Higher-Order Expression Procedures

David Sands

DIKU, University of Copenhagen

dave@diku.dk

Abstract

We investigate the soundness of a specialisation technique due to Scherlis, *expression procedures*, in the context of a higher-order non-strict functional language. An expression procedure is a generalised procedure construct providing a contextually specialised definition. The addition of expression procedures thereby facilitates the manipulation and specialisation of programs. In the expression procedure approach, programs thus generalised are transformed by means of three key transformation rules: composition, application and abstraction.

Arguably, the most notable, yet most overlooked feature of the expression procedure approach to transformation, is that the transformation rules always preserve the meaning of programs. This is in contrast to the unfold-fold transformation rules of Burstall and Darlington. In Scherlis' thesis, this distinguishing property was shown to hold for a strict first-order language. Rules for call-by-name evaluation order were stated but not proved correct. In this paper we show that the expression procedure approach is correct for call-by-name evaluation, including lazy data structures and higher order functions.

1 Introduction

The declarative style of programming encourages, and benefits from, a modular approach to the construction of programs. A standard motivation for the study of program transformation in this setting is that the modular style comes at the expense of program efficiency. Program transformation can be seen as an attempt to reconcile these conflicting goals, permitting program construction in the modular style, and gaining efficiency through meaning-preserving transformations. Correctness is an essential ingredient of this argument. If a transformation method cannot ensure that the extensional meaning of a program is preserved, then the principal benefit of the modular approach to program construction, namely correctness, is lost.

In this paper we study the correctness of a transformation method for recursive programs in a higher-order functional language. The method was introduced by Scherlis [Sch81][Sch80] for a first-order language, and proved correct with respect to a call-by-value semantics.

1.1 Expression Procedures

The basic approach of Scherlis' transformation is that of contextual specialisation: a function f is specialised to the case where it occurs in some syntactic context C. This allows specialisation of subparts of a program so that the specific way in which functions are composed can be taken into account. Scherlis calls this *internal specialisation.* The transformation method is based on an extension of a language of recursion equations of the form $\mathtt{f}\,x \triangleq e$, to include *expression procedures.* An expression procedure has the form $e \overset{ep}{=} e'$, ie. its left-hand side is a complex term containing function symbols already defined in the program. The operational view of a normal recursive definition, $\mathtt{f}\,x \triangleq e$, is that of a computation rule for the evaluation of instances of the function f. An expression procedure of the form $C[\mathtt{f}] \overset{ep}{=} e'$ can be viewed as a specialised computation rule for instances of f, occurring in the context C. The addition of expression procedures, as an intermediate stage during program transformation, enhances the ability to perform program derivations, because it facilitates syntactic representation of the interaction of various program components. As a consequence of this addition, the set of transformation rules required to describe a wide range of program optimisations is very small.

Our view of expression procedures is as a framework in which specific transformation tactics can be described.

Slightly revised version of a paper to appear: ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation, La Jolla, 1995

Contributions This paper shows that the expression procedure approach extends to a wider class of languages. Specifically, the main contribution of this paper is a proof of total correctness for Scherlis' transformation rules in the context of a language with higher-order functions and lazy data structures. The proof itself is highly non-trivial, and proved to be resilient to the techniques previously developed by the author [San94, San95b]. The basic tool for the technical development is a notion of *weighted improvement*: a relation which compares the relative efficiency of expressions, with respect to a given weight associated to each function definition. This is a generalisation of the notion of *improvement* used in [San94, San95b], and is of independent interest.

2 Expression Procedure Transformation: An Example

We introduce the rules of Scherlis' transformation, in the context of a lazy higher-order language, by means of a simple example. We begin with the function definitions given below:

$$\begin{array}{lcl}
\mathtt{filter}\,p\,xs & \triangleq & \mathtt{case}\;xs\;\mathtt{of} \\
 & & \quad nil \Rightarrow \mathtt{nil} \\
 & & \quad y : ys \Rightarrow \mathtt{if}\;p\,y\;\mathtt{then}\;y : \mathtt{filter}\;p\;ys \\
 & & \qquad\qquad\qquad\quad \mathtt{else}\;\mathtt{filter}\;p\;ys \\
\mathtt{iterate}\;f\,x & \triangleq & x : (\mathtt{iterate}\,f\,(f\,x))
\end{array}$$

Now suppose we wish to compute the composed expression

$$\mathtt{filter}\,p\,(\mathtt{iterate}\,f\,x).$$

(So for example, $\mathtt{filter\ even\ (iterate}\,(\lambda y.1 + y)\,1)$ produces the infinite stream of even integers greater than one.)

For the outer call of filter to proceed, it is clear that the call to iterate must be computed. We can view this expression as a version of the function iterate, to be specialised to the case where the surrounding context is $\mathtt{filter}\,p\,[\,]$. For the first step of the transformation, we **compose** this context with the definition of iterate to obtain an expression procedure:

$$\mathtt{filter}\,p\,(\mathtt{iterate}\,f\,x) \overset{ep}{=} \mathtt{filter}\,p\,(x : \mathtt{iterate}\,f\,(f\,x))$$

Now we transform the body of the expression procedure by **application** of the definition of `filter` (unfolding the call) followed by **simplification** of the case expression, yielding:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{filter}\,p\,(\mathtt{iterate}\,f\,x)} \\
\overset{ep}{=} & \mathtt{if}\;p\,x\;\mathtt{then}\;x : \mathtt{filter}\,p\,(\mathtt{iterate}\,f\,(f\,x)) \\
 & \quad\mathtt{else}\;\mathtt{filter}\,p\,(\mathtt{iterate}\,f\,(f\,x))
\end{array}$$

