

Generating Optimizing Specializers

Robert Glück*

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
e-mail: glueck@diku.dk

Jesper Jørgensen

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
e-mail: knud@diku.dk

Abstract

We propose a new method for improving the specialization of programs by inserting an interpreter between a subject program and a specializer. We formulate three specializer projections which enable us to generate specializers from interpreters. The goal is to provide a new way to control the specialization of programs, and we report the first practical results. This is a step towards the automatic production of specializers.

Using an existing, self-applicable partial evaluator we succeeded in generating a stand-alone specializer for a first-order functional language which is stronger than the partial evaluator used for its generation. The generated specializer corresponds to a simple super-compiler. As an example we show that the generated specializer can achieve the same speed-up effect as the Knuth, Morris & Pratt algorithm by specializing a naïve matcher with respect to a fixed pattern. The generated specializer is also strong enough to handle bounded static variation, a case which partial evaluators usually can not handle.

Keywords: partial evaluation, program generation, automatic programming, self-application, interpreters

1 Introduction

1.1 Program Specialization

Program specialization is a principle of program transformation which reconciles generality with efficiency by providing automated specialization and optimization of programs. The ultimate goal of program specialization is to improve the efficiency of programs

by exploiting known information about the input of a program. Program specialization has proven its usefulness in a number of areas ranging from the specialization of scientific computation to automatic compiler generation.

Partial evaluation is a method for program specialization based on fold/unfold transformations (constant propagation = unfolding). This field has been developing rapidly during the past decade due to advances achieved both in theory and practice [CD93, JGS93].

1.2 Critical Assessment

Despite the successful application of partial evaluation, one often faces the situation that a subject program can not be specialized in a satisfactory way (some known information is not exploited in the way one would expect or hope).

Traditionally, this problem is overcome by rewriting the subject program so that the specializer can produce the desired result [CD89, Bon93]. Such transformations of the subject program are known as *binding-time improvements*. These are semantics-preserving transformations which enable the specializer to propagate more information during specialization. Although some of these transformations have been formalized, in practice they are usually performed in an ad hoc manner. The main disadvantage is that the user has to analyse and modify each subject program individually.

The other alternative is to improve the program specializer. Various methods have been proposed for this purpose (they include polyvariant binding-time analysis, on-line specialization, and modifying the style in which a specializer is written, etc.). The main advantage of this approach is that the specialization of a large class of subject programs can be improved. But modifying a specializer is not a trivial task: it

*Supported by the Austrian Science Foundation (FWF) under grant number J0780.

requires considerable insights into an existing system (or even building a new specializer from scratch). Furthermore, the source code of a specializer might not be available. In conclusion, rewriting a specializer is generally far too complex, time-consuming, or even impossible for the user.

1.3 The Interpretive Approach

In this paper we use a new *interpretive approach* to improve the specialization of programs. The essence of this method is to insert an interpreter between a subject program and a specializer. This approach provides a new opportunity to control the specialization of programs: modifying an interpreter.

The interpretive approach gives control over the specialization of a large class of programs without rewriting the specializer or binding-time improving the subject programs. Using the *specializer projections* one may even generate new specializers from interpreters by self-application.

In this paper we describe how to write such interpreters. We demonstrate that the interpretive approach can be put to work with the existing partial evaluation technology. As a full example, we show how to construct an interpreter for a first-order functional language and how to generate an optimizing specializer that automatically performs a large class of binding-time improvements (e.g. bounded static variation). To the best of our knowledge, these are the first practical results using the interpretive approach with partial evaluators.

Our work can be seen as a step towards the automatic production of specializers: a collection of interpreters can be shared, reused and combined to achieve various specialization effects, so that specializers with different properties can be generated automatically (“bricks instead of tricks”). Specializer generation has another important aspect: the correctness of a generated specializer can be guaranteed by the correctness of the generic specializer (which has to be shown only once) and the correctness of the interpreter.

1.4 Two-Level Interpreters: an Outline

We give an outline of our method. Assume that a state-of-the-art self-applicable partial evaluator (such as Similix [Bon91] or Schism [Con88]) and an interpreter Int for some programming language L is given. The goal is to specialize an L -program P with respect to some of its input D_1 by specializing the L -interpreter Int with respect to P and D_1 .

However, using such a partial evaluator one will hardly be successful in obtaining anything but trivial residual programs for the L -program P because P 's static input D_1 is ‘mixed’ with P 's dynamic input in the L -interpreter. That is, the input D_1 becomes dynamic in the interpreter and the information is lost at specialization time. We face the problem that the specialization of the interpreter with respect to P and D_1 does not yield efficient residual programs. Therefore we have to “binding-time improve” the subject program, i.e. the interpreter (and *not* the L -program P).

The essence of our method is to make more information available to the specializer at specialization time, via the interpreter. Since this information is made available by the interpreter rather than by transforming the interpreted subject program, the mechanism implemented in the interpreter is independent of the particular program P to be specialized.

There are various ways of collecting information about a program in an interpreter. In this paper we construct an interpreter which uses descriptions of S -expressions and collects information provided by conditionals. This information includes assertions (positive information) and restrictions (negative information) about S -expression. The collected information is then passed along the corresponding program branches.

