Generating Transformers for Deforestation and Supercompilation

Robert Glück* and Jesper Jørgensen

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

Abstract. Our aim is to study how the *interpretive approach* — inserting an interpreter between a source program and a program specializer — can be used to improve the transformation of programs and to automatically generate program transformers by self-application of a program specializer.
We show that a few semantics-preserving transformations applied to a straightforward interpretive definition of a first-order, call-by-name language are sufficient to generate Wadler's deforestation algorithm and a version of Turchin's supercompiler using a partial evaluator. The transformation is guided by the need to binding-time improve the interpreters.

1 Introduction

Our aim is to study the *interpretive approach* to improve the transformation of source programs and to automatically generate stand-alone transformers [Tur93, GJ94]. The essence of the interpretive approach is to insert an interpreter between a source program and a generic program specializer. As defined by the *specializer projections* one may generate stand-alone specializers from interpreters by self-application of a program specializer [Glü91, Glü94].

We show that a few semantics-preserving transformations on a straightforward interpretive definition of a first-order, functional language are sufficient to generate Wadler's *deforestation algorithm* and a version of Turchin's *supercompiler*. This is noteworthy because the generated transformers achieve stronger transformations than the self-applicable partial evaluator used for their generation. The source language is a first-order language with *call-by-name* semantics.

Deforestation, due to Wadler [Wad90], can eliminate intermediate data structures, thus reducing the number of passes over data structures and making programs more efficient. Supercompilation, due to Turchin [Tur86], is another powerful transformation method which can achieve both the effect of deforestation and partial evaluation. Supercompilation is based on *driving*, a unification-based transformation technique for functional languages. This makes supercompilation strictly stronger than partial evaluation and deforestation (for a detailed discussion see e.g. [GK93, SGJ94]).

The generation of transformers from interpretive definitions provides an intriguing factorization of the problem of specifying, implementing and ensuring

* Supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foundation (FWF) under grant J0780 & J0964.

the correctness of transformers. It provides a direct link between the evaluation function and a transformer. The *correctness* of a generated transformer is guaranteed (i) by the correctness of the generic specializer (which has to be shown only once, e.g. [Gom92, San94]), (ii) by the correctness of the interpretive definition (often it is easier to read, write and verify interpreters). The question of *code generation* is entirely handled by the generic specializer that produces the transformer. By abstracting from code generation issues, the interpretive definitions can be shared and reused (for example, transformers with different code-generating back-ends may be generated from the same interpreter).

Last but not least, the presented approach is *practical*, i.e. the transformers were generated using only existing partial evaluation techniques (the results shown in this paper were produced using Similix [Bon93], a self-applicable partial evaluator for a large subset of Scheme). But our methods work with any partial evaluator of 'Mix-type', such as the classical Mix system [JSS89]. No advanced partial evaluation techniques, such as partially-static structures, are required. In fact, using stronger specializers may considerably simplify the construction of the interpreters.

In this paper we shall mainly be concerned with the question of how to transform a straightforward interpretive definition for a first-order language in order to achieve deforestation and driving, without writing the corresponding transformers. The results obtained are promising and clearly justify the further investigation of the interpretive approach, both theoretically and practically.

2 Program Specialization and the Interpretive Approach

We present the interpretive approach of program transformation, formulate the three specializer projections and conclude with a brief review of partial evaluation and two-level interpreters.

2.1 The Interpretive Approach: Transformation via an Interpreter

The essence of the interpretive approach is to insert an interpreter between a source program and a program specializer. It is a surprising fact that, given an appropriate interpreter, this method may drastically increase the power of the overall transformation, as demonstrated in [Tur93, GJ94]. The same approach works for other program transformers, such as optimizing compilers [Jør92].

Let α be a program specializer with two parameters: a source program and a list of static input values. The arguments of the source program are classified as either 's' (static, known) or 'd' (dynamic, unknown). For notational convenience we write the classification as a superscript on source programs, e.g. P^{sd}. To minimize the notation we use boldface for programs and their input/output and *slanted* for the semantic objects (for example, if P is a program, then P denotes the corresponding input-output function).

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

In the *interpretive approach* an interpreter is inserted between the source program P and the specializer α (Fig. 1). Assume that all interpreted programs have two parameters. Define Int to be an interpreter with three parameters: the interpreted program and its two parameters, such that Result = $P(D_1,D_2)$ = $Int(P, D_1,D_2)$. It can easily be verified that the result of specializing the interpreter Int with respect to the program P and the data D_1 is a residual program which returns the same result given the remaining input D_2 as the program P when applied to the input D_1 and D_2. The classification of the interpreter is '*ssd*'. Three levels are now involved at specialization time: the specializer α, the interpreter Int and the source program P.

Figure 1: Ordinary specialization vs. specialization via an interpreter

2.2 Generating Transformers from Interpreters

Three projections can be formulated for 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 *Futamura projections* [Fut71]. Whereas the Futamura projections enable us to generate compilers, the *specializer projections* [Glü91, Glü94] enable us to generate specializers (Fig. 2). The generation of compilers according to the Futamura projections was first achieved by [JSS85]. Recently it was demonstrated that the generation of specializers according to the specializer projections is computationally feasible [GJ94].

The *1st specializer projection* states that a program P can be specialized by specializing the interpreter Int with respect to P and to the static input D_1. 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. 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 '*sd*'.

Note that the specializer generator Specgen and the compiler generator Cogen in the third projections are actually the same. In fact, they implement 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 from an interpreter).