Now we have obtained an expression procedure which can be applied recursively. However, it is not intended that expression procedures should be directly implemented, but we can "concretise" the effect of this recursion in two further steps. First we **abstract** the whole body of the expression procedure to define a new function `fit`:

$$\begin{array}{rcl}
 & & \mathtt{filter}\,p\,(\mathtt{iterate}\,f\,x) \\
 & \overset{ep}{=} & \mathtt{fit}\,p\,f\,x \\
\mathtt{fit}\,p\,f\,x & \triangleq & \mathtt{if}\;px\;\mathtt{then}\;x : \mathtt{filter}\,p(\mathtt{iterate}\,f\,(f\,x)) \\
 & & \qquad\qquad \mathtt{else}\;\mathtt{filter}\,p(\mathtt{iterate}\,f\,(f\,x))
\end{array}$$

Finally we **apply** the expression procedure to the right-hand side of the definition of `fit` to obtain

$$\begin{array}{rcl}
\mathtt{filter}\,p(\mathtt{iterate}\,f\,x) & \overset{ep}{=} & \mathtt{fit}\,p\,f\,x \\
\mathtt{fit}\,p\,f\,x & \triangleq & \mathtt{if}\;px\;\mathtt{then}\;x : \mathtt{fit}\,p\,f\,(f\,x) \\
 & & \qquad\qquad \mathtt{else}\;\mathtt{fit}\,p\,f\,(f\,x)
\end{array}$$

The example results in a definition which exhibits a modest improvement over the original expression, since it avoids an intermediate level of list-construction. It serves to illustrate:

- the basic rules of *composition*, *application* and *abstraction*, which, together with simplification and redundant-definition-elimination, form the basis of the transformation method;
- the application of Scherlis' rules to programs with higher-order functions, non-strict evaluation order, and lazy constructors.

The principle differences between the expression procedure transformation, and the well-known unfold-fold method [BD77], are that:

- recursion is not introduced by explicit *folding*, so the computation rules are never used "backwards";
- each transformation step only depends on the current definitions and expression procedures;
- the unfold fold method requires old versions of the function definitions in order to give *folding* any power.

In the remainder of this paper we show (also in contrast to the unfold-fold method) that the transformation method, applied to lazy higher-order programs, always produces equivalent programs.

3 Preliminaries

We summarise the notation used in the paper (which is different from [Sch80]), the language studied and its operational semantics.

We assume that a *basic program* a flat set of mutually recursive (and curried) function definitions of the form $\mathtt{f}\,x_1 \ldots x_{\alpha_\mathtt{f}} \triangleq e_\mathtt{f}$ where $\alpha_\mathtt{f}$, the arity of function f, is greater than zero. Let f, g, h ..., range over function names, $x, y, z \ldots$ over variables and e, e_1, $e_2 \ldots$ over expressions. The syntax of expressions is as follows:

$$\begin{array}{rcll} e & = & x \mid \mathtt{f} & \text{(Variable; Function name)}\\ & \mid & e_1\,e_2 & \text{(Application)}\\ & \mid & \mathtt{case}\ e\ \mathtt{of} & \text{(Case expressions)}\\ & & \quad c_1(\vec{x}_1) \Rightarrow e_1 \ldots c_n(\vec{x}_n) \Rightarrow e_n & \\ & \mid & c(\vec{e}) & \text{(Constructor expressions)}\\ & \mid & p(\vec{e}) & \text{(Strict primitive ops)} \end{array}$$

We assume that each constructor c and each primitive function p has a fixed arity, and that the constructors include constants (ie. constructors of arity zero). Constants will be written as c rather than $c()$.

The primitives and constructors are not curried - they cannot be written without their full complement of operands. The expression written $e_0\{^{\vec{e}}/_{\vec{x}}\}$ will denote simultaneous (capture-free) substitution of a sequence of expressions $\vec{e}$ for free occurrences of a sequence of variables $\vec{x}$, respectively, in the expression e_0. The term FV(e) will denote the free variables of expression e. A *context*, ranged over by C, C_1, etc. is an expression with zero or more "holes", [ ], in the place of some subexpressions; $C[e]$ is the expression obtained by replacing the holes with expression e. Contrasting with substitution, occurrences of free variables in e may become bound in $C[e]$; if $C[e]$ is closed then we say it is a *closing context* (for e).

3.1 Operational Semantics

The operational semantics defines an evaluation relation (a partial function) $\Downarrow$. If $e{\Downarrow}w$ for some closed expression e, then we say that e *evaluates to weak head normal form* w; when there is such a w, we say that e *converges*, and write $e{\Downarrow}$. For a given closed e, if there is no such w then we say the e *diverges*. We make no finer distinctions between divergent expressions, so "errors" and "loops" are identified. The weak head normal forms, $w, w_1, w_2, \ldots \in$ WHNF are just the constructor-expressions $c(\vec{e})$, and the partially applied functions, $\mathtt{f} e_1 \ldots e_k$, $0 \le k < \alpha_\mathtt{f}$. The operational semantics is a standard call-by-name one, and $\Downarrow$ is defined in terms of a one-step evaluation relation using the notion of a reduction context [FFK87]: Reduction contexts, ranged over by $I\!R$, are contexts containing a single hole which is used to identify the next expression to be evaluated (reduced).

Definition 1 (Reduction Contexts) *A reduction context $I\!R$ is given inductively by the following grammar*

$$\begin{array}{rcl} I\!R & = & [\,] \mid I\!R\, e\\ & \mid & \mathtt{case}\ I\!R\ \mathtt{of}\ c_1(\vec{x}_1) \Rightarrow e_1 \ldots c_n(\vec{x}_n) \Rightarrow e_n\\ & \mid & p(\vec{c}, I\!R, \vec{e}) \end{array}$$

Now we define the one step reduction relation on closed expressions. We assume that each primitive function p is given meaning by a partial function $[\![p]\!]$ from vectors of constants (according to the arity of p) to the constants (nullary constructors). We do not need to specify the exact set of primitive functions; it will suffice to note that they are strict (all operands must be evaluate to weak head normal form before the application of primitive-function can), and are only defined over constants, not over arbitrary weak head normal forms.