In conclusion, the interpretive approach described in this paper has several advantages in comparison with the manual construction of specializers.

- In general it is easier to read, write and verify interpreters.
- The optimizations are integrated into the specializer by automatic means.
- One can instrument an interpreter with operations tracing various execution statistics (which results in a specializer doing these operations).
- One gets a specializer, a compiler, and an interpreter by writing an interpreter only.
- The code generation is handled by the generic specializer.
- One can share and reuse interpreters with different specializers. This provides a modular approach.

Some of the disadvantages are

- The residual language is fixed by the generic specializer.
- The control over the code generation phase is more difficult.

1.5 New Results

Specifically, our contributions in this paper are as follows:

- A method for *optimizing the specialization* using the existing partial evaluation technology.
- The *generation of stand-alone specializers* from interpreters using self-applicable partial evaluators.
- A technique of generating new specializers with different degrees of information propagation from *generic specializers*.

2 The New Approach

2.1 Ordinary Specialization vs. Interpretive Approach

A specializer α can be described as a program with two parameters: a subject program and a list of the static input values. The arguments of the subject program are classified as either static ('s') or dynamic ('d'). This classification is written as a superscript on programs, e.g. P^{sd} . This does not necessarily mean that the programs are binding-time analysed and annotated before specialization.

To minimize the notation we use **boldface** for data, i.e. programs and their input/output, and we use *slanted* for the semantic objects (i.e. if P is a program, then P denotes the corresponding input-output function).

Assume that a subject program P has two parameters and that the first argument D_1 is static. That is, the two parameters of the subject program P are classified as 'sd' (Fig. 1). The result of specializing the subject program P is a residual program Resid that returns the same result when given the remaining input D_2 as the subject program P when applied to the input D_1 and D_2 . This is called *ordinary specialization*.

The essence of the *interpretive method* is to insert an interpreter between the subject program P and the program specializer α . Three levels are now involved in specialization: the specializer α , the interpreter Int and the subject program P (Fig. 1).

For notational convenience we assume that all interpreted programs have two parameters. We define Int to be an interpreter with three parameters: the interpreted program P , the first argument, and the second argument of P : $\text{Result} = \text{Int}(P, D_1, D_2)$.

2.2 Futamura Projections vs. Specializer Projections

Three new projections can be derived from the interpretive approach which define the generation of stand-alone specializers from interpreters.

It is well-known that compilers can be generated from interpreters by self-application of a program specializer as defined by the *Futamara projections* [Fut71]. Whereas the Futamura projections enable us to generate compilers, the *specializer projections* enable us to generate specializers (Fig. 2).

1. The *1st specializer projection* states that a program P can be specialized by specializing the interpreter Int with respect to P and — in contrast to the first Futamura projection — to the static input D_1 . The classification of the interpreter is 'ssd'. The result is a residual program Resid which is a specialized version of the program P .
2. The *2nd specializer projection* follows from the 1st projection by specializing the specializer α with respect to the interpreter Int and the classification 'ssd'. The result is a specializer Spec which takes a program P and some part D_1 of P 's input, and generates a residual program Resid .
3. The *3rd specializer projection* defines the generation of a specializer generator by specializing the specializer α with respect to the specializer α and the classification 'ssd'.

Note that the specializer generator Specgen and the compiler generator Cogen are the same in the third projections. In fact, this program implements a general currying function which can be used with different classifications (e.g. with 'sdd' to generate a compiler and with 'ssd' to generate a specializer).

In contrast to the Futamura projections the specializer projections may be applied repeatedly (to generate specializers to generate specializers to ...). Note also that the Futamura projections can be regarded as a special case of the specializer projections (where D_1 is 'empty').

Applications of the specializer projections include, among others, *optimizing the specialization* of programs (by modifying the interpreter) and using existing specializers as *generic specializers* (by using the same specializer together with different interpreters).

3 Two-Level Interpreters

In this section we describe how to construct interpreters that can be used for generating new and opti-

$\text{Resid} = \alpha(\text{P}^{sd}, [\text{D}_1])$	$\text{Resid} = \alpha(\text{Int}^{ssd}, [\text{P}, \text{D}_1])$
--	---

Figure 1: Ordinary specialization vs. specialization via an interpreter

Futamura Projections	Specializer Projections
Target = $\alpha(\text{Int}^{sdd}, [\text{P}])$	Resid = $\alpha(\text{Int}^{ssd}, [\text{P}, \text{D}_1])$
Result = $\text{Target}(\text{D}_1, \text{D}_2)$	Result = $\text{Resid}(\text{D}_2)$
Comp = $\alpha(\alpha^{sd}, [\text{Int}^{sdd}])$	Spec = $\alpha(\alpha^{sd}, [\text{Int}^{ssd}])$
Target = $\text{Comp}(\text{P})$	Resid = $\text{Spec}(\text{P}, \text{D}_1)$
Cogen = $\alpha(\alpha^{sd}, [\alpha^{sd}])$	Specgen = $\alpha(\alpha^{sd}, [\alpha^{sd}])$
Comp = $\text{Cogen}(\text{Int}^{sdd})$	Spec = $\text{Specgen}(\text{Int}^{ssd})$

Figure 2: The Futamura projections and the specializer projections

mizing specializers. The next section presents a complete example.