2.3 Partial Evaluation and Binding-Time Analysis

Program specialization has by now more than proven its worth as a realistic program transformation paradigm. *Partial evaluation*, a specialization method based on constant propagation, as discussed in [JGS93], is an automatic instance of Burstall and Darlington's unfold/fold framework. In *off-line partial evaluation*

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

Figure 2: The Futamura projections and the specializer projections

the transformation process is guided by a *binding-time analysis* prior to the specialization phase. The result of the binding-time analysis is a program in which all expressions are annotated as either static or dynamic. Operations annotated as static are performed at specialization time, operations annotated as dynamic are delayed until run time (i.e. residual code is generated).

Despite the successful application of partial evaluation, one often faces the problem that some static information is not exploited in the way one would expect or hope. Traditionally, this problem is overcome by rewriting the source program. Such transformations are known as *binding-time improvements*, semantics-preserving transformations of a source program which enable the partial evaluator to make more operations static [JGS93, Bon93].

2.4 Two-level Interpreters: an Outline

Assume that a self-applicable, off-line partial evaluator and a straightforward L-interpreter Int written in the source language of the partial evaluator are 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 (cf. Sect. 2.1).

One will hardly be successful in obtaining anything but trivial residual programs for the L-program P using a straightforward L-interpreter because P's static input D_1 is usually 'mixed' with P's dynamic input in the interpreter. As a result, the input D_1 becomes dynamic in the interpreter and cannot be exploited at specialization time. In other words, one has to improve the binding-time of the source program, i.e. of the L-interpreter Int (and *not* of the L-program P).

The essence of our method [GJ94] is to make more information about L-programs available to a partial evaluator by transforming the L-interpreter (rather than by transforming each L-program individually). Depending on the properties of the interpreter different transformation effects can be achieved. Conceptually, we proceed in three steps:

(i) write a straightforward interpreter for L;
(ii) transform the L-interpreter into a semantically equivalent *two-level interpreter* that has two separate environments (keeping in mind that the interpreter will be specialized with respect to an L-program and one environment containing only the static input).
(iii) make more properties of L static by further modifying the L-interpreter.

Finally, an L-transformer can be generated from the L-interpreter by self-application of a partial evaluator, or by using a compiler-generator (cf. Sect. 2.2).

3 The Language

We will now demonstrate how to generate transformers for deforestation and supercompilation starting from a straightforward interpretive definition of a first-order functional language with a call-by-name semantics[2]. After introducing notational conventions, we define the syntax and semantics of the language.

Notation. The vector notation $\overline{e}$ denotes a sequence of zero or more syntactic objects of type e, e.g. expressions. If f is a function then $\overline{f}$ means "map f" (i.e. apply f to each element of a sequence). Sometime we write $\overline{R}[\![\overline{e}]\!]\rho$ for "mapping" $\lambda e.R[\![e]\!]\rho$ onto each element of the sequence $\overline{e}$.

The substitution of an expression e' for a variable x in an expression e is written as $e[e'/x]$. The environment that binds variables $\overline{x}$ to values $\overline{v}$ is written as $[\overline{x}\mapsto\overline{v}]$. Updating an environment ρ with a new binding is written as $[x\mapsto v]\rho$. Parallel substitution is denoted by $e[\overline{e}/\overline{x}]$ and parallel update by $[\overline{x}\mapsto\overline{v}]\rho$.

To distinguish elements in a sequence of syntactic objects we use the notation $\{\mathrm{M}_i\}_i$, where i ranges over some finite (unspecified or given by the context) set of index values.

Syntax. The language, the same as in [Wad90], has four syntactic categories (Fig. 3): variables x, constructor application $c(\overline{e})$, function application $f(\overline{e})$ and case-expressions `case` e `of` as including a sequence of alternatives as, each containing a pattern and an expression. Patterns are *linear*, i.e. no variable appears more than once in a pattern. Note that there is no 'catch-all' alternative in case-expressions because all patterns start with a constructor. For programs to be correct the patterns in case-expressions must be exhaustive. We assume that the only way a program can fail to evaluate is through non-termination. Values (returned when the interpretation terminates) are expressions build exclusively from constructor applications.

pgm	::= $f(\overline{x})=e$ ... $f(\overline{x})=e$	(programs)
e	::= x | $c(\overline{e})$ | $f(\overline{e})$ | `case` e `of` as	(expressions)
as	::= p`->`e ... p`->`e	(alternatives)
p	::= $c(\overline{x})$	(patterns)
v	::= $c(\overline{v})$	(values)

Figure 3: Syntactic categories

[2] The optimizations we are interested in – removing intermediate data structures and unification-based information propagation – are independent of the optimization achieved by lazy evaluation: sharing of computation. We do not consider the difference between call-by-name and lazy evaluation any further in this paper.

Semantics. The semantics of the language is specified by an interpretive definition (Fig. 4). The metalanguage used in the interpretive definition is strict. We use the brackets $[\![\ ]\!]$ only to separate terms of the object language from terms of the metalanguage (the definition is not compositional, i.e. not denotational).

$$
\begin{array}{lll}
(1.1) & R[\![e]\!] & = \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = E[\![e]\!] \textbf{ in } \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]) \\
(1.2) & E[\![c(\overline{e})]\!] & = c(\overline{e}) \\
(1.3) & E[\![f(\overline{e})]\!] & = E[\![e_f[\overline{e}/\overline{x}_f]]\!] \\
(1.4) & E[\![\texttt{case } e \texttt{ of } as]\!] & = \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = E[\![e]\!] \textbf{ in } E[\![e_{\mathrm{c}}[\overline{\mathrm{e}}/\overline{x}_{\mathrm{c}}]]\!]
\end{array}
$$

Figure 4: Semantics

The parameter containing the interpreted program *pgm* is omitted because the program does not change during the interpretation. Given the name of a function f defined in *pgm*, $\overline{x}_f$ and e_f denote the formal parameters and the body of f respectively. Given a sequence of alternatives *as* in a case-expression, $\overline{x}_c$ and e_c denote the variables in the pattern and the expression of the alternative selected with constructor c. The meta-variable c in rule (1.1) and (1.4) ranges over constructors and $\overline{\mathrm{e}}$ ranges over sequences of expressions. The expression $\mathrm{c}(\overline{\mathrm{e}})$ is a pattern on the meta level which, when matched against an object expression $c(\overline{e})$, binds c to c and $\overline{\mathrm{e}}$ to $\overline{e}$.