Definition 2 (One-step Reduction) *One-step reduction $\mapsto$ is the least relation on closed expressions, which is closed under reduction contexts, ie.*

$$e \mapsto e' \Rightarrow I\!R[e] \mapsto I\!R[e'].$$

and which satisfies the rules in Figure 1.

The one step evaluation relation is deterministic; this relies on the fact that if $e_1 \mapsto e_2$ then e_1 can be uniquely factored into a reduction context $I\!R$ and a redex e' such that $e_1 = I\!R[e']$.

Definition 3 (Convergence) *Closed expression e converges to weak head normal form w, $e{\Downarrow}w$, if and only if $e \mapsto^* w$ (where $\mapsto^*$ is the transitive reflexive closure of $\mapsto$).*

From this we define the standard notions of operational approximation and equivalence. The operational approximation we use is the standard Morris-style contextual ordering, or *observational approximation* [Plo75, Mil77]. The notion of "observation" we take is just the fact of convergence, as in the lazy lambda calculus [Abr90]. Operational equivalence equates two expressions if and only if in all closing contexts they give rise to the same observation - ie. either they both converge, or they both diverge. Note that for this language if we choose to observe more—such as the actual constructor produced, the observational approximation and equivalence relations will be unchanged.

$$\begin{array}{rcll} \mathtt{f}\; e_1 \ldots e_{\alpha_{\mathtt{f}}} & \mapsto & e\{{}^{e_1 \cdots e_{\alpha_{\mathtt{f}}}}/_{x_1 \ldots x_{\alpha \mathtt{f}}}\} & (\mathit{if}\;\; \mathtt{f}\; x_1 \ldots x_{\alpha_{\mathtt{f}}} \triangleq e) \\ \mathtt{case}\; c_i(\vec{e})\; \mathtt{of}\; c_1(\vec{x}_1) \Rightarrow e_1 \ldots c_n(\vec{x}_n) \Rightarrow e_n & \mapsto & e_i\{{}^{\vec{e}}/_{\vec{x}_i}\} & (0 < i \leq n) \\ p(\vec{c}) & \mapsto & c' & (\mathit{if}\;\; [\![p]\!]\vec{c} = c') \end{array}$$

Figure 1: One-step reduction rules

Definition 4 (Observational Equivalence)

(i) e observationally approximates e', $e \sqsubseteq\!\!\!\!\sim\; e'$, *if for all contexts C such that $C[e]$, $C[e']$ are closed, if $C[e]\Downarrow$ then $C[e']\Downarrow$.*

(ii) e *is* observationally equivalent *to* e', $e \cong e'$, *if* $e \sqsubseteq\!\!\!\!\sim\; e'$ *and* $e' \sqsubseteq\!\!\!\!\sim\; e$.

4 Expression Procedures and Correctness Preserving Program Transformation

In this section we describe Scherlis' basic rules for transformations of call-by-name programs, and define the key notions of *consistency* and *progressiveness* which will be instrumental in their correctness proof.

There are three fundamental transformation rules, which were illustrated in the introductory example: *abstraction* which introduces new basic definitions; *application* which unfolds either a basic definition, or an expression procedure (ie. replaces a substitution instance of the left-hand-side by the corresponding instance of the right), and *composition* which introduces a new expression procedure as a contextual instance of an existing definition. The other rules are *simplification* which use basic simplification properties of primitive functions, and finally *definition elimination* which eliminates redundant intermediate function/expression procedure definitions.

Generalised Programs A generalised program is one which comprises both basic definitions and expression procedures. Each transformation rule produces a new generalised program, and does not make reference to earlier programs in the transformation.

Expression procedures are intended only as intermediate forms in the transformation of programs, rather than a device which should be supported by a language implementation. But to discuss correctness, and why the method yields only equivalent programs, it is necessary to give *some* meaning to expression procedures. For the the purposes of this section, we follow Scherlis in giving:

- an *operational* interpretation of expression procedures as an extension of the evaluation process;
- *progressiveness* and *consistency* restrictions for expression procedures, motivated by the operational interpretation;
- intuitions as to why the transformation rules preserve consistency and progressiveness

For the moment, we keep in mind an informal notion of correctness, which is that any expression e computed using functions defined in the original program should yield the same "observable outcome" as it would when computed using the transformed functions.

4.1 Operational Interpretation of Expression Procedures

We can specify that an expression procedure, $e_1 \overset{ep}{=} e_2$, can be used in computation by simply extending the one-step evaluation rules.

Definition 5 (Extended One-step Evaluation) *With respect to a generalised program, let $\mapsto_{ep}$ denote the result of extending the one-step evaluation (Definition 2) rule according to the following rule-schema (for any reduction context $I\!\!R$ and substitution σ):*

$$I\!\!R[e_1\sigma] \mapsto_{ep} I\!\!R[e_2\sigma], \mathit{if}\; e_1 \overset{ep}{=} e_2$$

From this operational interpretation of expression procedures there are two natural correctness conditions with respect the basic definitions in the program, which Scherlis called *consistency* and *progressiveness.*

Consistency Notice that $\mapsto_{ep}$ is not deterministic, since if $e \mapsto_{ep} e'$ then it is no longer the case that e uniquely factorises into a redex and a reduction context. From the point of view of correctness, allowing the computation steps to be nondeterministic is not a problem: we just need to ensure that all the possibilities are operationally equivalent, so that computation using the expression procedures never leads to a different observable outcome. This is the *consistency* condition.

Progressiveness The second condition we require is that computation which employs expression procedures should never be able to loop in cases where $\mapsto$ converges[1]. This is the *progressiveness* condition. Informally, progressiveness requires that each expression procedure must represent a "useful" computation step.

