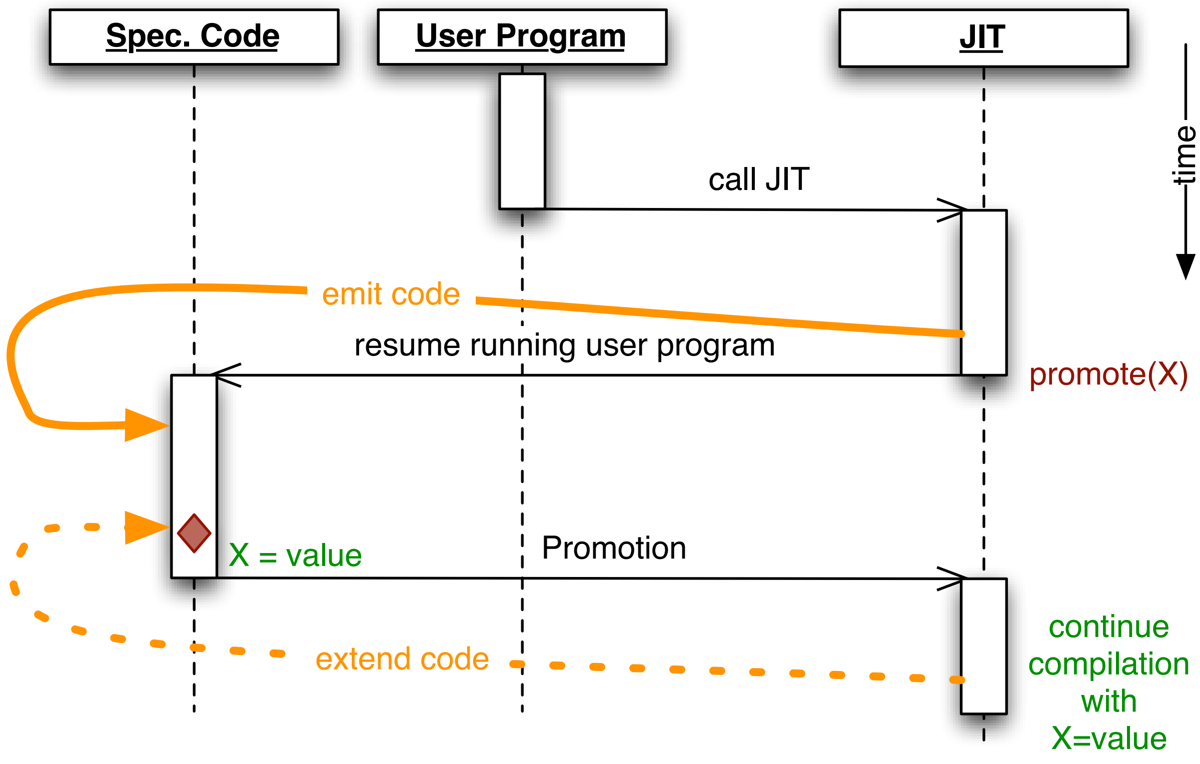
4 Promotion: Lazy Choice Points

4.1 Basic Scheme

One of the fundamental building blocks for the partial evaluator to make use of the dynamic 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 explosion. Therefore it inserts a stub into the residual code and stops the partial evaluation to let the residual code run. When the stub is reached, the compiler is invoked again and compiles exactly the switch case that is needed by the running code. After compilation has finished, this new code is generated. See Figure 6 for a diagram of the interactions involved.

This behaviour can be interpreted differently. Under this interpretation, lazy choice points are used by the partial evaluator to get information about a so far unknown term. When the actual runtime value (or some partial info about the value, like the functor and arity) of an unknown term (i.e., a `varbox`) is needed by the partial evaluator during com-

Figure 6: Interaction Between Partial Evaluation and Code Execution



pilation, promotion can be used to obtain it. If this is happening, compilation needs to stop, because the value is not actually available yet. Then the residual code generated so far is executed until the promotion 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 compiler is invoked with the now known term and more code can be produced.

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

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

For now we rewrite it in the following style, 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 `X`'s functor will be `true` or `false`. Therefore the specialization process stops, and code looking as follows is generated:

```
'$negation1'(X, Y) :-
    '$promotion1'(X, Y).
'$promotion1'(X, Y) :-
    functor(X, F, N),
    continue_compilation(F/N, '$promotion1', ...),
    '$promotion1'(X, Y).
```

The predicate `'$negation1'` is the entry-point of the specialized version of `negation`. The `'$promotion1'` predicate is the lazy choice point. At this point this predicate has only one clause, which is for invoking the compiler again. More clauses will be added later. If it is executed, partial evaluation will be resumed by calling `continue_compilation`, passing in the functor and the arity of the argument as information for compiling 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 `continue_compilation` 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 `continue_compilation` is called, it will use its first argument to decide which of the cases it should partially evaluate further.

Let us assume that `'$negation1'` is called with `false(X)` as an argument. Then `'$promotion1'` will be executed, calling `continue_compilation(false/1, '$promotion1", ...)`. This will generate residual code only for the case where `X` is of the form `false(_)`, which 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 callback to compilation 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 compilation will be performed. The cut is necessary to prevent the backtracking into the compilation case.

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 compiled, saving time and memory. This might not matter for such a trivial case as the one above, but it strongly reduces compilation time and size of the residual code for more realistic cases. If the other case will be compiled 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 compilation case[5].