In the following we assume that the specializer is a self-applicable partial evaluator for a first-order functional language (e.g. the classical partial evaluator Mix [JSS89]). We do not require partially static data-structures or more sophisticated specialization techniques. That is, we show that our methods may produce interesting results even if applied to the simplest self-applicable partial evaluator.

3.1 Step 1: Writing an Interpreter for Partial Evaluation

The first step is to write a straightforward interpreter Int_0 for a language L (not necessarily the subject language of the specializer). We run the interpreter in the following way:

$$\text{Int}_0(\text{P}, \text{D}) = \text{Result}$$

and can specialize it to get a target program:

$$\text{Mix}(\text{Int}_0^{sd}, [\text{P}]) = \text{Target-program-of-P}$$

3.2 Step 2: Writing a Two-Level Interpreter

In the second step we modify the interpreter in such a way that it takes two lists of values as input, keeping in mind that we are going to specialize the interpreter with respect to an L-program and some of the program's input. Assume that the first list contains only values that will be static and the second list all values that will be dynamic at specialization time. We call such an interpreter a *two-level interpreter* ('unmixed interpreter'). We run the interpreter in the following way:

$$\text{Int}_1(\text{P}, \text{D}_1, \text{D}_2) = \text{Result}$$

and specialize it in the following way:

$$\text{Mix}(\text{Int}_1^{ssd}, [\text{P}, \text{D}_1]) = \text{Residual-program-of-P}$$

Our goal is to generate a non-trivial residual program for P (written in the target language of the specializer) and not just a residual program of Int_1 . Thus we have to ensure that all operations in Int_1 depending only on P and D_1 will be static at specialization time. This is done by using two environments. When the value of a variable depends only on values known at specialization time then the first ('static') environment is updated, otherwise the second ('dynamic') environment is updated.

The principle is simple: each time the interpreter performs an operation it first tries to execute the operation using only the first ('static') environment. Only if this fails the second ('dynamic') environment is accessed. Consider for example a conditional: if the test can be decided using the first ('static') environment then the interpreter will do so. That is, the test can be performed at specialization time since it requires only static data. Whether the interpreter needs only the first ('static') environment for the interpretation of an expression can be decided either 'on-line' (during the interpretation), or 'off-line' (using a pre-analysis which gives the necessary hints to the interpreter e.g. in the form of annotations).

This second step enables us to generate a non-trivial specializer for L automatically (instead of writing it by hand). Note that any two-level interpreter can also be used for generating a compiler. That is, we get a specializer, a compiler, and an interpreter by writing an interpreter only!

3.3 Step 3: Adding more Binding-Time Improvements

Consider the following example (written in some example language):

```
if x=17 then (if x=5 then 1 else 2) else 3
```

If we try to specialize this expression assuming that the value of x is dynamic by using the specializer generated from the L-interpreter (step 2), then the specialized expression will be identical to the original expression. Of course, this is not an optimal result since the test of the inner conditional is always false and the branch 1 is unreachable, i.e. the resulting program should be

```
if x=17 then 2 else 3
```

Most partial evaluators can not perform this kind of simplifications (e.g. Similix). In a more realistic example the two conditionals may be located in different parts of the program and only sometimes be evaluated after each other (this situation arises for example in the string pattern matcher in Section 4.4.1).

Such information can be used in the interpreter in a variety of ways. The essence of all these methods is that more information is made available to the specializer at specialization time, via the interpreter. Since this information is made available by the interpreter and not by transforming the subject program, the mechanism is independent of the particular program being specialized. The way in which such descriptions are maintained and used in the interpreter determines the degree of specialization and the specialization effects one wants to achieve. In this paper we construct an interpreter which uses descriptions of S-expressions and propagates all the information provided by conditionals.

A call to the interpreter then looks as follows (Desc are descriptions containing static values and information about dynamic values):

$$\text{Int}_2(P, \text{Desc}, D_2) = \text{Result}$$

The interpreter works very much like the two-level interpreter, except it can use the full generality of descriptions (for example, the description of x would be changed to 17 before interpreting the then-branch of the outer conditional). When specializing the interpreter the description is static and the call looks like:

$$\text{Mix}(\text{Int}_2^{ssd}, [P, \text{Desc}])$$

which will yield an optimized residual program of P .

4 Example: Generating Specializers

We performed a number of experiments with the generation of specializers using the specializer projections (Section 2) and writing different interpreters according to our recipe (Section 3).

- (i) By fixing the specializer and varying the interpreter we generated specializers with different degrees of information propagation. Writing interpreters for annotated/un-annotated versions of a language we generated off-/on-line specializers.
- (ii) By fixing the interpreter and varying the specializer we generated specializers inheriting different properties from the different generic specializers. For example, specializers generated from Unmix [Rom88] are implemented in a first-order subset of Scheme, while specializers generated from Similix are implemented in a higher-order subset of Scheme and inherit the semantics-preserving property (e.g. no computations are discarded).