Definition 6 (Consistency and Progressiveness) *Let EP be a set of expression procedures with respect to a given program P. We define EP to be*

(i) consistent *if for all expression procedures $e_1 \overset{ep}{=} e_2$ in EP we have that $e_1 \cong e_2$, and*

(ii) progressive *if for all closed expressions e, whenever $e \mapsto_{ep} e_1 \mapsto_{ep} \cdots \mapsto_{ep} e_i \mapsto_{ep} \cdots$ for some $\{e_i\}_{i \in \mathbf{N}}$, then e diverges—ie. there is no w such that $e {\Downarrow} w$.*

4.2 Transformation Rules

Now we give the definition of the transformation rules. We will need to employ a notion of *strictness* for syntactic contexts. We say that a context C is strict[2] if for all closed expressions e, and substitutions σ, if $C[e]\sigma$ converges then e must converge.

In the following let GP, GP' range over generalised programs, and a definition of the form $e \overset{*}{=} e'$ range over both basic definitions and expression procedures. We say that a pair of expressions $\langle e_1, e_2\rangle$ is a *definition instance* of GP, if there exists a definition $e \overset{*}{=} e'$ in GP (either basic or an expression procedure) such that $e\sigma = e_1$ and $e'\sigma = e_2$ for some substitution σ. We have the following transformation rules ([Sch80](p61–63), stated using our notation):

Definition 7 (Basic Transformation Rules) *For a given program GP, and any definition instance $\langle e_1, e_2\rangle$, we define the following transformation rules:*

(i) **Composition** *produces new program, GP', which is GP plus the expression procedure $C[e_1] \overset{ep}{=} C[e_2]$, for some strict context C.*

(ii) **Application** *produces a new program GP' by replacing some definition $e \overset{*}{=} C[e_1]$ in GP with $e \overset{*}{=} C[e_2]$.*

(iii) **Abstraction** *Let*

$$\begin{array}{rcl} e_1 & \overset{*}{=} & C_1[e\sigma_1] \\ & \cdots & \\ e_n & \overset{*}{=} & C_n[e\sigma_n] \end{array}$$

be definitions in a given program GP. Abstraction produces a new program GP' by replacing these definitions by new definitions:

$$\begin{array}{rcl} e_1 & \overset{*}{=} & C_1[(\mathtt{f}\,\vec{x})\sigma_1] \\ & \cdots & \\ e_n & \overset{*}{=} & C_n[(\mathtt{f}\,\vec{x})\sigma_n] \\ \mathtt{f}\,\vec{x} & \triangleq & e \end{array}$$

where $\mathrm{FV}(e) = \vec{x}$, and $\mathtt{f}$ is a new name.

In addition, there is a definition-elimination rule allows us to drop expression procedures from a program, and to eliminate unnecessary intermediate functions. We will not give any further consideration of this relatively straightforward rule.

There are also simplification rules which relate to properties of the primitive functions and the case-expression. We will need to place some mild restrictions on the *simplification laws* we use. Simplification laws are those operational equivalences which do not involve recursive function definitions (ie. their validity is independent of the definitions in the program. Simplification laws include the one-step evaluation rules for case- and primitive-functions, plus an all-important case-distribution laws (used form left to right)

$$\begin{array}{l} C[\mathtt{case}\ e\ \mathtt{of} \\ \qquad c_1(\vec{x}_1) \Rightarrow e_1 \ \ldots c_n(\vec{x}_n) \Rightarrow e_n] \\ \cong \ \ \mathtt{case}\ e\ \mathtt{of} \\ \qquad\quad c_1(\vec{x}_1) \Rightarrow C[e_1] \ \ldots c_n(\vec{x}_n) \Rightarrow C[e_n] \end{array}$$

for strict contexts C not containing any of the variables in $\vec{x}_1 \ldots \vec{x}_n$.

Note that (following Scherlis) we also used strict contexts in the definition of the composition rule for introducing an expression procedure. This is necessary to guarantee the progressiveness property. For example, given definitions

$$\begin{array}{rcl} \mathtt{loop}\,x & \triangleq & \mathtt{loop}\,(x+1) \\ \mathtt{K}\,x\,y & \triangleq & x \end{array}$$

If we allowed non-strict contexts, we could compose context $K\,x\,[\,]$ with the definition of `loop` to obtain

$$\mathtt{K}\,x\,(\mathtt{loop}\,y) \overset{ep}{=} \mathtt{K}\,x\,(\mathtt{loop}\,(y+1)),$$

which could give a looping computation on $\mathtt{K}\,1\,(\mathtt{loop}\,1)$ where there was none previously.

The importance of strictness properties for correct transformations on non-strict programs was noted by Runciman *et al* [RFJ89] in a study of correct *instantiation* in unfold-fold transformation. The use of strictness in [RFJ89] corresponds to its use in the above distribution law for case expressions. In many cases, a simple syntactic condition is enough to establish the necessary strictness property —eg. all reduction contexts $I\!\!R$ are strict.

[1] The other possibility, that computation using $\mapsto_{ep}$ gets "stuck" in an error state, is prevented by the consistency condition.

[2] In the case where C is a function application of the form $\mathtt{f}\,[\,]$, this coincides with the usual denotational definition of strictness

4.3 Additional Restriction on Abstraction

The additional restriction we impose concerns the contexts allowed in the definition of the abstraction rule. The restriction is that the holes are not allowed within expressions in argument positions ie. they are not allowed in the right hand expression of any application, so for example a context of the form $\mathtt{f}\ (C[\,])$ would not be allowed. This does not seem to present a problem in practice—in fact, the most common form of abstraction involves abstracting the whole right-hand side of an expression procedure, in which case the context is the trivial one '$[\,]$'.