4 A Naïve Two-Level Interpreter

The next step after providing a straightforward interpreter, is to convert it into a two-level ('unmixed') interpreter (cf. Sect. 2.4). Conceptually, this can achieved by introducing two separate environments – one containing values that will be static and one containing values that will be dynamic at specialization time. However, the rewrite interpreter substitutes all values directly into program expressions. Our solution (Fig. 5) is to introduce an environment ρ and to use *placeholders* in program expressions in order to refer to *values* bound in the environment ρ (placeholders correspond to configuration-variables in [Tur86], see also [GK93]). All input values that will be dynamic at specialization time are bound to placeholders and all input values that will be static are substituted directly into program expressions, thus eliminating the need for a second environment. This provides the desired initial separation of static and dynamic values.

The only modification in the interpreter is the addition of a single rule accessing the environment ρ if a placeholder x is found. Otherwise, the rules of the interpreter are unchanged, only the environment ρ is carried around as extra parameter. In the following we will refer to placeholders simply as variables.

Theorem 1. *Let R_1 be the interpreter defined in Fig. 4 and R_2 the new interpreter defined in Fig. 5. Given an expression e with free variables $\overline{x}$ and values $\overline{e}$ then the two interpreters are equivalent, in the sense that*

$$
\begin{array}{lll}
(2.1) & R[\![e]\!]\rho & = \textbf{let } \mathrm{c}(\bar{\mathrm{e}}) = E[\![e]\!]\rho \textbf{ in } \mathrm{c}(\overline{R}[\![\bar{\mathrm{e}}]\!]\rho) \\
(2.2) & E[\![x]\!]\rho & = \rho[\![x]\!] \\
(2.3) & E[\![c(\overline{e})]\!]\rho & = c(\overline{e}) \\
(2.4) & E[\![f(\overline{e})]\!]\rho & = E[\![e_f[\overline{e}/\overline{x}_f]]\!]\rho \\
(2.5) & E[\![\texttt{case } e \texttt{ of } as]\!]\rho & = \textbf{let } \mathrm{c}(\bar{\mathrm{e}}) = E[\![e]\!]\rho \textbf{ in } E[\![e_c[\bar{\mathrm{e}}/\overline{x}_c]]\!]\rho
\end{array}
$$

Figure 5: Naïve two-level interpreter

1. *$R_1[\![e[\overline{e}/\overline{x}]]\!]$ terminates if and only if $R_2[\![e]\!][\overline{x}\mapsto\overline{e}]$ terminates.*
2. *If $R_1[\![e[\overline{e}/\overline{x}]]\!]$ terminates then $R_1[\![e[\overline{e}/\overline{x}]]\!] = R_2[\![e]\!][\overline{x}\mapsto\overline{e}]$*

However, the naïve two-level interpreter is not yet suited for the generation of a transformer using a partial evaluator because the result of E depends on the dynamic values in ρ. The expression $\mathrm{c}(\bar{\mathrm{e}})$ in rule (2.5) becomes dynamic, and since $\bar{\mathrm{e}}$ is substituted into the argument e_c of E, the syntactic argument of E is classified as dynamic. Similarly, in rule (2.1) where the syntactic argument of R becomes dynamic. As a result no computation on the static input can be performed in the interpreter at specialization time.

5 The Deforesting Interpreter

A few semantics-preserving transformations are sufficient to transform the naïve two-level interpreter into a *deforesting interpreter* that is well suited for the generation of the deforestation algorithm. Our goal is to binding-time improve the naïve two-level interpreter, so that the syntactic arguments of R and E become static. We do so by instantiating rule (2.1) to the four syntactic categories of the language and subsequently simplifying the instantiated rules by unfold/fold transformations. At the same time this constitutes a correctness proof of the deforesting interpreter relative to the naïve two-level interpreter.

5.1 Transforming the Two-level Interpreter

We transform the interpreter using transformations in the style of Burstall-Darlington [BD77]: instantiation and unfold/fold transformations. However, unfolding in a strict language is not always a semantics-preserving transformation (a program may terminate more often after unfolding). This should be checked for each individual step. The resulting transformed interpreter is shown in Fig. 6. We now explain each clause of it.

Instantiation with Variable. Let e in rule (2.1) be a variable x and assume that $\rho[\![x]\!] = c(\overline{v})$ for some value $c(\overline{v})$. The result of the transformation is rule (3.1).

$$
\begin{array}{ll}
& \mathrm{R}[\![x]\!]\rho \\
= & \textbf{let } \mathrm{c}(\bar{\mathrm{e}}) = \mathrm{E}[\![x]\!]\rho \textbf{ in } \mathrm{c}(\overline{R}[\![\bar{\mathrm{e}}]\!]\rho) \qquad \text{(inst. 2.1)}
\end{array}
$$

$= \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}}) = \rho[\![x]\!]\ \textbf{in}\ \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho)$ (unfold 2.2)
$= \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}}) = c(\overline{v})\ \textbf{in}\ \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho)$ (by assumption)
$= c(\overline{R}[\![\overline{v}]\!]\rho)$ (unfold let)
$= c(\overline{v})$ (by Lemma 2)
$= \rho[\![x]\!]$ (by assumption)

Lemma 2. *Let v be a value, then $\forall\rho.R[\![v]\!]\rho = v$.*
Proof. By induction over the structure of v. □

Instantiation with Constructor/Function Application. The instantiation of e in rule (2.1) with a constructor/function application is straightforward. The definition of R is instantiated and the call to E unfolded. The result follows from unfolding the resulting let-expression. This transformation is omitted. The resulting rules are (3.2) and (3.3).