4.2 Refinements

There are some refinements to the scheme described above. An important one is that the choice point is evaluated directly without lazy stopping, if enough information is available, i.e., if the argument is already a `termbox`. A simple (if artifical) example would be:

[5]In fact, one optimization that is performed is that the general case is retracted as soon as all the cases of a switch have been compiled.

```
f(X, Y) :- negation(false(X), Y).
```

If now `f(X, Y)` is specialized, the `switch_functor` in `negation` will be called with `termbox(false, ...)` as an argument. Therefore the partial evaluation process can immediately decide which case of the switch to chose. The residual code will look like this:

```
'$f1'(X, Y) :- Y = true(X).
```

Another important refinement is supporting unbound arguments to the `switch_functor`. As described so far, the choice points change the Prolog semantics if the switched-upon variable is unbound at runtime: In this case solutions are potentially generated in the wrong order, also the compiler-callback case relies on the first argument being bound. Therefore we must ensure that this never happens, which can be done by putting a helper predicate into the residual code that generates the possible bindings in the right order. For the original example the residual code would thus look like this:

```
'$negation1'(X, Y) :-
    '$case1'(X),
    '$promotion1'(X, Y).
'$case1(true(_)).
'$case1(false(_)).
'$promotion1'(X, Y) :-
    functor(X, F, N),
    continue_compilation(F/N, '$promotion1', ...),
    '$promotion1'(X, Y).
```

What changed is the introduction of the `'$case1'` predicate which ensures that the first argument of `'$promotion1'` is always bound and which generates solutions in the right order of the argument which was unbound before.

So far one of these case predicates is always inserted, in the future a boundness-analysis could be done to decide when they are not needed.

4.3 Other uses of lazy switches

The `switch_functor` primitive has some other uses apart from the obvious ones that it was designed for, namely for lazy choice points. 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). A disjunction `A;B` can be implemented by switching on a new variable like this:

```
switch_functor(_, [case(a, A), case(b, B)])
```

This will have the effect of compiling the two cases of the disjunction lazily, i.e., only when both are actually needed.

A related use of `switch_functor` is the “lazy stop”. It can be used to stop compilation at any point in time and only continue compiling once execution reaches that branch. To achieve this, instead of continuing to compile using some continuation `Cont` we use the following as a continuation:

```
switch_functor(_, [case(a, Cont)])
```

Inserting such a lazy stop can be done at arbitrary points in the partial evaluation process without making the residual code significantly worse. Of course each lazy stop has a small performance cost, but the basic structure of the residual code will not change.

5 Full Prolog Support

5.1 Preprocessing Step

Since the partial evaluator itself cannot deal with choice points at all apart from in the form of `switch_functors`, there needs to be a pre-processing step that turns normal Prolog code with potentially more than one clause per predicate into a form where all these clauses are joined by suitable calls to `switch_functor`.

A correct implementation would be to just use the disjunctions as described in Section 4.3. However, this would not expose functor-switches that are implicit in the various clause to the partial evaluator. What happens instead is that the pre-processor tries to find an argument that is deconstructed by several clauses. Then a `switch_functor` can be used for that argument. Let us look at an example, the typical `append` predicate:

```
append([], T, T).
append([H | L1], L2, [H | L3]) :- append(L1, L2, L3).
```

The predicate deconstructs its first argument (and also its third). Therefore, it should be turned into code looking like this:

```
append(A, B, C) :-
    switch_functor(A, [
        case([]/0, B = C),
        case(./2, A = [H | L1], C = [H | L3], append(L1, B, L3))]).
```

Another example would be the following polymorphic addition:

```
add(int(X), int(Y), int(Z)) :- Z is X + Y.
add(str(X), str(Y), str(Z)) :- atom_concat(X, Y, Z).
add(list(X), list(Y), list(Z)) :- append(X, Y, Z).
```

Which is turned into this:

```
add(A, B, C) :-
    switch_functor(A, [
        case(int/1, (A = int(X), B = int(Y), C = int(Z), Z is X + Y)),
        case(str/1, (A = str(X), B = str(Y), C = str(Z), atom_concat(X, Y, Z))),
        case(list/1, (A = list(X), B = list(Y), C = list(Z), append(X, Y,Z)))]).
```

The preprocessing step is dependent on the context where the call happened. If this `add` predicate would be called in the following context:

```
is_suffix(A, B) :- add(_, A, B).
```

Then the preprocessor would produce the following for `add`:

```
add(A, B, C) :-
    switch_functor(B, [
        case(int/1, (A = int(X), B = int(Y), C = int(Z), Z is X + Y)),
        case(str/1, (A = str(X), B = str(Y), C = str(Z), atom_concat(X, Y, Z))),
        case(list/1, (A = list(X), B = list(Y), C = list(Z), append(X, Y,Z)))]).
```

because the first argument of `add` is known to be free, switching on it would not make as much sense.

The current implementation of the preprocessing step is rather simple and therefore won't be described in all detail here. It should be made more advanced in the future (potentially by the same techniques that are used for unification factoring, a related topic, see [DRSS96]).

5.2 Builtins

In this section we describe the handling of builtins by the partial evaluation system. When the partial evaluator encounters the call to a builtin it needs to produce a residual version of the call, taking the available information about the arguments of the builtin into account. In addition it must sometimes change its knowledge about the arguments of the builtins, particularly about `freeboxes`, which can in the general case no longer assumed to be free after a call to a builtin.

There is a general strategy for doing this, that is correct for almost all builtins. The idea is to just put the call to the builtin into the residual code unchanged and change all `freeboxes` in the arguments into `varboxes`. This means that the partial evaluator essentially ignores builtins and just puts them into the residual code. In addition it assumes that the builtin can bind any of its known-to-be-free arguments (the `termboxes` cannot be changed anyway and the `varboxes` are already assumed to contain anything, therefore both do not need to change). This is achieved by using `get_runtime_term` (see Figure 5) on all the arguments of the builtin call.

For builtins that have side-effects (like I/O builtins) this is essentially the only strategy that is correct. Even if the partial evaluator fully knows the argument to a predicate like `print`, it still needs to put it into the residual code and not just execute it once at compile time.

For builtins that do not have side-effects this does not apply. If enough information is available, the partial evaluator can fully evaluate the builtin at compile time and thus produce no residual code at all. An example of this would be the builtin call `X is 1 + 2`[6]. If the partial evaluator knows that `X` is free at this point, it can just bind `X` to `3` and not produce any residual code. This binding is done by replacing the `freebox` representing `X` by a `termbox(3, [], ...)` in the continuation.

Sometimes emitting no residual code for a builtin at all is not possible, but just emitting the full builtin is also not necessary. In this case the partial evaluator can produce any other bit of residual code that preserves semantics. A simple example is again `X is 1 + 2` where `X` is a `varbox` (and thus possibly but not necessarily free). This means the partial evaluator cannot just bind `X` to `3` at compile-time, since at runtime `X` might be something that does not unify with `3`. Instead it must emit `X = 3` into the residual code, which should still be more efficient than actually having to perform the arithmetic operation.