In this section we will describe a complete example using Similix and an interpreter for a simple functional programming language. All run-times in this paper are given in cpu-seconds using Similix 5.0, Chez Scheme 3.2 and a Sparc Station 2/Sun OS 4.1 (excluding time for garbage collection, if any).

4.1 Similix

We will use Similix [Bon93], a self-applicable partial evaluator for a large subset of Scheme. An example of a call to Similix is:

```
(similix 'append '((a b c) ***) "append.sim")
```

Here `append` is the name of the function in the program `"append.sim"` (the usual list append program) which we want to specialize and `((a b c) ***)` is a specification of the input to `append`. The stars `***` represent the dynamic part of the input.

4.2 The Interpreter

4.2.1 Source Language

We use a simple, first-order Scheme-like source language (Fig. 3) for the interpreter to show the essence of our method. The language is not a Scheme subset since our equality is only true for two equal atoms (otherwise it returns false). Our actual implementation can in fact handle an extended language where

one can use expressions at all places where simple expressions can be used. This is done by having a *desugaring* phase that inserts appropriate let-bindings.

```

P ∈ Program ; D ∈ Definition ; E ∈ Expression
SE ∈ SimpleExpression ; K ∈ Constant ; V ∈ Variable
F ∈ ProcName ; T ∈ TestExpression

P ::= D*
D ::= (define (F V*) E)
E ::= (if T E E) | (let ((V E)) E)
    | (F SE*) | SE
SE ::= (quote K) | V | (cons SE SE)
    | (car SE) | (cdr SE)
T ::= (equal? SE SE) | (pair? SE)

```

Figure 3: Scheme-like language

4.2.2 Information Propagation

One way to improve the interpreter is to maintain descriptions of values. The interpreter also needs two environments. The first ‘static’ environment maps program variables to *descriptions* containing *configuration variables* as placeholders for unknown entities. The second ‘dynamic’ environment maps configuration variables to values. The interpreter is similar to the two-level interpreter in step 2, except that it maintains and uses descriptions during the interpretation.

If the interpreter evaluates an expression it tries to compute the value using the first environment only (i.e. the descriptions). In addition, it updates the descriptions using the information about values provided by the test in a conditional. Such information can only be relevant for the unknown parts of description, i.e. for the parts described by configuration variables. We use the techniques of information propagation described in [GK93].

In our interpreter configuration variables (c-variables) have the form (cv 0), (cv 1), etc. and descriptions of S-expressions are defined by the following grammar

```

Desc ::= (quote Atom) | (cons Desc Desc)
    | (cv Integer)

```

If we introduce a c-variable for the second argument of `match` then the interpreter is called as follows (we use a shorthand notation for `cons`, and write `'(a b)` instead of `(cons 'a (cons 'b '()))` etc.):

```
(int 'match '( (a b a) (cv 0)) '( (b a b a)) pgm)
```

where `'(a b a) (cv 0)` is the description of the input and `(b a b a)` is the value of `(cv 0)`.

We can now specialize the interpreter. At specialization time the description environment holds the static values and the c-variable environment the dynamic values. In our example the partial evaluator is called as follows

```
(similix 'int
        '(match '(a b a) (cv 0)) *** ,pgm)
      "int.sim")

```

where `int.sim` is the file containing the text of the unmixed interpreter.

In the interpreter (Appendix C) the environment `p` maps program variables to descriptions, `n` maps configuration variables to negative descriptions and `r` maps configuration variables to values. Negative information puts restrictions on c-variables, i.e. which values a c-variable will not take. For each c-variable there is a set of restrictions

```
Restr ::= (quote Atom) | cons | (cv Integer)
```

The function `Expr` evaluates expressions. If an expression is a conditional then `Expr` first calls the function `Test-s` which tries to determine the result of the test by using the description environment alone, and only if this fails will it call `Test-d` which also uses the c-variable environment.

The function `refute` checks whether a list of binding of c-variables to a certain values is possible according to the negative information known about the c-variables. For example, if the interpreter is evaluating a test `(equal x '1)` and the description of `x` is `(cv 0)` then the interpreter will at some point call `refute` with the list of bindings `((cv 0) 1)` and if the negative information known about `(cv 0)` contains the information that `(cv 0)` cannot be equal to 1, then the call to `refute` will return true (i.e. there is a contradiction).

4.2.3 Normalizing Descriptions

Adding descriptions of program variables etc. to the original interpreter adds more static data to the specialization process of the interpreter. Since Similix, as most partial evaluators, folds (that is, creates a call to a previously generated residual function) only if two specialization points are identical, this may cause to non-termination in more cases than in ordinary specialization. Furthermore, if the same description can have different representations, Similix will not consider them as equal and this may cause even more

non-termination. Our solution to this last problem is to normalize the representation of descriptions.

4.3 Generating an Optimizing Specializer

We have generated an optimizing specializer, which will be denoted by β , from the interpreter using the compiler generator `Cogen` produced by Similix (this is equivalent to the 2nd specializer projection).