Definition 8 (Abstractable Contexts) *We define* abstractable contexts*, ranged over by $A, A_1 \ldots$, to be contexts with zero or more occurrences of a hole,* $[\,]$*, by the following grammar:*

$$\begin{array}{rcl} A & = & e \ \mid\ [\,] \ \mid\ A\,e \ \mid\ p(A_1 \ldots A_n) \\ & \mid & \mathtt{case}\ A\ \mathtt{of}\ c_1(\vec{x}_1) \Rightarrow A_1 \ldots c_n(\vec{x}_n) \Rightarrow A_n \end{array}$$

This restriction on abstraction has some desirable properties, although its motivation is technical and relates to the proof of correctness. We do not know if the additional condition is necessary for progressiveness.

The restriction guarantees that the additional function call introduced by the abstraction has a "limited" effect. The intuition is as follows: suppose we perform an abstraction within the body of some expression procedure. Now suppose that we apply one step of computation using the new expression procedure. The abstraction has introduced an overhead of some additional function calls. The restriction guarantees that the number of times we incur the extra function call (with respect to a single use of this new expression procedure) is bounded by a constant, which independent of the arguments to the expression procedure. In a strict first-order language this property holds for all contexts.

5 Correctness

Viewing a program transformation as an operation which changes some definitions in a program (possibly also adding some new definitions), we say that the transformation is correct if, for any expression e which mentions functions defined in the original program, e converges using the original program if and only if e converges using the new definitions[3].

In the study of the correctness of program transformation in [San95b], an equivalent condition is given, reformulated in terms of observational equivalence (which implicitly refers to a fixed set of function definitions); a transformation is viewed as a process of deriving new functions $\mathtt{f}'_i$ (possibly plus some auxiliaries) from some existing $\mathtt{f}_i$. The transformation is then considered to be correct if $\mathtt{f}'_i \cong \mathtt{f}_i$. This formulation takes advantage of the fact that definitions of new functions conservatively extends operational equivalence.

In this paper we take the more direct view of correctness, and attempt to follow the general lines of Scherlis' proof for similar rules in the context of a strict first order language. Although the technical details of our proof are necessarily quite different from Scherlis', in essence correctness comes down to showing that the transformation rules introduce, and preserve, consistent and progressive expression procedures.

5.1 Weighted-Improvement

By far the most difficult part of the correctness proof concerns a proper treatment of progressiveness. Our definition of progressiveness, although simple, has rather subtle properties. For example, for any program P, assuming addition is one of our primitive functions, the consistent expression procedures $1 + 2 \overset{cp}{=} 2 + 1$ and $2 + 1 \overset{cp}{=} 1 + 2$, are individually progressive, but when taken together they are not.

We will capture the progressiveness property of the expression procedures in terms of a rather more tractable class of *improvement* relations. We characterise progressiveness by a less general property, *strict improvement*, which is adequate to prove progressiveness for the expression procedures obtained by the transformation rules.

We begin by defining a class of improvement relations. An improvement relation is a binary relation on expressions which is based on the relative "efficiency" of two expressions. Roughly speaking, improvement is a refinement of operational approximation which says that an expression e is improved by e' if, in all closing contexts, computation using e is no less efficient than when using e'. The measurement of efficiency is with respect to a given *weighting*, which assigns a positive integer to each function symbol in a given program. The "efficiency" is then the total "weight" of non-primitive function calls computed.

The improvement relations are instances of the improvement theories from [San91], and generalise the specific improvement theory used in [San95b] (although we do not give an associated improvement *theorem*). From this we define a notion of *strict improvement*, and show that if $\langle e, e' \rangle$ is in the strict improvement relation for

[3] This implies, for example, that if e evaluates to the constant `true` in the original program, it will do so in the new program (and vice-versa)—if this were not the case, then the expression $\mathtt{case}\ e\ \mathtt{of}\ \mathit{true} \Rightarrow \mathtt{true}$ would converge with respect to the original program, but not with respect to the new.

some weighting, then the expression procedure $e \overset{ep}{=} e'$ is progressive.

Finally, correctness follows by showing that the transformation rules preserve strict improvement property of expression procedures.

Weighting, and Evaluation Cost In the definitions that follow, assume a fixed program. We write $e \overset{\mathtt{f}}{\mapsto} e'$ if $e \mapsto e'$, and the reduction step involves a redex of the form $\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}$ (see definition 2). If $e \mapsto e'$ by reducing any other kind of redex, then we write $e \overset{\delta}{\mapsto} e'$.

A *weighting* is a mapping which assigns a positive integer to each function name (defined in a given program). Let $\Psi, \Psi_1 \ldots$ range over weightings.

Definition 9 (Weighted Cost) *For all closed expressions e, natural numbers n, and weightings Ψ, we say that, relative to Ψ, e converges with cost* **n** *to weak head normal form w, written $\Psi \vdash e{\Downarrow}^n w$, and defined inductively as follows:*

(i) if $e = w$ for some w then $\Psi \vdash e{\Downarrow}^0 w$;

(ii) if $e \overset{\delta}{\mapsto} e'$ and $\Psi \vdash e'{\Downarrow}^n w$ then $\Psi \vdash e{\Downarrow}^n w$;

(iii) if $e \overset{\mathtt{f}}{\mapsto} e'$ and $\Psi \vdash e'{\Downarrow}^m w$ then $\Psi \vdash e{\Downarrow}^n w$ where $n = m + \Psi(\mathtt{f})$.

It should be clear that we can make a judgement of the form $\Psi \vdash e{\Downarrow}^n w$ if and only if $e{\Downarrow}w$.