In fact, the most typical occurrence of this sort of behaviour occurs when the unification builtin = is itself found in the source program. Let us assume the following unification needs to be specialized: $f(\bar{A}, b) = f(c, B)$. Then the residual code would be `B = b`. Note that the binding of $\bar{A}$ needs no residual code, since the specializer knows it is free and can thus directly bind it by replacing all its occurrences with c. This reasoning does not apply to B, which might already be bound at runtime at this point. This sort of residualization of a builtin can be seen to be a form of "strength-reduction". The partial evaluator performs as many parts of the builtin as it can at compile-time, what it cannot perform must be done by the residual code that is emitted.

The strategies described above do not work at all for the cut and if-then-else (`A -> B ; C`). The problem with those is that they are position-dependant, which means that they depend on where in a predicate they occur and on how many clauses a predicate has. This is a problem for the partial evaluator. It cannot just produce cuts in the residual code, since the clauses in the residual code do not at all correspond to clauses in the original code. Therefore the cuts and if-then-else calls are transformed away into a form where they are implemented using exceptions. The transformation that is used is essentially that described in [Pre92]. The benefit of using exceptions is that they are position-independent. See Figure 7 for an example.

The `';'` builtin (used for disjunction) is handled by replacing it with a `switch_functor`, as described in section 4.3.

Builtins that are particularly interesting in the context of dynamic partial evaluation are meta-call builtins like `call` or `findall`. Most classical partial evaluators cannot handle such builtins when they do not manage to infer what the functor of the called predicate is. If the functor cannot statically be inferred the partial evaluator needs to give up often by putting the original meta-call into the residual code. This prevents any partial evaluation

[6]Note that the programmer did not necessarily write this slightly nonsensical bit of code. The actual code could for example be `X is 1 + A` but the partial evaluator found out that `A` is always `2` here.

Figure 7: Transforming the Cut into Exceptions

Original code:

```
f(0, []) :- !.
f(X, [_ | T]) :- X0 is X - 1, f(X0, T).
```

Transformed code that is no longer using the cut:

```
f(X, Y) :- catch(f_new(X, Y, CutPoint),
               cuthere(CutPoint2),
               rethrow_cuthere(CutPoint, CutPoint2)).
f_new(0, [], CutPoint) :- cuthere(CutPoint).
f_new(X, [_ | T], _) :- X0 is X - 1, f(X0, T).

cuthere(CutPoint).
cuthere(CutPoint) :- throw(cuthere(CutPoint)).
rethrow_cuthere(CutPoint, CutPoint2) :-
    CutPoint \== CutPoint2, throw(cuthere(CutPoint2)).
```

from happening. Our prototype can easily handle such builtins. If the call happens on a `termbox`, the partial evaluator knows which predicate to call and can just proceed. Otherwise, it can use the same mechanism that `switch_functor` uses to observe the actual program and thus get to know the functor of the called term.

An extreme example for a case where a classical partial evaluator does not work very well is a read-eval-print-loop (repl) where each goal that the user wants to evaluate should be partially evaluated. Figure 8 shows an extremely simplified repl together with an example session and the residual code that our partial evaluator produced after the example session. A classical partial evaluator can obviously have no knowledge in advance about what goals the user will type in, so would not perform any interesting partial evaluation.

6 Merging

The specialization algorithm described so far produces code that has a tree as its call-graph. See Figure 9 for an example of how a call graph could look like. This means that all loops in the specialized program will be fully unfolded, leading to arbitrarily much generated code. If the program contains some sort of loop, it will be fully unfolded as well. Clearly this is not desirable so a mechanism is needed to reuse already compiled code. This mechanism is called merging. Merging is triggered by a special predicate called `jit_merge_point` that the user of the specializer needs to put at certain points into his program. The fact that the user needs to trigger merging herself is clearly not optimal and we plan to examine ways of finding good places where to do merging automatically in the future. However, it is also not too burdensome, typically the number of `jit_merge_points` that need to be put into an interpreter is small (mostly the number is even one).

If a call to this predicate is encountered during compilation, the specializer stops the

Figure 8: A Simple Repl for Prolog

Repl code and some example predicates:

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

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

Example session:

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

No
```

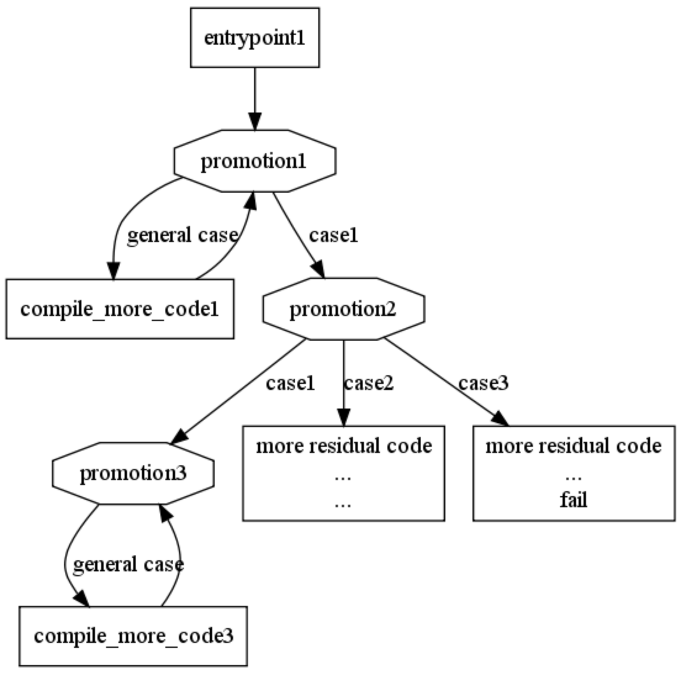
Produced residual code (promotion compilation cases not shown):

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

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

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

Figure 9: Example Call Graph Without Merging



specialization process. It then compares its current state against the states it had at earlier merge points. If none of the earlier states are similar enough to the current one or if this is the first merge point that is encountered, a snapshot of the current state of the specializer is taken and stored in the Prolog database for the benefit of future merging attempts. Then a new predicate is started and associated with that snapshot.

If the snapshotted state of an earlier merge point matches the current state, then we know that the old residual code can be reused. Therefore the specializer inserts a call to the predicate that is associated with that old state. Afterwards, specialization can stop, and the residual code that is currently being built can be asserted. Since the inserted call is the last thing that went into the residual code, it is a tail call.

The "state" we have talked about so far is the `Continuation` argument of the `interpret` main predicate. This argument contains all the calls that are still left to specialize and therefore contains enough information to decide whether an older `Continuation` was similar enough to reuse the code that was generated for it.

Whether two states match or not is a complicated question, and will be the subject of the following subsections. The exact algorithm that is used will be described in a piecemeal fashion, first giving the simplest condition that is needed for a successful merge and then refining it step by step.

6.1 Sound Merges

First some terminology: When a merge is attempted, the *current state* (which is a Prolog term in ground representation) is compared against a *target state* (also a Prolog term). Every target state has a predicate attached called the *target predicate*, which contains the residual code the specializer produced when specializing the target state, called the *target code*. The residual code that the specializer would produce if it continued specializing with the current state is called the *potential current code*. When the two states are considered similar enough the merge is called *successful*.

A necessary and thus the most important condition that a merge must fulfill to be successful is the condition of *soundness*. A merge is called *sound*, if behaviour is conserved, i.e., if the residual code attached to the target state behaves the same way as the code the specializer would generate for the current state. The code does not need to be exactly identical, it is enough if the behaviour is the same.

For this condition to be fulfilled, the target state needs to be equal to or more general than the current state. In other words, the current state, as a Prolog term, must be an instance of the target state. This mean that all parts of the target state that are `termboxes` should be `termboxes` in the current state as well and also have the same functor and the same number of arguments. The `varboxes` in the target state can however be replaced by either `freeboxes` or `termboxes` in the current state.

The reasoning behind this is that if the target state is strictly more general than the current state, the target code is also more general than the potential current code would be. The target code was generated under the assumption that the `varboxes` can be anything, so it is legitimate to pass anything there when doing a merge.

The places where the current state is more instantiated than the target state are where a successful merge loses information. If the target code was generated without any knowledge about a certain term (i.e., a `varbox`) and the current state contains more information about this term (i.e., it is a `freebox` or a `termbox`) then this information is not used, when the merge succeeds. Merges that do not lose information can always succeed. Those that are sound but lose information need careful treatment, see next section.

Let us look at a simple example: We specialize the standard `append` predicate. This does not actually lead to any speed-improvement, but shows the fundamental behaviour of merges. The predicate and the residual code can be seen in Figure 10. When the specializer hits the merge point for the first time, it needs to create a new predicate. When the merge point it hit the second time, the state of the partial evaluator is the same as when the merge point was hit the second time, thus the merge is successful (and does not lose any information).

6.2 Efficient Merges

When a merge loses information, disastrous effects can ensue. It is possible, that the information that is lost by the merge was actually needed and the target code contains a promotion to gain this information. If this is the case, the residual code contains a clear inefficiency, since promotions are not cheap (they to perform a match on the functor of

Figure 10: Merging Example: A Sound Merge

```
1  % original code:
2
3  append([], X, X).
4  append([H | T1], T2, [H | T3]) :-
5          jit_merge_point, append(T1, T2, T3).
6
7  % ----------------------------------------------------------
8  % residual code for append([a, b, c], [d], X):
9
10
11 '$entrypoint1'(A, B, C) :-
12         '$case1'(A), '$promotion1'(A, C, B).
13
14 '$case1'([]). '$case1'([_|_]).
15
16 '$promotion1'([D|E], C, B) :- !,
17         C=[D|F],
18         '$mergepoint1'(F, B, E).
19
20 '$mergepoint1'(F, B, E) :-
21         '$case1'(E), '$promotion2'(E, F, B).
22
23 '$promotion2'([G|H], F, B) :- !,
24         F=[G|I],
25         '$mergepoint1'(I, B, H).
26 '$promotion2'([], F, B) :- !,
27         B=F.
```

The specializers state at line 18 was $append(F, B, E)$. At line 25 it was the $append(I, B, H)$.

a term). This situation is especially bad in the case where the successful merge closes a loop in the residual code. For every iteration a useless promotion is executed.

This leads to the definition of an *inefficient merge*: An inefficient merge is one that closes a loop in such a way that the merge loses information that the loop body reacquires with a promotion.

Let us look at an example. In figure 11 there is a simple predicate, which takes a list and adds the atom `a` either once or twice in front of each element of the list. The predicate checks its `Howmany` argument again for every list element, even though `Howmany` does not change throughout the loop over the list. Ideally, we would like the specializer to recognize this, and specialize the predicate in such a way that the check only occurs once and is followed by a loop specialized for only one of these cases.

What happens when just soundness is used for merging can be seen in the residual code in figure 11. The merge in predicate `$promotion3` is successful, even though the information that `Howmany = 1` is lost in the merge. The code following `'$mergepoint1'` promotes `Howmany`, so the merge was inefficient. As it is, the residual code is essentially equivalent to the original predicate which is not the desired outcome.

At the time where the merge is attempted, enough information is available to see that having this merge succeed will lead to an inefficient loop. The merge loses information in the third argument to `$mergepoint1`, and that third argument of the predicate is soon promoted.

Since it is not really practical to inspect the generated and asserted residual code, the specializer keeps a condensed version of the residual code that it has generated so far that can be used to decide whether a merge is efficient or not. This is called the `History` (see Figure 12). The history records all merges that failed, all promotions (whether the promoted term was known already or not) and unifications of two variables. The history is a linear list of such events, with the event that happened most recently at the beginning of the list.

When a merge is attempted this information can then be used to decide as to whether the merge is efficient or not. However, this is only possible when the target merge point is actually within the history. This is not always the case, for example the target merge point could lie in another branch of an older promotion. To still allow the detection of inefficient merges, we restrict the merge points that are tried as targets to those merge points that are in the current history.

A successful merge with a target merge point in the current history has the interesting property that it always produces a loop in the call graph (which corresponds to a normal loop due to tail call optimization). Since loops are the parts of a program where most execution time is spent it is beneficial trying to concentrate on loops when deciding whether a particular bit of code is efficient or not.

For the inefficient example above the history would look as in Figure 12. Looking at the history of `promotion3` it can be seen that first and the third argument of `mergepoint1` will be promoted after the merge point. Therefore the merge that succeeded in Figure 11 should not succeed. What should happen instead can be seen in Figure 13: A specialized version of the loop is generated that works only for `Howmany = 1`.

Figure 11: Merging Example: An Inefficient Merge