The generation time of β was 10 s. The size of β is 6437 cells (the number of “cons” cells needed to represent it in memory). The generation time and size compare favorably with the numbers obtained for compiler generation according to the Futamura projections (Fig. 2). The specializer β inherits its languages and functionality from Similix and the interpreter. The specializer β is too large to be shown here, but we will characterize its properties.

Subject, Residual and Implementation Language

Both the implementation and the residual language of the specializer β are determined by the residual language of Similix, that is, a large high-order subset of Scheme. The subject language of β is determined by the interpreter `Int` (as in the case of compiler generation according to the 2nd Futamura projection). In our example, the subject language is a first-order Scheme-like language.

Functionality

The specializer β inherits its folding strategy and the code generation part from Similix, the information propagation mechanism from the interpreter. Since the interpreter accepts programs ‘as they are’, that is, without binding-time annotations, so does the generated specializer β . The new specializer β can be characterized as an on-line, polyvariant program specializer.

The interpreter `Int` is written in such a way that it maintains sufficient descriptions to detect all unreachable branches without referring to actual values (cf. [GK93]). Since neither the interpreter nor Similix performs generalization, and since descriptions maintained in the interpreter are static, the generated specializer β will also recognize all unreachable branches without referring to dynamic data (i.e. there is no information loss caused by generalizing static values). So, the generated specializer β corresponds to a simple supercompiler, and achieves some of the same strong specializations for this reason.

Termination and Generalization

During program specialization there is always the risk of non-termination, and the generated specializer faces the same problem. Since all available non-trivial program specializers written by hand have termination problems, so this is not a drawback of specializer generation.

4.4 Examples of Optimized Specialization

4.4.1 Specializing a Naïve String Pattern Matcher

As an example we show that the generated specializer β is strong enough to achieve the efficiency of a matcher generated by the Knuth, Morris & Pratt algorithm [KMP77] by specializing a naïve string pattern matcher with respect to a fixed pattern. We use string pattern matching since this example is often used to compare the power of binding-time improvements [CD89, JGS93] and stronger specialization strategies [FN88, GT90].

The subject program is a naïve string pattern matcher which checks whether a string `p` (the pattern) occurs within another string `s`. The matcher is straightforward: the function `match` searches a string `match` for a position where `p` is a prefix of the rest of the string. The strategy used is not optimal because the same elements in the string may be tested several times. In case of a mismatch the string is shifted by one and no further information is used for advancing the string.

```
(define (match p s)
  (if (equal? (prefix? p s) '#t)
      'Success
      (if (pair? s)
          (match p (cdr s))
          'Fail)))

(define (prefix? p s)
  (if (pair? p)
      (if (pair? s)
          (if (equal? (car p) (car s))
              (prefix? (cdr p) (cdr s))
              '#f)
          '#f)
      '#t))
```

The residual program generated by the specializer β for the pattern `(a b a b)` is shown in Appendix A. It is equivalent to the one obtained by Consel and Danvy [CD89].

In addition, we applied our specializer to two other versions of the string pattern matcher: one is a tail-recursive version and one is a binding-time improved

version of the naïve matcher (from [JGS93]). In both cases exactly the same optimal residual program was generated.

4.4.2 Example of Bounded Static Variation

This section shows that our specializer is also strong enough to handle *bounded static variation* automatically, a case which usually requires a binding-time improvement named “The Trick” [JGS93].

As an example we show how a general regular expression matcher can be specialized with respect to a specific regular expression to produce a dedicated matcher in the form of a DFA (deterministic finite automaton). The example was first developed by Bondorf, Jørgensen and Mogensen.

The matcher `match` works by examining the first symbol `sym` of the string `s` and if this symbol is a possible first symbol of any string generated by the regular expression `r` then it proceeds with a new regular expression (`next r sym`) which generate the rest of the string, if and only if `r` generate the whole string. A detailed description can be found in [JGS93]. The core part of the matcher is shown below.

```
(define (match r s)
  (if (pair? s)
      (let ((sym (car s)))
        (if (equal? (member sym (first r)) '#t)
            (match (next r sym) (cdr s))
            '#f))
      (accept-empty? r)))

(define (member e s)
  (if (pair? s)
      (if (equal? e (car s))
          '#t
          (member e (cdr s)))
      '#f))
```

If this program is specialized with respect to a given regular expression, e.g. $(ab|bab)^*$, using an existing partial evaluator, such as Similix, one will not get a ‘good’ result since the regular expression will be classified as dynamic, because `(next r sym)` depends on `sym` which is dynamic.

The trick is to recognize that `sym` can only be bound to one of the first symbols of `r` computed by `(first r)` (static bounded variation). This insight was used to modify the original matcher to one where the regular expression will be classified as static (see [JGS93] for the details).

If we specialize the matcher using our specializer we directly obtain the same optimal result (Appendix B), that is, a residual program which corresponds ex-

actly to a three-state automaton as derived by standard methods.