Instantiation with Case-Expression. Let e in rule (2.1) be a case-expression `case` e' `of` as:

$R[\![\texttt{case}\ e'\ \texttt{of}\ as]\!]\rho$
$= \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}}) = E[\![\texttt{case}\ e'\ \texttt{of}\ as]\!]\rho\ \textbf{in}\ \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho)$ (inst. 2.1)
$= \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}}) = (\textbf{let}\ \mathrm{c}'(\overline{\mathrm{e}}') = E[\![e']\!]\rho\ \textbf{in}\ E[\![e_{\mathrm{c}'}[\overline{\mathrm{e}}'/\overline{x}_{\mathrm{c}'}]]\!]\rho)\textbf{in}\ \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho)$ (unfold 2.5)

Now we face the same problem: the syntactic arguments of E and R become dynamic at specialization time. We use the same strategy and instantiate e' to the four possible syntactic categories. In addition, we use the following property of the metalanguage: **let** y = (**let** x = e **in** f) **in** g = **let** x = e **in** (**let** y = f **in** g) if and only if x does not occur in g.

Let e' be a variable x. The result of the transformation is rule (3.4).

$R[\![\texttt{case}\ x\ \texttt{of}\ as]\!]\ \rho$
$= \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}}) = (\textbf{let}\ \mathrm{c}'(\overline{\mathrm{e}}') = \rho[\![x]\!]\ \textbf{in}\ E[\![e_{\mathrm{c}'}[\overline{\mathrm{e}}'/\overline{x}_{\mathrm{c}'}]]\!]\rho)\ \textbf{in}\ \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho)$ (unfold 2.2)
$= \textbf{let}\ \mathrm{c}'(\overline{\mathrm{e}}') = \rho[\![x]\!]\ \textbf{in}\ (\textbf{let}\ \mathrm{c}(\overline{\mathrm{e}}) = E[\![e_{\mathrm{c}'}[\overline{\mathrm{e}}'/\overline{x}_{\mathrm{c}'}]]\!]\rho\ \textbf{in}\ \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho))$ (swap lets)
$= \textbf{let}\ \mathrm{c}'(\overline{\mathrm{e}}') = \rho[\![x]\!]\ \textbf{in}\ R[\![e_{\mathrm{c}'}[\overline{\mathrm{e}}'/\overline{x}_{\mathrm{c}'}]]\!]\rho$ (fold 2.1)
$= \textbf{let}\ \mathrm{c}'(\overline{\mathrm{e}}') = \rho[\![x]\!]\ \textbf{in}\ R[\![e_{\mathrm{c}'}]\!][\overline{x}_{\mathrm{c}'}\mapsto\overline{\mathrm{e}}']\rho$ (Lemma 3)

Lemma 3. *Let $\overline{e}$ be values, e an expression and ρ an environment, then:*

1. *$R[\![e[\overline{e}/\overline{x}]]\!]\rho$ terminates if and only if $R[\![e]\!][\overline{x}\mapsto\overline{e}]\rho$ terminates.*
2. *If $R[\![e[\overline{e}/\overline{x}]]\!]\rho$ terminates then $R[\![e[\overline{e}/\overline{x}]]\!]\rho = R[\![e]\!][\overline{x}\mapsto\overline{e}]\rho$.*

Proof. By induction on the length of the computation (no. of unfoldings). □

The cases for constructor application and function application are similar and are omitted. The resulting rules are (3.5) and (3.6).

The last task is to instantiate e' to a case-expression. Let e' be `case` e'' `of` $\{c(\overline{x}''_c)\texttt{->}\ e''_c\}_c$. The resulting rule is (3.7).