```
1  % original code:
2
3  add_one_or_two_as([], _, []).
4  add_one_or_two_as([H | T], Howmany, [H | R]) :-
5      jit_merge_point, add_as(Howmany, R, T2),
6      add_one_or_two_as(T, Howmany, T2).
7
8  add_as(1, [a | T], T).
9  add_as(2, [a, a | T], T).
10
11 % ------------------------------------------------------------
12 % residual code for add_one_or_two_as([a, b, c, d], 1, X):
13
14 '$entrypoint1'(A, B, C) :-
15         '$case1'(A), '$promotion1'(A, C, B).
16
17 '$case1'([]). '$case1'([_|_]).
18
19 '$promotion1'([D|E], C, B) :- !,
20         C=[D|F],
21         '$mergepoint1'(E, F, B).
22
23 '$case2'(1). '$case2'(2).
24
25 '$mergepoint1'(E, F, B) :-
26         '$case2'(B), '$promotion2'(B, E, F).
27
28 '$promotion2'(1, E, F) :- !,
29         F=[a|G],
30         '$case1'(E), '$promotion3'(E, G).
31
32 '$promotion3'([H|I], G) :- !,
33         G=[H|J],
34         '$mergepoint1'(I, J, 1).
35 '$promotion3'([], G) :- !,
36         G=[].
```

The specializers state at line 21 is $add\_as(B, F, \bar{X}), add\_one\_or\_two\_as(E, B, \bar{X})$. At line 34 it is $add\_as(1, J, \bar{X}), add\_one\_or\_two\_as(I, 1, \bar{X})$

Figure 12: History Example

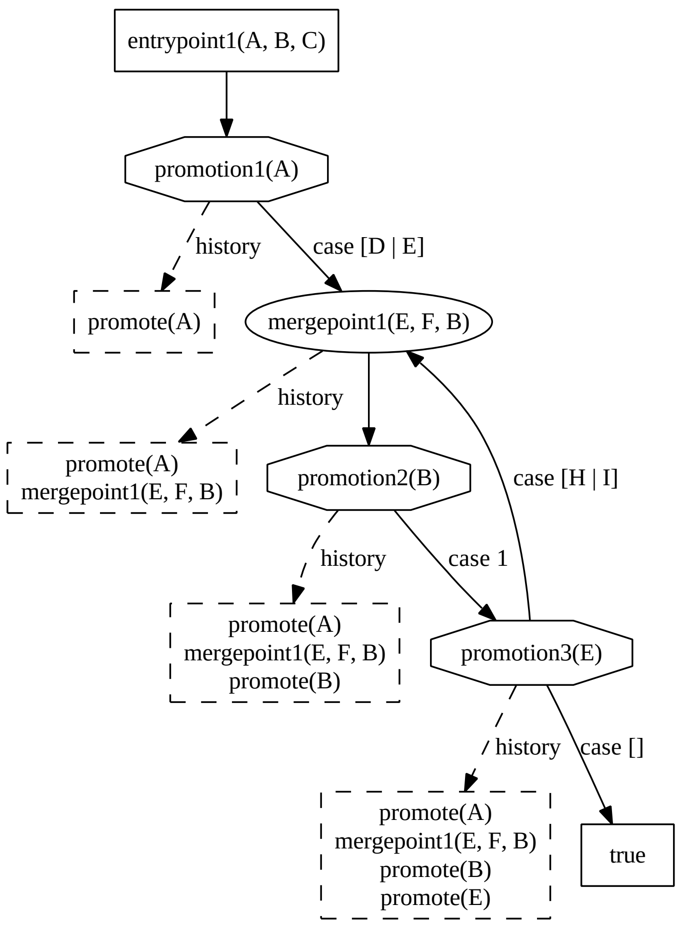


Figure 13: Merging Example: Preventing the Inefficient Merge