Improvement is defined in an analogous way to observational approximation:

Definition 10 (Improvement) e is improved by e' *with respect to Ψ, written $\Psi \vdash e \underset{\sim}{\trianglerighteq} e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed,*

$$\textit{if } \Psi \vdash C[e]{\Downarrow}^m w \textit{ then } \Psi \vdash C[e']{\Downarrow}^n w' \textit{ for some } n \leq m.$$

Immediately from the definition, it can be seen that, with respect to some weighting, $\underset{\sim}{\trianglerighteq}$ is a *precongruence* (transitive, reflexive, closed under contexts, ie. $e \underset{\sim}{\trianglerighteq} e' \Rightarrow C[e] \underset{\sim}{\trianglerighteq} C[e']$) and is a refinement of operational approximation, ie. $\Psi \vdash e \underset{\sim}{\trianglerighteq} e' \Rightarrow e \underset{\sim}{\sqsubset} e'$. It turns out that the improvement relations have an elegant characterisation in terms of a bisimulation-like definition [San91]. An outline of this characterisation below. We refer the reader to [San91] for the general theory, and [San95b] for a specific example (where the weight of all functions is one). We state the following properties of improvement, which follow either directly from the definition, or from the characterisation below.

Proposition 11 *For all weightings Ψ,*

(i) $e \mapsto e' \Rightarrow \Psi \vdash e \underset{\sim}{\trianglerighteq} e'$

(ii) $e \overset{\delta}{\mapsto} e' \Rightarrow \Psi \vdash e' \underset{\sim}{\trianglerighteq} e$

(iii) $\Psi \vdash \mathtt{f}\ x \underset{\sim}{\trianglerighteq} e$ if $\mathtt{f}\ x \triangleq e$

(iv) $$\begin{array}{l} \Psi \vdash C[\mathtt{case}\ x\ \mathtt{of} \\ \qquad\qquad c_1(\vec{y}_1) \Rightarrow e_1 \cdots c_n(\vec{y}_n) \Rightarrow e_n] \\ \quad \underset{\sim}{\trianglerighteq}\ \ \mathtt{case}\ x\ \mathtt{of} \\ \qquad\qquad c_1(\vec{y}_1) \Rightarrow C[e_1] \cdots c_n(\vec{y}_n) \Rightarrow C[e_n] \end{array}$$
if C is strict and does not use variables $\vec{y}_1 \ldots \vec{y}_n$

5.2 The Context Lemma for Weighted Improvement

Finding a more tractable characterisation of improvement (than that provided by Definition 10) is essential in establishing improvement laws. The characterisation we use says that two expressions are in the improvement relation if and only if they are contained in a certain kind of *simulation* relation. This is a form of *context lemma* [Mil77, Abr90, How89], and the proof of the characterisation uses previous technical results concerning a more general class of improvement relations [San91].

Definition 12 *For a fixed weighting Ψ, A relation $\mathcal{IR}$ on closed expressions is an* improvement simulation for *Ψ if whenever $e\ \mathcal{IR}\ e'$, if $\Psi \vdash e{\Downarrow}^m w_1$ then $\Psi \vdash e'{\Downarrow}^n w_2$ for some $n \leq m$ and some w_2 such that either:*

(i) $w_1 \equiv c(e_1 \ldots e_n)$, $w_2 \equiv c(e'_1 \ldots e'_n)$, *and $e_i\ \mathcal{IR}\ e'_i$, $(i \in 1 \ldots n)$,* or
(ii) $w_1 \equiv \mathtt{f}_1\, e_1 \ldots e_m$, $w_2 \equiv \mathtt{f}_2\, e'_1 \ldots e'_n$,
(ie. w_1 and w_2 are partially applied functions) and for all closed e_0, $(w_1\, e_0)\ \mathcal{IR}\ (w_2\, e_0)$.

So, intuitively, if an improvement-simulation relates e to e', then if e converges, e' does so at least as efficiently (with respect to Ψ), and yields a "similar" result, whose "components" are related by that improvement-simulation.

The key to reasoning about the improvement relation is the fact that, with respect to some weighting Ψ, $\underset{\sim}{\trianglerighteq}$, restricted to closed expressions, is itself an improvement simulation (and is in fact the *maximal* improvement simulation for Ψ). Furthermore, improvement on open expressions can be characterised in terms of improvement on all closed instances. This is summarised in the following:

Lemma 13 (Improvement Context-Lemma) *For all e, e', $\Psi \vdash e \underset{\sim}{\trianglerighteq} e'$ if and only if there exists an improvement simulation $\mathcal{IR}$ for Ψ such that for all closing substitutions σ, $e\sigma\ \mathcal{IR}\ e'\sigma$.*

The lemma provides a basic proof technique, sometimes called *co-induction*:

to show that $e \underset{\sim}{\triangleright} e'$ it is sufficient to find an improvement-simulation containing each closed instance of the pair.

5.3 Strict Improvement $\Rightarrow$ Progressiveness

The connection between improvement and progressiveness is via a definition of *strict* improvement. Strict improvement is a refinement of improvement which says that for strict contexts the cost of computation (relative to some weighting) must be *strictly* less. The simplest definition is in terms of the "tick" function of [San95b]:

Definition 14 (Strict Improvement) *Let $\surd$ denote an identity function, ie. $\surd x \triangleq x$, with the distinguished property that we assume $\Psi(\surd) = 1$ for all weightings Ψ.*

We define e to be strictly improved by e' with respect to Ψ, written $\Psi \vdash e \triangleright e'$, if $\Psi \vdash e \underset{\sim}{\triangleright} \surd e'$

Strict improvement satisfies the following properties (part of the "tick algebra" of [San95b]):

Proposition 15 *If $\Psi \vdash e \triangleright e'$, then with respect to Ψ we have:*

(i) $e\sigma \triangleright e'\sigma$ *for any substitution* σ

(ii) *if* $e_0 \underset{\sim}{\triangleright} e$ *and* $e' \underset{\sim}{\triangleright} e_1$ *then* $e_0 \triangleright e_1$

(iii) $C[e] \triangleright C[e']$ *for any strict context* C

(iv) $C[e] \underset{\sim}{\triangleright} C[e']$ *for any context* C*;*

(v) $\mathbb{R}[e] \triangleright \mathbb{R}[e']$

Theorem 16 *If there exists Ψ such that $\Psi \vdash e_1 \triangleright e_2$ for all expression procedures $e_1 \overset{cp}{=} e_2$ in some generalised program GP, then the expression procedures are progressive.*

We omit the details of the proof; the key property is that $\triangleright$ is a well-founded ordering on the set of convergent programs, and so every step made by the expression procedure rules must make progress, whenever progress is possible.