4.4.3 Performance of the Specializer

To illustrate the performance of the specializer we compare the time required to specialize the naïve string pattern matcher `M` and the binding-time improved string pattern matcher `Mbti` (from [JGS93]) with respect to the pattern `Pat = (a b a b)` via the interpreter `Int`, and using our generated specializer β . The run-times (Fig. 4) illustrate that

1. the specialization using β is typically 10 times faster than the specialization via the interpreter. This shows that the interpretive overhead caused by the intermediate interpreter was removed during the generation of the specializer β .
2. the automatically generated specializer β is not much slower compared to the hand-written specializer Similix (note that the run times of Similix do *not* include the time required for its pre-processing phase and that β is an on-line specializer). On the one hand the propagation of additional information requires extra work in the specializer β , on the other hand the efficiency of the resulting program may be improved considerably.

However, a fair comparison of Similix and β is not possible because the subject language of Similix is much larger and the specializers generate different residual programs (except in the case of the `Mbti`).

4.4.4 Specializing a Naïve Interpreter

We showed how to achieve non-trivial specialization via a binding-time improved interpreter using a partial evaluator. An intriguing question: is the generated specializer β strong enough to obtain non-trivial specialization by specializing a program via a naïve interpreter? The answer is yes!

To illustrate this point we wrote a naïve self-interpreter in the functional language defined in Subsection 4.2.1 (this is a slightly simplified version of our original interpreter, simplified because our language is not as rich as the source language of Similix), and specialized the self-interpreter with respect to the naïve matcher and the same pattern as in Section 4.4.1. The self-interpreter `Int` was written in a straight-forward way and not optimized with efficiency or specialization in mind.

$$\beta(\text{Int}^{ssd}, [\text{M}, \text{Pat}]) = \text{Kmp-matcher}$$

	$Similix(M^{sd}, [Pat])$	$Similix(M_{bti}^{sd}, [Pat])$	$Similix(Int^{ssd}, [M, Pat])$	$\beta(M^{sd}, [Pat])$
Run-Time	120 ms	290 ms	5810 ms	470 ms

Figure 4: Performance

The generated residual program is textually identical to the matcher generated in Subsection 4.4.1!

This results shows that one can arrive at a strong specializer by starting from a partial evaluator and that the next step (using a naïve interpreter) profits from the previous step (writing a binding-time improved interpreter).

5 Related Work

This work is based on recent work elaborating the techniques of information propagation for a language with Lisp-like data structures [GK93]. A similar method is used for driving in the Refal supercompiler [Tur86]. Specializer generation and its potential applications were first discussed in [Glu91]. In [Tur93] it was shown that an interpreter can be used to improve supercompilation (however, the interpreter was not coupled with the supercompiler).

In [Jor92] it was shown that optimizing compilers can be generated from interpreters by self-application of a specializer. For example, an optimizing compiler for a realistic lazy functional language with pattern matching was obtained by carefully rewriting a straightforward interpreter.

Specializing a string pattern matcher is a typical problem to which several specialization methods have been applied [FN88, CD89, GT90]. Consel and Danvy showed that — after gaining a “small insight” on identifying static components in the naïve matcher and re-programming the naïve matcher — a partial evaluator can achieve the same non-trivial result [CD89]. The same kind of optimal matchers can be achieved with a Refal supercompiler [GT90]. However, we used only a small part of the whole supercompilation method embedded into an interpreter.

6 Conclusion and Future Work

We showed that the interpretive approach — inserting an interpreter between a subject program and a program specializer — can achieve stronger specializations than a program specializer alone. By using self-application we succeeded in generating a specializer

from an interpreter which is stronger than the partial evaluator used for its generation. The generated specializer corresponds to a simple supercompiler. These results can be seen as a step towards automating the construction of program specializers.

The results reported in this paper are also a strong evidence that one should not confine program specialization to compilation and the generation of compilers as stated by the Futamura projections. Indeed, we have applied program specialization to the problem of specializer construction.

Some intriguing questions remain: How much can the specialization be improved by a repeated application of the method? What is the minimal functionality a generic specializer must provide? Can one generate self-applicable specializers automatically from interpreters? The interpretive approach is new and it is difficult to say what the limitations are or how far it can be taken, but we believe that this will be a touchstone for the effectiveness of program specialization.

Acknowledgements

This work could not have been carried out without the pioneering work of Valentin Turchin. We greatly appreciate fruitful discussions with the members of the Refal group in Moscow and the TOPPS group at DIKU. Special thanks to Anders Bondorf, Olivier Danvy, Neil Jones, Andrei Klimov, Arkady Klimov, Julia Lawall, Alexander Romanenko, Sergei Romanenko, and David Sands.

References

- [Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Revision of paper in ESOP’90, LNCS 432, May 1990.
- [Bon93] Anders Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, Denmark, May 1993. Included in Similix distribution, 82 pages.
- [CD89] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.
- [CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Twentieth Annual ACM*

SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Charleston, South Carolina, pages 493–501, ACM Press, 1993.