$$
\begin{array}{ll}
& R[\![\texttt{case (case } e''\texttt{ of } \{c(\overline{x}''_c)\texttt{->}e''_c\}_c\texttt{) of } as]\!]\rho \\
= & \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = \textbf{let } \mathrm{c}'(\overline{\mathrm{e}}') = (\textbf{let } \mathrm{c}''(\overline{\mathrm{e}}'') = E[\![e'']\!]\rho \textbf{ in } E[\![e''_{c''}[\overline{\mathrm{e}}''/\overline{x}''_{c''}]]\!]\rho) & \text{(unf. 2.5)} \\
& \qquad \textbf{in } E[\![e_{c'}[\overline{\mathrm{e}}'/\overline{x}_{c'}]]\!]\rho \\
& \textbf{in } \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho) \\
= & \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = \textbf{let } \mathrm{c}''(\overline{\mathrm{e}}'') = E[\![e'']\!]\rho & \text{(swap lets)} \\
& \qquad \textbf{in let } \mathrm{c}'(\overline{\mathrm{e}}') = E[\![e''_{c''}[\overline{\mathrm{e}}''/\overline{x}''_{c''}]]\!]\rho \textbf{ in } E[\![e_{c'}[\overline{\mathrm{e}}'/\overline{x}_{c'}]]\!]\rho \\
& \textbf{in } \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho) \\
= & \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = \textbf{let } \mathrm{c}''(\overline{\mathrm{e}}'') = E[\![e'']\!]\rho \textbf{ in } E[\![\texttt{case } e''_{c''}[\overline{\mathrm{e}}''/\overline{x}''_{c''}] \texttt{ of } as]\!]\rho & \text{(fold 2.5)} \\
& \textbf{in } \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho) \\
= & \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = \textbf{let } \mathrm{c}''(\overline{\mathrm{e}}'') = E[\![e'']\!]\rho & \text{(prop. of subst.)} \\
& \qquad \textbf{in } E[\![(\texttt{case } e''_{c''}[\overline{y}_{c''}/\overline{x}''_{c''}] \texttt{ of } as)[\overline{\mathrm{e}}''/\overline{y}_{c''}]]\!]\rho \\
& \textbf{in } \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho) \\
= & \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = E[\![\texttt{case } e'' \texttt{ of } \{c(\overline{y}_c)\texttt{->(case } e''_c[\overline{y}_c/\overline{x}''_c] \texttt{ of } as)\}_c]\!]\rho & \text{(fold 2.5)} \\
& \textbf{in } \mathrm{c}(\overline{R}[\![\overline{\mathrm{e}}]\!]\rho) \\
= & R[\![\texttt{case } e''\texttt{of } \{c(\overline{y}_c)\texttt{->(case } e''_c[\overline{y}_c/\overline{x}''_c] \texttt{ of } as)\}_c]\!]\rho & \text{(fold 2.1)}
\end{array}
$$

The last rule (3.7) in Fig. 6 is responsible for the deforestation effect (and the fact that the source language is interpreted in a call-by-name manner): it pulls out the inner case-expression and pushes the outer case-expression into the alternatives of the inner case-expression.

The interpreter *never* builds intermediate data structures during the interpretation of a program (even if the interpreted program 'wants' to construct intermediate data structures). This can easily be seen by inspecting the interpreter: R is never called on expressions that are the result of another call to R. This means that any data structure decomposed by R must be part of the input and any data structure produced by R must be part of the output. Instead of deforesting the source programs, the deforestation is done 'on-line' by the interpreter. Thus, we call the transformed two-level interpreter a deforesting interpreter.

It is noteworthy that the *extensional behavior* of the interpreter is unchanged (i.e. the semantics of the defined language), only the intensional behavior of the interpreter is modified using semantics-preserving transformations.

$$
\begin{array}{lll}
(3.1) & R[\![x]\!]\rho & = \rho[\![x]\!] \\
(3.2) & R[\![c(\overline{e})]\!]\rho & = c(\overline{R}[\![\overline{e}]\!]\rho) \\
(3.3) & R[\![f(\overline{e})]\!]\rho & = R[\![e_f[\overline{e}/\overline{x}_f]]\!]\rho \\
(3.4) & R[\![\texttt{case } x \texttt{ of } as]\!]\rho & = \textbf{let } \mathrm{c}(\overline{\mathrm{e}}) = \rho[\![x]\!] \textbf{ in } R[\![e_c]\!][\overline{x}_c \mapsto \overline{\mathrm{e}}]\rho \\
(3.5) & R[\![\texttt{case } c(\overline{e}) \texttt{ of } as]\!]\rho & = R[\![e_c[\overline{e}/\overline{x}_c]]\!]\rho \\
(3.6) & R[\![\texttt{case } f(\overline{e}) \texttt{ of } as]\!]\rho & = R[\![\texttt{case } e_f[\overline{e}/\overline{x}_f] \texttt{ of } as]\!]\rho \\
(3.7) & R[\![\texttt{case (case } e \texttt{ of} & = R[\![\texttt{case } e \texttt{ of}\{c(\overline{y}_c)\texttt{->case } e_c[\overline{y}_c/\overline{x}_c] \texttt{of } as\}_c]\!]\rho \\
 & \quad \{c(\overline{x}_c)\texttt{->}e_c\}_c\texttt{) of } as]\!]\rho & \text{where } \overline{y}_c \text{ are fresh variables}
\end{array}
$$

Figure 6: Deforesting Interpreter

Theorem 4. *Let R_2 be the two-level interpreter defined in Fig. 5 and R_3 the new interpreter defined in Fig. 6. Given an environment ρ and an expression e the two interpreters are equivalent, in the sense that*

1. *$R_2[\![e]\!]\rho$ terminates if and only if $R_3[\![e]\!]\rho$ terminates.*
2. *If $R_2[\![e]\!]\rho$ terminates then $R_2[\![e]\!]\rho = R_3[\![e]\!]\rho$*

Proof. This follows from the development of the interpreter R_3 from R_2. □

5.2 The Generation of the Deforestation Algorithm

The deforesting interpreter is not efficient because it performs the deforestation during the interpretation of a source program. Using a partial evaluator one can convert the interpreter into a transformer that does the deforestation 'off-line'. Essential features of the transformer (code generation, memoization) are generated automatically by the partial evaluator.

Note the close connection to Wadler's deforestation algorithm [Wad90]. Replace R by T, erase the environment ρ and introduce code generation in Fig. 6 then the rules are the same as in the deforestation algorithm. In fact, this is almost what the partial evaluator (or compiler generator) does during the generation of the deforestation algorithm.

Correctness. There are two issues regarding the correctness of the generated transformer. Firstly, there is the *extensional correctness* of the transformer: given a source program, the transformer produces an equivalent residual program. This is a consequence of the Mix-equations [JGS93]. The (partial) correctness of the generated transformer follows from the (partial) correctness of the specializer and the correctness of the interpreter R which is a consequence of Theorem 4.