If an expression procedure is a strict improvement, then we will say that is *improvement-progressive.*

5.4 Improvement-Progressiveness $\Rightarrow$ Correctness

To prove correctness it is sufficient to show that the transformation rules preserve consistency and progressiveness, and it sufficient to consider the application of the rules to expression procedures only.

Proposition 17 *To prove correctness of the expression procedures method, it is sufficient to show, for all generalised programs with consistent and improvement-progressive expression procedures, that any application of either*

(i) the composition rule, or

(ii) the application, abstraction or simplification rules to any of the expression procedures

always yields consistent and improvement-progressive expression procedures.

Proof. (Outline) The crux of the proposition is that we do not have to consider the case where we apply transformations to the basic definitions. This is true because we can show that:

(i) if $\mathtt{f}\,\vec{x} \triangleq e$ is a definition in a given program then $\mathtt{f}\,\vec{x} \overset{cp}{=} e$ is a consistent and progressive expression procedure;

(ii) any transformation step applied to a definition $\mathtt{f}\,\vec{x} \triangleq e$ can be simulated by the corresponding transformation on the expression procedure $\mathtt{f}\,\vec{x} \overset{cp}{=} e$;

(iii) if $\mathtt{f}\,\vec{x} \overset{cp}{=} e'$ is a consistent and progressive expression procedure, then it is correct to replace a function definition $\mathtt{f}\,\vec{x} \triangleq e$ by $\mathtt{f}\,\vec{x} \triangleq e'$.

□

So we have reduced the correctness problem to preservation of consistency with respect to application of the transformation rules to expression procedures.

The main problem is proving progressiveness. With regard to consistency, we will state the following without proof:

Proposition 18 *For any generalised program with consistent expression procedures, a transformation step using either the composition rule, or the application, abstraction or simplification rules on any of the expression procedures, gives a generalised program with consistent expression procedures.*

This follows easily from the fact that operational equivalence is closed under substitution and under context (compatible) and that for any definition $\mathtt{f}\,\vec{x} \triangleq e$ we have that $\mathtt{f}\,\vec{x} \cong e$.

Following Proposition 17, to prove correctness it remains to show that that progressiveness is preserved:

Theorem 19 *For all generalised programs with improvement-progressive expression procedures, any application of either*

(i) the composition rule, or

(ii) the application or abstraction rules to any of the expression procedures

yields a generalised program with improvement-progressive expression procedures.

The difficult case is abstraction, since it introduces additional function calls. In this case we need to construct a new weighting which makes the resulting expression procedures improvement-progressive. First we need a couple of technical lemmas. The informal idea (Section 4) that abstractable contexts do not evaluate the expression placed in their holes more than a bounded number of times, is made precise by the following:

Lemma 20 *For all abstractable contexts A, there exists some $k \geq 1$ such that for all e*

$$\surd^k A[e] \underset{\sim}{\triangleright} A[\surd e],$$

where $\surd^k$ is k applications of $\surd$.

Proof. Induction in the structure of abstractable contexts. □

Lemma 21 *For any weighting Ψ and integer $k > 0$, let Ψ^{+k} be the weighting satisfying $\Psi^{+k}(\mathtt{g}) = \Psi(\mathtt{g}) + k$ for all functions $\mathtt{g}$ in the domain of Ψ (with the exception of $\surd$, for which we have $\Psi^{+k}(\surd) = 1$).*

If neither e_1, e_2 nor any definition in the program contains the distinguished function $\surd$, then

$$\Psi \vdash e_1 \triangleright e_2 \Longrightarrow \Psi^{+k} \vdash e_1 \triangleright \surd^k e_2.$$

Proof. Easy from the fact that for e not containing $\surd$, $\Psi \vdash e{\Downarrow}^n w$ implies $\Psi^{+k} \vdash e{\Downarrow}^{k \cdot n} w$. □

Proof. [Theorem 19] Let GP be a generalised programs with improvement-progressive expression procedures, with respect to weight Ψ. From the improvement-progressiveness assumption (and the fact that $\mathtt{f}\,\vec{x} \triangleright e$ if $\mathtt{f}\,\vec{x} \triangleq e$) for any definition instance $\langle e_1, e_2 \rangle$ of GP we have that $e_1 \triangleright e_2$. Now consider each transformation in turn:

Composition Composition introduces a new expression procedure of the form $C[e_1] \stackrel{ep}{=} C[e_2]$ from some definition instance $\langle e_1, e_2 \rangle$, some substitution σ, and some strict context C. Since $\Psi \vdash e_1 \triangleright e_2$ and since C is strict, by the properties of strict improvement (Proposition 15), $\Psi \vdash C[e_1\sigma] \triangleright C[e_2\sigma]$.

Application Suppose we replace an expression procedure $e \stackrel{ep}{=} C[e_1]$ by $e \stackrel{ep}{=} C[e_2]$, for some definition instance $\langle e_1, e_2 \rangle$. We know $\Psi \vdash e_1 \triangleright e_2$, and hence $\Psi \vdash C[e_1] \underset{\sim}{\triangleright} C[e_2]$. ($C$ is not necessarily strict.) Also, by improvement-progressiveness, $\Psi \vdash e \triangleright C[e_1]$, and hence $\Psi \vdash e \triangleright C[e_2]$.