```
 1 % original code:
 2
 3 add_one_or_two_as([], _, []).
 4 add_one_or_two_as([H | T], Howmany, [H | R]) :-
 5     jit_merge_point, add_as(Howmany, R, T2),
 6     add_one_or_two_as(T, Howmany, T2).
 7
 8 add_as(1, [a | T], T).
 9 add_as(2, [a, a | T], T).
10
11 % ————————————————————————————————————————————
12 % residual code for add_one_or_two_as([a, b, c, d], 1, X):
13
14 '$entrypoint1'(A, B, C) :-
15         '$case1'(A), '$promotion1'(A, C, B).
16
17 '$case1'([]). '$case1'([_|_]).
18
19 '$promotion1'([D|E], C, B) :- !,
20         C=[D|F],
21         '$mergepoint1'(E, F, B).
22
23 '$case2'(1). '$case2'(2).
24
25 '$mergepoint1'(E, F, B) :-
26         '$case2'(B), '$promotion2'(B, E, F).
27
28 '$promotion2'(1, E, F) :- !,
29         F=[a|G],
30         '$case1'(E), '$promotion3'(E, G).
31
32 '$promotion3'([H|I], G) :- !,
33         G=[H|J],
34         '$mergepoint2'(I, J).
35
36 '$mergepoint2'(I, [a|K]) :-
37         '$case1'(I), '$promotion4'(I, K).
38
39 '$promotion4'([L|M], K) :- !,
40         K=[L|N],
41         '$mergepoint2'(M, N).
42 '$promotion4'([], K) :- !,
43         K=[].
```

The specializers state at line 21 is $\mathit{add\_as}(B, F, \bar{X}), \mathit{add\_one\_or\_two\_as}(E, B, \bar{X})$. At line 34 it is $\mathit{add\_as}(1, J, \bar{X}), \mathit{add\_one\_or\_two\_as}(I, 1, \bar{X})$. At line 41 it is $\mathit{add\_as}(1, N, \bar{X}), \mathit{add\_one\_or\_two\_as}(M, 1, \bar{X})$

6.3 Generalizing the Continuation

The merging algorithm described so far – trying to find sound efficient merges – has cases where a merge never happens. This occurs when the continuation grows from merge attempt to merge attempt, but the most recent continuation is never an instance of any of the older ones.

An example of this is when the specialized program builds up a result. In Figure 14 we specialize a predicate that reverses a list using an accumulator. The partial evaluator knows the accumulator, which grows and grows. Therefore a merge is never possible. That is bad, since knowing the accumulator is not actually helpful for the partial evaluation.

To prevent this effect, we need to introduce a new step in the merging algorithm. When all merge attempts have failed, the current continuation is widened. To do this, we look at the merge points in the history, starting from the most recent ones. We are looking for an older state that is different from the current state. The interesting differences are those where a `termbox` in the current state differs from a `termbox` in the older state (either because the functor or because the arity is different). We would like to continue with a new state where the places where the states differ are replaced by a `varbox`.

This is achieved by an operation called the *most specific generalization* [LMM88] of two Prolog terms (also called *least general generalization* or *anti-unification*. Given two Prolog terms A and B a generalization of these terms is a term C so that A and B are instances of C. The most specific generalization is a generalization M of A and B so that for all generalizations C of A and B, M is an instance of C.

When we found an older state that has differences to the current state, we widen the current state so that the new state is the most specific generalization of the current state and the older state (we do not just pick any random older state but only let widening be performed under a condition that will be described further down). This will replace the `termboxes` that differ between the two states by `varboxes`. Then we continue partial evaluation with the new state as the continuation.

The reasoning behind this is that we generalize away those and only those parts of the state that prevented the merge to succeed this time. Therefore it is likely that the merge will succeed when the next merge point is hit.

In Figure 15 we see how widening solves the problem of Figure 14. Since the continuation is growing, the merge from `'$promotion2'` to `'$mergepoint2'` is prevented. Instead, the continuation $revacc(F, [E, C], B)$ is widened, by computing the most specific generalization between it and the older state $revacc(D, [C], B)$. The result is the new continuation $revacc(F, [E|G], B)$. The next time a merge is attempted, it succeeds.

There is one small addition to the widening scheme described so far. When computing the most specific generalization it is possible that a `termbox` that was used to decide which case of a lazy switch should be used is replaced by a `varbox`. Thus we would lose information that was actually used successfully by the partial evaluator when producing the residual code of the older state. To prevent this from happening we can again use the information in the history to prevent such a widening.

Figure 14: Merging Example: Merge Never Occurs

```
 1 revacc([], R, R).
 2 revacc([H | T1], T2, R) :-
 3     jit_merge_point, revacc(T1, [H | T2], R).
 4
 5 rev(E, R) :-
 6     revacc(E, [], R).
 7
 8 % -----------------------------------------------------
 9 % residual code for rev([a, b, c, d], X):
10
11 '$entrypoint1'(A, B) :-
12         '$case1'(A), '$promotion1'(A, B).
13
14 '$case1'([]). '$case1'([_|_]).
15
16 '$promotion1'([C|D], B) :- !,
17         '$mergepoint1'(B, C, D).
18
19 '$mergepoint1'(B, C, D) :-
20         '$case1'(D), '$promotion2'(D, B, C).
21
22 '$promotion2'([E|F], B, C) :- !,
23         '$mergepoint2'(B, C, E, F).
24
25 '$mergepoint2'(B, C, E, F) :-
26         '$case1'(F), '$promotion3'(F, B, C, E).
27
28 '$promotion3'([G|H], B, C, E) :- !,
29         '$mergepoint3'(B, C, E, G, H).
30
31 '$mergepoint3'(B, C, E, G, H) :-
32         '$case1'(H), '$promotion4'(H, B, C, E, G).
33
34 '$promotion4'([I|J], B, C, E, G) :- !,
35         '$mergepoint4'(B, C, E, G, I, J).
36
37 '$mergepoint4'(B, C, E, G, I, J) :-
38         ...
39 ...
```

The specializers state at line 17 is $revacc(D, [C], B)$. At line 23 it is $revacc(F, [E, C], B)$. At line 29 it is $revacc(H, [G, E, C], B)$. At line 35 it is $revacc(J, [I, G, E, C], B)$, and so on.

Figure 15: Merging Example: Widening the Continuation

```
revacc([], R, R).
revacc([H | T1], T2, R) :-
    jit_merge_point, revacc(T1, [H | T2], R).

rev(E, R) :-
    revacc(E, [], R).

% ----------------------------------------------------------
% residual code for rev([a, b, c, d], X):

revacc([], R, R).
revacc([H | T1], T2, R) :-
    jit_merge_point, revacc(T1, [H | T2], R).
'$entrypoint1'(A, B) :-
        '$case1'(A), '$promotion1'(A, B).

'$case1'([]). '$case1'([_|_]).
'$promotion1'([C|D], B) :- !,
        '$mergepoint1'(B, C, D).

'$mergepoint1'(B, C, D) :-
        '$case1'(D), '$promotion2'(D, B, C).

'$promotion2'([E|F], B, C) :- !,
        '$mergepoint2'(B, [C], E, F).

'$mergepoint2'(B, G, E, F) :-
        '$case1'(F), '$promotion3'(F, B, G, E).