Secondly, there is the *intensional correctness* of the generated transformer, that is, the transformer does not only produces correct programs, but also programs that behave operationally in a certain way. In the case of deforestation this means, that the generated programs build no intermediate data structures. One may argue as follows: the partial evaluator used in the generation of the deforestation algorithm generates residual programs that consist only of operations that were classified as dynamic in the subject program. Therefore any operation performed by a residual program must have been present in the subject program. In the case of the deforesting interpreter (the subject program) we know that it never builds intermediate data structures during interpretation (not considering the syntactic arguments which are static), so the residual program generated from this interpreter will not build intermediate data structures either. A more detailed proof would require an operational analysis of the involved partial evaluator.

Termination. The deforestation algorithm is guaranteed to terminate if applied to a composition of *treeless terms* [Wad90]. Termination of the algorithm requires the insertion of memoization points just before applying rules (3.3) and (3.6). To guarantee the termination of the transformer it is sufficient that a new term encountered at a memoization point is a renaming of a previous term. Whenever a term is encountered for a second time, it is not transformed, but an

appropriate function application is inserted. If the deforestation algorithm is applied to other terms, its termination is not guaranteed (but when the algorithm terminates it returns an equivalent treeless term).

Insertion of Memoization Points. The partial evaluator does not only convert all dynamic statements in the deforesting interpreter into corresponding code generating statements, but also (automatically) inserts memoization points.

The places where memoization points are inserted in the generated deforestation algorithm can be determined automatically, but this depends on the strategy of the partial evaluator. There are two well-known methods: (i) structural induction and (ii) dynamic conditionals.

The Mix system inserts memoization points when function calls do not satisfy the *structural induction condition* [Ses88], i.e. the size of a static argument may increase. This is clearly the case on the right hand sides of the rules (3.3) and (3.6) where the syntactic argument f is replaced by the expression e_f. That is, this strategy automatically ensures the termination of the generated transformer.

Similix inserts memoization points at *dynamic conditionals* (and dynamic lambdas). This is only the case in rule (3.4) where the selection of the alternative depends on the dynamic value of x in the environment ρ. As a consequence the deforestation algorithm generated by Similix may fail to terminate, e.g. given the program `ones = 1:ones` (which makes sense in a call-by-name language). Fortunately, the user may turn off the default strategy and insert memoization points at any place, e.g. at the right hand sides of the rules (3.3) and (3.6).

Normalization. To guarantee termination of the generated deforestation algorithm it is sufficient that a term encountered at a memoization point is a *renaming* of a previous term. However, the partial evaluator used in the generation of the deforestation algorithm has no notion of what a proper renaming is. Most partial evaluators considers two object terms as equivalent only if they are textually identical. This may cause non-termination if the deforesting interpreter permanently introduces new variable names during the application of rule (3.7). To ensure that the generated transformer always recognizes two terms as equal when they are α-equivalent (i.e. equivalent under renaming of bound variables), a *normalization* of variable names is added to the interpreter just before the memoization points at rules (3.3) and (3.6).

Generating the Deforestation Algorithm using Similix. The deforesting interpreter is almost ready for generating the corresponding deforestation algorithm using Similix as the generic program specializer. However, a technical problem is that e_{c} (recall that e_{c} is short for "select e_{c} in as") in

$$R[\![\texttt{case}\ x\ \texttt{of}\ as]\!]\rho\ =\ \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}})\texttt{ = }\rho[\![x]\!]\ \textbf{in}\ R[\![e_{\mathrm{c}}]\!][\overline{x}_{\mathrm{c}}{\mapsto}\overline{\mathrm{e}}]\rho$$

depends on c which is dynamic (the selection depends on the dynamic values in the environment ρ). Our solution to this problem is 'The Trick', a well-known binding-time improvement used in partial evaluation [JGS93]. The knowledge that c is always a constructor from a finite set of constructors which is statically given in the patterns of the alternatives as can be used to binding-time improve

the rule (in which e_c will be static):

$$R[\![\texttt{case}\ x\ \texttt{of}\ as]\!]\rho = \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}})\texttt{=}\ \rho[\![x]\!]\ \textbf{in}\ A[\![as]\!]\ \mathrm{c}\ \overline{\mathrm{e}}\ \rho$$

where A is defined as:

$$A[\![c(\overline{x})\texttt{->}e\texttt{;}\ as]\!]\ \mathrm{c}'\ \overline{\mathrm{e}}\ \rho = \textbf{if}\ c\texttt{=}\ \mathrm{c}'\ \textbf{then}\ R[\![e]\!][\overline{x}\mapsto\overline{\mathrm{e}}]\rho\ \textbf{else}\ A[\![as]\!]\ \mathrm{c}\ \overline{\mathrm{e}}\ \rho$$

We generated a deforestation algorithm using the compiler generator from Similix and annotating the deforesting interpreter (written in Scheme) in such a way that only the values in the environment ρ are dynamic, while all other arguments of the interpreter are static.

The generation time was 2.51 s[3]. The size of the transformer is 2416 cons cells (the number of 'cons' needed to represent the program in memory). The generated deforestation algorithm is an L→Scheme-transformer implemented in Scheme. It accepts L programs as input and generates Scheme programs as output. The generated transformer is too large to be shown here, but we will illustrate it with some examples.

5.3 Examples

Deforestation. The transformation of `append (append` xs ys`)` zs is a 'classical' example and noteworthy because the initial program requires passing the list xs twice, whereas the transformed program passes xs only once. While the initial program requires $2|xs| + |ys|$ steps to compute, the residual program takes only $|xs| + |ys|$ steps. The residual program produced by the generated deforestation algorithm is tail-recursive and equivalent to the one shown in [Wad90].