- [Con88] Charles Consel. New insights into partial evaluation: the Schism experiment. In Harald Ganzinger, editor, *ESOP'88, Nancy, France. Lecture Notes in Computer Science 300*, pages 236–247, Springer-Verlag, March 1988.
- [FN88] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151, North-Holland, 1988.
- [Fut71] Yoshihiko Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
- [GK93] Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Static Analysis. Proceedings. Lecture Notes in Computer Science, Vol. 724*, pages 112–123, Springer-Verlag, 1993.
- [Glu91] Robert Glück. *On the generation of S→R-specializers*. Technical Report, University of Technology Vienna, 1991. (Presented at the NYU Partial Computation and Program Analysis Day. June 21, 1991, New York).
- [GT90] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the ISSAC '90 (Tokyo, Japan)*, pages 286–287, ACM Press, 1990.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
- [Jor92] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Albuquerque, New Mexico, pages 258–268, January 1992.
- [JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
- [KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *Siam Journal on Computing*, 6(2):323–350, 1977.
- [Rom88] Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463, North-Holland, 1988.
- [Tur86] Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
- [Tur93] Valentin F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

A The Residual Matcher

This appendix contains the residual program of the string pattern matcher (Section 4.4.1) specialized with respect to the pattern (a b a b):

```
(define (int-0 v*_0)
  (define (test-d-0-1 r_0)
    (if (pair? r_0)
        (let ([g_1 (cdr r_0)])
          (if (equal? 'a (car r_0))
              (test-d-0-3 g_1)
              (test-d-0-1 g_1)))
        'fail))
  (define (test-d-0-3 r_0)
    (if (pair? r_0)
        (let* ([g_1 (cdr r_0)] [g_2 (car r_0)])
          (if (equal? 'b g_2)
              (if (pair? g_1)
                  (let ([g_3 (cdr g_1)])
                    (if (equal? 'a (car g_1))
                        (if (pair? g_3)
                            (let* ([g_5 (cdr g_3)]
                                    [g_6 (car g_3)])
                              (if (equal? 'b g_6)
                                  'success
                                  (test-d-1-9 g_6 g_5)))
                            'fail)
                        (test-d-0-1 g_3)))
                  'fail)
              (test-d-1-9 g_2 g_1)))
        'fail))
  (define (test-d-1-9 r_0 r_1)
    (if (equal? 'a r_0)
        (test-d-0-3 r_1)
        (test-d-0-1 r_1)))
  (cdr v*_0)
  (test-d-0-1 (car v*_0)))
```

B The Residual Regular Expression Matcher

This appendix contains the residual program of the regular expression matcher (Section 4.4.2) specialized with respect to the regular expression (ab|bab)*:

```
(define (int-0 v*_0)
  (define (test-d-0-1 r_0)
    (if (pair? r_0)
        (let* ([g_1 (cdr r_0)] [g_2 (car r_0)])
          (if (equal? g_2 'a)
              (test-d-0-3 g_1)
              (and (equal? g_2 'b)
                   (pair? g_1)
                   (let ([g_3 (cdr g_1)])
                     (and (equal? (car g_1) 'a)
                          (test-d-0-3 g_3))))))
        '#t))
  (define (test-d-0-3 r_0)
    (and (pair? r_0)
```

```

(let ([g_1 (cdr r_0)])
  (and (equal? (car r_0) 'b)
       (test-d-0-1 g_1))))
(cdr v*_0)
(test-d-0-1 (car v*_0))

```

C The Interpreter

This appendix contains a listing of the interpreter (Section 4.2). It is written in the subject language of the partial evaluator Similix [Bon93], which is a subset of Scheme extended with user-defined constructors and pattern matching. The `casematch`-form is used for pattern matching on ordinary Scheme S-expressions. The primitive operations are defined in a separate file "int.adt" and are not listed here.

```

(loadt "int.adt")
(define (int f p v* pgm)
  (let* ((p (mkinitp p))
        (pgm (elaborate pgm))
        (def (Pgm->Def f pgm)))
    (casematch
     (normcv (list (mkde (Def->Var* def) p)))
     ((ps s)
      (Expr (Def->Expr def) '()
            ps (restrict-ne s '())
            (restrict-cve
             s
             (mkcve (find-cv* p) v*))
            pgm))))))
(define (Expr E ES p n r pgm)
  (casematch E
   (('IF T E1 E2)
    (casematch (normcv p)
     ((p0 s)
      (let* ((r1 (restrict-cve s r))
            (n0 (restrict-ne s n))
            (i (length s)))
        (casematch (Test-s T p0 n0 i)
         (('true p1 n1)
          (Expr E1 ES p1 n1 r1 pgm))
         (('false p1 n1)
          (Expr E2 ES p1 n1 r1 pgm))
         (('bot T1 p1 n1 p2 n2)
          (Test-d
           T1 E1 E2 ES p1 n1 p2 n2 r1 i pgm))))))
   (('LET ((V E1)) E2)
    (Expr E1 (cons (cons V E2) ES) (push p) n r pgm))
   (('CALL F . SE*)
    (let ((def (Pgm->Def F pgm))
          (Expr (Def->Expr def) ES
                (cons (mkde (Def->Var* def)
                          (SEExpr-s* SE* p))
                     (pop p))
                n r pgm)))
      (else
       (cont ES (SEExpr-s E p) p n r pgm))))))
(define (cont ES SE p n r pgm)
  (casematch ES