'$promotion3'([H|I], B, G, E) :- !,
        '$mergepoint2'(B, [E|G], H, I).
'$promotion3'([], B, G, E) :- !,
        B=[E|G].
```

$revacc(D, [C], B)$, $revacc(F, [E, C], B)$, $revacc(F, [E|G], B)$, $revacc(I, [H, E|G], B)$,

6.4 The Full Merging Algorithm

In this section we give a summary of the full merging process. The merging process has two inputs, one is the *current state*, which is the continuation of the partial evaluator when it hit the `jit_merge_point`, the other is the *history*. For the purpose of this section, the history is simply a list of target merge points together with their target state. The list is in the order in which the merge points were produced, the most recent merge points being at the front of the list. The target states in the history are the continuations that the partial evaluator had when the older merge points where produced. The target states in the history have some additional information: some of the `termboxes` and `varboxes` in them are marked. A mark on a box means that the corresponding box was promoted on the path between the target state and the current state.

Merging proceeds in four steps:

1. Searching for an information-loosing efficient match. For every target state in the history we do the following:

- If the target state is not more general than the current state:
  - The potential match is not sound. Continue with the next target state in the history.
- Else:
  - Compute the most general unifier of the target state and the current state. This is a list of bindings of the `varboxes` from the target state to `termboxes` or `varboxes`.
  - If these bindings map any marked `varbox` from the target state to a `termbox`:
    - The potential match is not efficient. Continue with the next target state in the history
  - Else:
    - The match is sound and efficient. Insert a call to the corresponding predicate and stop the partial evaluation process.

2. Searching for an exact match. For all target states (not only those in the history) we do the following:

- Check whether the target state is structurally equal to the current state. This is done using the Prolog-builtin `=@=`. If this is the case:
  - We have found a non-information-losing match. Insert a call to the corresponding predicate and stop the partial evaluation process.
- Else:
  - Continue with the next target state.

Using structural equality ensures that no `varbox` in the target state is bound to any `termbox`, thus ensuring that no information is lost.

3. Trying to widen the current continuations. At this point no existing target state that would match has been found. We try to widen the current continuation before going on. For this we do the following for every target state in the history:

- Compute a potential *new state*, which is the most specific generalization of the target state and the current state.
- Compute the most general unifier of the new state and the target state. This is a list of bindings of the `varboxes` from the new state to `termboxes` or `varboxes` from the target state.
- Check if any `varbox` is bound to a marked `termbox` from the target state. If that is the case:
  - The widening would lose information that is likely to be needed again. Therefore, we continue trying to widen with the next state in the history.
- Else:
  - Use the new state as the continuation and continue the partial evaluation process.

4. Default behaviour. Neither did any of the potential merges succeed, nor did we manage to widen the continuation. Thus we just continue the partial evaluation process with the current state as the continuation.

7 Partial Evaluation Algorithm

This section gives a succinct summary of the various elements of the partial evaluation algorithm.

Partial evaluation is started by the user calling `compile_and_call(Goal)`. Then:

- If the user did not yet call `compile_and_call` with a goal that had the same functor and arity than the current goal, a new residual predicate is started and associated with the current functor and arity. All the arguments of `Goal` are replaced by `varboxes`. Then partial evaluation starts with that goal as the continuation. When partial evaluation stops, the new residual predicate is called.
- If the user already called `compile_and_call` with a goal that had the same functor and arity than the current goal, call the residual code associated with that functor and arity.

In the first case, partial evaluation is performed, until the process is stopped or until the continuation is empty. The continuation consists of goals, joined together with a conjunction. Partial evaluation always treats the leftmost goal of the continuation next. When doing this, one of the following cases applies:

1. The leftmost goal of the continuation is a `switch_functor` that is sufficiently instantiated. Then the corresponding case will be chosen and partial evaluation will continue there.

2. The leftmost goal of the continuation is a `switch_functor` that is not sufficiently initiated. Then a lazy switch will be inserted into the residual code with a general case that calls back into the compiler. Then the partial evaluation process will stop.

3. The leftmost goal of the continuation is a builtin and we can statically detect that it fails. Then the branch which is currently being specialized is pruned and partial evaluation stops.

4. The leftmost goal of the continuation is a builtin which cannot be statically detected to fail. Then it is residualized, which yields a single computed answer general enough for all cases, and a specialized version of that builtin.

5. The leftmost goal of the continuation is a call to a user-predicate. Then the preprocessing step is applied to the various clauses of the predicate and the result is inserted into the continuation.

6. The leftmost goal of the continuation is a `jit_merge_point`. In this case merging is attempted, as described in section 6.4. When a successful merge is found, partial evaluation stop. Otherwise it continues.

When the continuation is empty or the partial evaluation process is stopped by case 2., 3. or 6., the residual code that was built so far is asserted using `asserta`.

7.1 Ensuring Termination

Since a classical partial evaluator is run in advance, it should always terminate, even though the user program itself does not. There are various techniques for ensuring this [LB02]. The situation is different for dynamic partial evaluation since it happens at runtime. There it is enough to make sure that running the partial evaluator does not change the termination behaviour of the program, i.e., the partial evaluator is only allowed to not terminate, when the original program does not terminate either. For the rest of the section let us assume that the original program terminates.

In this work, partial evaluation and execution of the residual code is interleaved. To ensure termination it is therefore enough to make sure that the residual code is executed from time to time, because eventually the residual code will terminate. To ensure that the residual code is executed from time to time we need to prevent the partial evaluator running for arbitrarily long periods. The partial evaluation process can be stopped at any point without losing much performance by inserting a *lazy stop* into the continuation, as described in Section 4.3.

Of the cases above, only case 5. increases the size of the continuation, all other cases reduce it. Therefore we can count the number of times the partial evaluator executes case 5. without running the residual code in the meantime. If that number grows bigger than some number N, we can insert a lazy stop into the continuation to give the residual code a chance to run. In this way, termination is ensured.

We can even reduce at will the overhead of ensuring termination by changing the value of N. (Note however that although increasing the value of N reduces the runtime overhead, there is no guarantee that any given value of N is large enough to keep the runtime overhead below some threshold for all specialized programs. Indeed, in extreme cases, more than N user calls can be unfolded by the specializer without any residual code being generated, leading to several consecutive lazy stops.)