Specialization. It is less known that deforestation can do program specialization. For example, transforming `append [1,2]` ys using the generated deforestation algorithm yields `1:2:`ys. In fact, rule (3.5) in the deforestation interpreter is responsible for this effect. Wadler's deforestation algorithm does just the same. This application of deforestation might be surprising, but has been observed earlier [SGJ94].

Combined Deforestation and Specialization. The effect of deforestation and specialization can be combined as illustrated by the transformation of `append (append` xs `[1,2])` zs. The result is a residual program with a single recursion on the list xs. While the initial program requires $2|xs| + |[1,2]|$ steps to compute, the residual program requires only $|xs|$ steps. Note that partial evaluation (without deforestation) is not able to achieve this effect. Transforming

```
goal xs zs = append (append xs [1,2]) zs
append xs ys = case xs of
                 []      -> ys
                 (x1:xs1)-> x1:append xs1 ys
```

[3] Run-times 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).

with the generated deforestation algorithm yields[4]

```
append0 xs zs = case zs of
                  []      -> 1:2:zs
                  (x1:xs1)-> x1:append0 xs1 zs
```

6 The Driving Interpreter

The power of unification-based information propagation has been recognized very early in the context of partial deduction, but, for various reasons, has taken much longer to be understood in the context of functional languages.

Adding this capability to the deforesting interpreter and generating a supercompiler for this language is surprisingly simple. This is due to the restriction of case-expressions to linear patterns, i.e. there is no need to enforce equality between variables in a pattern. To propagate *assertions* (positive information) about the structure of a variable x, it is sufficient to replace all occurrences of x by the constructor-expression defined by the corresponding pattern. While this does not change the result of the interpretation, it makes more information static. It is easy to verify the correctness of the modification. Rule (3.4)

$$R[\![\texttt{case}\ x\ \texttt{of}\ as]\!]\rho\ =\ \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}})\texttt{ = }\rho[\![x]\!]\ \textbf{in}\ R[\![e_{\mathrm{c}}]\!][\overline{x}_{\mathrm{c}}\mapsto\overline{\mathrm{e}}]\rho$$

is replaced by:

$$R[\![\texttt{case}\ x\ \texttt{of}\ as]\!]\rho\ =\ \textbf{let}\ \mathrm{c}(\overline{\mathrm{e}})\texttt{ = }\rho[\![x]\!]\ \textbf{in}\ R[\![e_{\mathrm{c}}[p_{\mathrm{c}}/x]]\!][\overline{x}_{\mathrm{c}}\mapsto\overline{\mathrm{e}}]\rho$$

6.1 The Generation of a Supercompiler

It is a surprising fact that this, seemingly small change, radically increases the power of the generated transformer, allowing transformations which neither the generic partial evaluator (i.e. Similix) nor the deforestation algorithm can perform. For example, the transformer is now able to pass 'the KMP test' and to perform a certain class of theorem proving tasks.

In fact, the information propagation in the generated transformer is *perfect* [GK93] in the sense that all redundant branches introduced by unfolding function definitions can be removed. This is due to the fact that all patterns in case-expressions start with a constructor and do not contain 'catch all' cases. Consequently, there is no need to propagate *restrictions* (negative information).

The restriction to linear and exhaustive patterns can be overcome by propagating assertions and restrictions as shown in [GK93]. Essentially the same method was used in [GJ94] for generating optimizing specializers. Another way to overcome the restriction to linear patterns is to introduce an if-construct with an equality test in the language [SGJ94].

[4] The residual programs are generated in Scheme, but here they are rewritten in the source language for the sake of readability. Note that programs generated by the deforestation algorithm are evaluation order independent. Appendix A shows the actual output in Scheme.

Correctness. The extensional correctness of the generated supercompiler follows from the (partial) correctness of the specializer, the correctness of the driving interpreter ensured by Theorem 4 and the correctness of the above modification.

The intensional correctness can be argued as follows: it is clear that the transformer performs deforestation, even though the interpreter may introduce new constructors when applying the new rule. The new rule essentially performs a unification of a variable and a pattern. The result of this unification is always a constructor application $c(\overline{x})$ which then replaces all occurrences of the variable in the corresponding branch of the case-expression. This operation is fully static in the driving interpreter. Therefore this unification-based information propagation is performed in the generated transformer. This is what driving does in supercompilation [Tur86].

Termination. Because static data can grow unboundedly, as a result of changing rule (3.7), the termination of the generated supercompiler can not be guaranteed as in the case of the deforestation algorithm. This is also true for supercompilers implemented by hand, so this not a drawback of automatic transformer generation. To ensure termination more sophisticated folding strategies are required. This is a topic of current research.

Generating a Supercompiler using Similix. We generated a version of the supercompiler using the compiler generator from Similix and annotating the driving interpreter in such a way that all values in the environment ρ are dynamic, while all other arguments are static. The generation time was 2.58 s. The size of the generated supercompiler is 2544 cons cells. It has the same technical characteristics as the generated deforestation algorithm plus the power of unification-based information propagation.

6.2 Examples

String Pattern Matching. The generated specializer is strong enough to achieve the efficiency of a matcher generated by the Knuth, Morris & Pratt algorithm by specializing a naïve bit-string pattern matcher with respect to a fixed pattern. We use 'the KMP Test' since this example is often used to compare the power of specializers [SGJ94]. Note that neither partial evaluation nor deforestation can achieve this effect directly.