```

```

('() (on_arg-d r SE))
(((V . E2) . ES1)
 (Expr E2 ES1 (bind-p V SE (pop p)) n r pgm)))
(define (SEExpr-s* SE* p)
  (if (null? SE*)
      ()
      (cons (SEExpr-s (car SE*) p)
            (SEExpr-s* (cdr SE*) p))))
(define (SEExpr-s SE p)
  (casematch SE
   (('QUOTE A) SE)
   (('CAR SE1)
    (casematch (SEExpr-s SE1 p)
     (('CONS A _) A)
     (_ (proj-error "car" SE))))
   (('CDR SE1)
    (casematch (SEExpr-s SE1 p)
     (('cons _ A) A)
     (_ (proj-error "cdr" SE))))
   (('CONS SE1 SE2)
    (list 'cons (SEExpr-s SE1 p) (SEExpr-s SE2 p)))
   (else
    (on_arg-s p SE))))
; Try to determine the outcome of the test
; from the descriptions alone.
(define (Test-s T p n i)
  (casematch T
   (('EQUAL? A1 A2)
    (let ((v1 (SEExpr-s A1 p))
          (v2 (SEExpr-s A2 p)))
      (casematch (list v1 v2)
       (('quote A) ('quote B))
       (if (equal? A B)
           (list 'true p n)
           (list 'false p n)))
      ((('cv X) ('cv Y))
       (if (equal? X Y)
           (list 'true p
                (add-neg
                 (list (cons v1 'cons))
                 n))
           (both (list 'equal? v1 v2)
                  (on_p (bind-s v1 v2 ()) p)
                  p
                  (on_n
                   (bind-s v1 (get-info v2) ())
                   n)
                  (list (cons v2 'cons))
                  (list
                   (cons v1 v2)
                   (cons v2 v1))))))
      ((('cv _) _)
       (both (list 'equal? v1 v2)
              (on_p (bind-s v1 v2 ()) p)
              p
              (on_n
               (bind-s v1 (get-info v2) ())
               n)
              n
              (list (cons v1 (get-info v2))))))
      ((_ ('cv _))

```

```

(both (list 'equal? v1 v2)
      (on_p (bind-s v2 v1 ()) p)
      p
      (on_n (bind-s v2 (get-info v1) ()) n)
      n
      ()
      (list (cons v2 (get-info v1))))))
(((cons _ _) _)
 (list 'false p n)
 (_ ('cons _ _))
 (list 'false p n))))
(('PAIR? A)
 (let ((v (SEExpr-s A p)))
  (casematch v
   (('cons _ _) (list 'true p n))
   (('quote _) (list 'false p n))
   (('cv Z)
    (let ((cv1 (list 'cv (+ 1 i)))
          (cv2 (list 'cv (+ 2 i))))
     (both (list 'pair? v)
            (on_p
             (bind-s
              v
              (list 'cons cv1 cv2) ())
             p)
            (on_n (bind-s v 'cons ()) n)
            n
            ()
            (list (cons v 'cons))))))))))

; Check the negative information!
(define (both T p1 p2 n1 n2 b1 b2)
  (if (refute b2 n2)
      (list 'false p2 n2)
      (list 'bot T p1 (add-neg b1 n1)
            p2 (add-neg b2 n2))))

; Do it the hard way!
(define (Test-d T E1 E2 ES p1 n1 p2 n2 r i pgm)
  (casematch T
   (('EQUAL? A1 A2)
    (if (equal?
         (on_arg-d r A1)
         (on_arg-d r A2))
        (Expr E1 ES p1 n1 r pgm)
        (Expr E2 ES p2 n2 r pgm))))
  (('PAIR? A)
   (let* ((w (on_arg-d r A)))
    (if (pair? w)
        (Expr E1 ES p1 n2
              (bind-d
               (list 'cv (+ 1 i))
               (car w)
               (bind-d
                (list 'cv (+ 2 i))
                (cdr w)
                r))
              pgm)
        (Expr E2 ES p2 n2 r pgm))))))

```

```

;-----
; Auxiliary functions:
;
; add binding to c-variable environment:
(define (bind-d V w r)
  (if (anil? r)
      (acons (acons V w) (anil))
      (if (equal? (acar (acar r)) V)
          (acons (acons V w) (acdr r))
          (acons
           (acar r)
           (bind-d V w (acdr r))))))

; make a c-variable environment:
(define (mkcve V* w*)
  (if (null? V*)
      (anil)
      (bind-d (car V*) (car w*)
              (mkcve (cdr V*) (cdr w*)))))

; substitute values for c-variable:
(define (on_arg-d r Arg)
  (casematch Arg
   (('quote A) A)
   (('cons A B)
    (cons (on_arg-d r A) (on_arg-d r B)))
   (('cv n)
    (lookup Arg r)))

; lookup in c-variable environment:
(define (lookup V r)
  (if (acons? r)
      (let ((elm (acar r)))
        (if (equal? (acar elm) V)
            (acdr elm)
            (lookup V (acdr r))))
      'error))

; restrict a c-variable environment:
(define (restrict-cve s r)
  (if (null? s)
      (anil)
      (let ((d1 (car s)))
        (bind-d (cdr d1) (lookup (car d1) r)
                (restrict-cve (cdr s) r)))))

```