8 Benchmarks and Applications

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 [LMDS98], 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 the "classic mode" which uses normal partial evaluation and in "conjunctive mode" (which uses conjunctive partial deduction with characteristic trees and homeomorphic embedding; see [DSGJ+99]). Conjunctive partial evaluation is considerably more powerful, but also much more complex.

Figure 16 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. While our prototype is rather straightforward and was written from scratch over the course of some months and consists of about 1500 lines of Prolog code, ECCE is a mature system that employs serious theoretical results and consists of 25000 lines of Prolog code.

Some of the speed difference might also be explained by the different ways of using the Prolog VM. Asserted code is typically slower than compiled code, and our chain of tail-call-only generated predicates that mostly move data around does not look like typical Prolog code, so the SWI-Prolog might not really optimized for that.

As we have also seen in Section 2.4 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 underspecialization.

9 Related Work

Promotion is a concept that was already explored in other contexts. Psyco is a run-time specializer for Python that uses promotion (called "unlift" in [Rig04]). Similarly, the PyPy project [RP06, BR07], in which the author is also involved, contains a runtime specialization system built on promotion [RP07].

Greg Sullivan first introduced Dynamic Partial Evaluation [Sul01] and describes an implementation for a small dynamic language based on lambda calculus.

One of the earliest works on runtime specialization is Tempo for C [CN96, CHN+96]. 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 [GMP+00], 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).

On the conceptual level polymorphic inline caches (PIC) [HCU91] 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 source 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 [GPF06] (originating from Dynamo [BDB00]) shares many characteristics of our work. Trace-based JITs also 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 [GF06] on Java, these guards' behaviour is extended to be similar to our promotion. This has been used twice to implement a dynamic language (JavaScript), by Tamarin[7] and in [CBY+07].

At a higher level, our idea of keeping track of which parts of a term are useful for specialization, and generalising away the rest, can be traced back to the notion of characteristic trees in [GB91]. This technique is also employed in the ECCE system which we have used for performance comparisons.

[7] `http://www.mozilla.org/projects/tamarin/`

10 Conclusion and Future Work

We described the concepts and the implementation of a dynamic partial evaluation system for the Prolog language. The prototype we implemented uses the concept of *promotion* to help address some of the problems that trouble classical partial evaluation, namely code explosion, under-specialization, termination problems and support for large parts (particularly extra-logical) of the Prolog language. Despite our prototype's relative simplicity it is quite full-featured, supporting a large range of builtins and giving competitive performance. The builtin support is generally notable, since on the one hand it is very simple to support builtins and on the other hand even builtins that are not explicitly supported still work reasonably due to the generic builtin code.

Due to the use of promotion our 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 as well. In particular promotion needs a Prolog system that supports `assert` well, since the whole approach depends on them 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.

Merging is another area that still needs further work. We plan to explore ways of inserting the `jit_merge_points` automatically. Furthermore, the merging strategy needs further evaluation and possible refinement, as over-specialization is not always prevented.

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.

On a conceptual level we plan to try to incorporate some of the ideas (particularly the approach to merging) of this work into the partial evaluator of the PyPy project.

Figure 16: Experimental Results

	Experiment	Inferences	CPU Time	Speedup
A vanilla meta-interpreter [HG98, MD95] running append with a list of 100000 elements. The interpreter can be seen in Figure 4.	Vanilla - Append			
	original	500008	0.35 s	1.0
	JIT spec+run	281842	0.13 s	2.69
	JIT 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 spec+run	1577228	0.66 s	2.15
	JIT 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 4. Input was a full tree of depth 18.	Vanilla - Replace Leaves			
	original	2621438	2.76 s	1.0
	JIT spec+run	2493636	1.77 s	1.56
	JIT 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 spec+run	5082861	3.53 s	4.74
	JIT 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 ($\sim$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 spec+run	5699992	1.46 s	2.14
	JIT run	200019	0.08 s	39.13
	ecce classic	100003	0.05 s	62.6
	ecce conjunctive	100003	0.04 s	78.25

References

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35:1–12, 2000.

[BR07] Carl Friedrich Bolz and Armin Rigo. How to *not* write a virtual machine. In *Proceedings of 3rd Workshop on Dynamic Languages and Applications (DYLA 2007)*, 2007.

[CBY$^+$07] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael 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.

[CHN$^+$96] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volansche. A uniform approach for compile-time and run-time specialization. In *Dagstuhl Seminar on Partial Evaluation*, pages 54–72, 1996.

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *POPL*, pages 145–156, 1996.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, New York, NY, USA, 1993. ACM Press.

[DCED96] Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: the Standard: Reference Manual*. Springer-Verlag, London, UK, 1996.

[DRSS96] Steven Dawson, C. R. Ramakrishnan, Steven Skiena, and Terrance Swift. Principles and practice of unification factoring. *ACM Trans. Program. Lang. Syst.*, 18(5):528–563, 1996.

[DSGJ$^+$99] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems · Computers · Controls*, 2(5):45–50, 1971. Reprinted in Higher-Order and Symbolic Computation 12(4):381-391, 1999, with a foreword.

[GB91] John Gallagher and Maurice Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[GF06] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, November 2006.

[GMP$^+$00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248:147–199, 2000.

[GPF06] Andreas Gal, Christian W. Probst, and Michael 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.

[HCU91] Urs Hölzle, Craig Chambers, and David 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.

[HG98] Patricia Hill and John 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.

[LB02] Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[LMDS98] Michael Leuschel, Bern Martens, and Danny 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.

[LMM88] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[MD95] 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.

[Pre92] Steven Prestwich. An unfold rule for full Prolog. In Kung-Kiu Lau and Tim Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, pages 199–213, University of Manchester, 1992. Springer-Verlag.

[PyP] PyPy development team. PyPy. An Implementation of Python in Python. http://codespeak.net/pypy.

[Rig] Armin Rigo. Psyco. http://psyco.sourceforge.net.

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[RP06] Armin Rigo and Samuele 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.

[RP07] Armin Rigo and Samuele Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[Sal04] Michael Salib. Starkiller: A static type inferencer and compiler for python, 2004.

[Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 238–256. Springer-Verlag, 2001.

[War83] David Warren. An abstract prolog instruction set. Technical Report Technical Note 309, SRI International Artificial Intelligence Center, October 1983.

List of Figures