Theorem Proving. Another interesting application is theorem proving. Let `plus` define the addition for numbers as used in recursive arithmetic. In this representation the symbol `0` stands for zero; and `S(n)` for the successor of `n`. The definition of `plus` is almost the same as the theoretical definition. The predicate `eqint` recursively defines the equality of two natural numbers. The example, proposed in [Tur80], shows the transformation of the program `p(x) = 0+x=x+0`. If it is possible to reduce the program to `p(x) = True` then one can say that the theorem `0+x=x+0` is proven. Indeed the generated supercompiler is almost able to reduce the theorem to `True`: the result is a recursive function which returns `True` for all numbers. To reduce this function to `p(x) = True` requires using induction. The initial program is:

```
p x = eqint (plus 0 x) (plus x 0)

plus x y =
  case x of
    0    -> y
    S(x1)-> S(plus x1 y)
```

```
eqint x y =
  case x of
    0    -> case y of
              0    -> True
              S(y1)-> False
    S(x1)-> case y of
              0    -> False
              S(y1)-> eqint x1 y1
```

The residual program is:

```
p0 x = case x of
         0    -> True
         S(x1)-> p0 x1
```

It should be added, that a slightly more difficult commutativity theorem `x+y=y+x` cannot be directly proven (the generated specializer does not terminate). In this case two inductions are necessary. As conjectured in [Tur80] this can be achieved with a metasystem transition, but this is beyond the scope of this paper.

7 Related Work

In the present paper we used a first-order language with a call-by-name semantics. This extends our previous work studying the generation of specializers for a Lisp-like language (i.e. a language with call-by-value semantics) [GJ94]. In [Tur93] it was shown that with the interpretive approach transformations become possible which the direct application of supercompilation cannot perform. Examples include the merging of iterative loops and function inversion (however, the interpreters were not coupled with the supercompiler and no transformers were generated).

The automatic generation of a compiler for a lazy higher-order functional language from an interpreter was studied in [Bon91]. In [Jør91, Jør92] it was shown that optimizing compilers for a realistic lazy functional language with pattern matching can be generated by carefully rewriting a straightforward interpreter.

8 Conclusion and Future Work

We showed that the interpretive approach can be used for generating the deforestation algorithm and a simple supercompiler from an interpretive definition of a first-order, call-by-name language. It is worth to note that the generated transformers are more powerful than the partial evaluator used for their generation.

A rather pleasing consequence of our formulation is that it shows the relation of partial evaluation, deforestation and supercompilation in a clear way. Driving can be considered as the most powerful method in this 'triumvirate', subsuming the transformations achieved by partial evaluation and deforestation.

During program specialization there is the risk of non-termination and the generated transformers face the same problem. Since all existing non-trivial program specializers implemented by hand have termination problems, this is not a drawback of automatic transformer generation.

Future work is desirable in several directions: How to tame termination and generalization in the generated transformers? Can one automatically generate more powerful, self-applicable specializers? What is the minimal functionality a generic specializer must provide?

Another interesting question is whether the generation of transformers can be done repeatedly in practice, using a form of bootstrapping [Glü94]. For example, it should be possible to couple the generated supercompiler with the interpreters (meta functions) suggested for the Refal supercompiler which may lead to transformations outside the scope of ordinary supercompilation and thereby achieving 'the stairway effect' of Turchin [Tur93].

Acknowledgements. We greatly appreciate fruitful discussions with the members of the Refal group in Moscow and the Topps group at DIKU. Special thanks to Morten Voetman Christiansen, Neil Jones, Andrei Klimov, Kristian Nielsen, Sergei Romanenko, David Sands, Morten H. Sørensen and Valentin Turchin. We would like to thank the Topps group for providing an excellent working environment.

References

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[Bon91] Anders Bondorf. Compiling laziness by partial evaluation. In [JHH91], pages 9–22, 1991.

[Bon93] Anders Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, Denmark, May 1993. Included in Similix distribution, 82 pages.

[Fut71] Yoshihiko Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[GJ94] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.

[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.

[Glü91] 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).

[Glü94] Robert Glück. On the generation of specializers. *Journal of Functional Programming*, 4(3):(to appear), 1994.

[Gom92] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM TOPLAS*, 14(2):147–172, 1992.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[JHH91] Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990. Workshops in Computing*. Springer-Verlag, August 1991.

[Jør91] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In [JHH91], pages 177–195, 1991.

[Jør92] 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.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140. Springer-Verlag, 1985.

[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.

[San94] D. Sands. Total correctness and improvement in the transformation of functional programs. Unpublished, May 1994.

[Ses88] Peter Sestoft. Automatic call unfolding in a partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Programming Languages and Systems - ESOP '94. Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500, Edinburgh, Scotland, 1994. Springer-Verlag.

[Tur80] Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657, Noordwijkerhout, Netherlands, 1980. Springer-Verlag.

[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.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Transformed Programs

This appendix shows a Scheme program produced by the generated deforestation algorithm (see Sect. 5.3). The functions `con->c` and `con->es` select the constructor name and the subexpressions in a constructor term. The function `build->con` builds a constructor application from a constructor name and a list of argument expressions.

```
(define (run-0 vs_0)
 (define (do-case-0-1 c_0 nvs_1 dvs_2)
  (if (equal? c_0 'nil)
      (buildcon
       'pair
       (list (buildcon 1 ()) (buildcon 'pair (list (buildcon 2 ()) dvs_2))))
      (let ([g_5 (car (cdr nvs_1))])
        (buildcon
         'pair
         (list (car nvs_1) (do-case-0-1 (con->c g_5) (con->es g_5) s_2))))))
 (let ([g_1 (car vs_0)])
   (do-case-0-1 (con->c g_1) (con->es g_1) (car (cdr vs_0)))))
```

This article was processed using the LaTeX macro package with LLNCS style