Abstraction This is the most difficult case. We must construct a new weighting from Ψ which makes the new expression procedures progressive. Assume that we abstract a single expression procedure (the generalisation will be straightforward) $e_1 \stackrel{ep}{=} A[e\sigma]$, to obtain $e_1 \stackrel{ep}{=} A[(\mathtt{f}\,\vec{x})\sigma]$ and $\mathtt{f}\,\vec{x} \triangleq e$ for some abstractable context A, and substitution σ.

By Lemma 20, we have an integer k such that $\Phi \vdash \surd^k A[e\sigma] \underset{\sim}{\triangleright} A[\surd e\sigma]$. We will show that Φ', defined by:

$$\Phi'(h) = \begin{cases} 1 & \text{if } h = \mathtt{f} \\ \Phi(h) + k & \text{otherwise.} \end{cases}$$

is a weighting for the new program which makes all the expression procedures improvement-progressive. All unchanged expression procedures are improvement-progressive under Φ' (since $\mathtt{f}$ is unreachable from these expressions, under normal evaluation). By Lemma 21 we have

$$\Phi' \vdash e_1 \triangleright \surd^k A[e\sigma].$$

By the choice of k above, we get that

$$\Phi' \vdash \surd^k A[e\sigma] \underset{\sim}{\triangleright} A[\surd e\sigma],$$

and since $\mathtt{f}$ has weight 1, we also have that

$$\Phi' \vdash A[\surd e\sigma] \underset{\sim}{\triangleright} A[(\mathtt{f}\,\vec{x})\sigma].$$

Hence by the transitivity properties of strict improvement (Proposition 15(ii)) we can conclude that $\Phi' \vdash e_1 \triangleright A[(\mathtt{f}\,\vec{x})\sigma]$ as required. □

Finally, with the simple restriction that the simplification steps must be improvements (which is easily enforced, since eg. all right-linear equivalences involving only primitive functions and constants are improvements for any weighting), the correctness proof is complete.

6 Conclusions

The significance of the expression procedure method is that it is guaranteed to preserve the meaning of a program, and this property is built into the rules of the system. And yet, like the unfold-fold method, it is still very general in terms of the class of transformation "tactics" that it can describe. This paper has shown that this approach extends to a wider class of languages, namely higher-order functional languages with lazy data structures. We anticipate that Scherlis' call-by-value rules can be similarly extended to the higher-order case.

Expression procedures have received little attention in the literature on program transformation. We believe that is because the whole problem of correctness for "recursive" transformations on functional programs has been "conveniently forgotten". This fact is all the more striking when one draws comparison with the situation in logic programming; there it is almost always that case that specific transformation methods are expressed, and justified, within a framework which guarantees correctness (such as a variant of Tamaki and Sato's unfold-fold approach [TS84] —eg. [Deb88] [Ale92]).

Because of their "built-in" correctness property, we believe that the expression procedure method is a good high-level framework for describing more specific (and therefore more mechanisable) transformations. For example, Turchin's supercompilation methods [Tur86], applied to a non-strict language[4] can be described very naturally in terms of Scherlis' expression procedure transformations. Turchin describes transformations of REFAL (a first-order functional language) via the construction of a "process graph" (in the terminology of [GK93]). The aim is to construct a finite graph by "folding" equivalent nodes, according to some folding strategy (for example, syntactic equivalence modulo renaming) and finally to re-interpret the graph as a program. Expression procedures are a good fit for describing this style of transformation. The construction of a finite process graph can be represented by the construction of self-sufficient (recursive) expression procedures, which can be converted to program form by abstraction and application (in the manner of the last two steps of the example in Section 2). With the insights from Sørensen *et al*'s work, [Sør94] [SGJ94], the basic ingredients of supercompilation are also seen to be present in Scherlis' transformation: *driving* corresponds to distribution laws for conditionals, *information propagation* (in a general sense such as that of [FN88]) corresponds to *qualified* expression procedures (see [Sch80]). These components are also explicitly present in Wegbreit's early study of transformation [Weg76], and also in Wadler's *deforestation* [Wad90]. We have checked that the transformation in this paper is sufficient to provide a natural and direct encoding of the deforestation algorithm from [Wad90]. In this way we can show that the restrictions we have placed on the abstraction rule are not severe in practice[5]. We believe that it can also cope with higher-order extensions, thus providing an alternative correctness proof to [San95a].

We leave the details to a longer version of the paper. The significance of viewing the above transformations in the expression-procedure framework is that it ensures correctness by construction. The results of this paper have enabled us to extend this claim to a much wider class of languages.

6.1 Related Work

From the point of view of correctness (which is the main topic of this paper) we should compare the expression-procedure method with methods for guaranteed *correct* program transformation. Related methods for correct program transformation are reviewed extensively in [San94]. Scherlis compared his approach to that of Kott [Kot78] (rather less critically than the comparison in [San94]) and Manna and Waldinger [MW79]. Subsequently, the bulk of work on correct unfold-fold transformations is in the setting of logic programming (although references to Scherlis' method are rare). The first, and probably best-known approaches to correct unfold-fold transformations of logic programs is that of Tamaki and Sato [TS84]. As for many subsequent variations on their ideas, the conditions for correctness are dependent on the *transformation history*, rather than being built into the rules of the system themselves. To some degree this is inevitable, since the unfold-fold method always depends on older versions of functions in order to perform interesting fold-steps. In contrast, Bossi *et al* [BCE92] considered a replacement condition which is not history-sensitive—but it is a model theoretic condition, rather than a syntactic one. In previous work, the author also proposed a semantic condition, based on *improvement*, for correct transformations on a higher-order functional language [San94, San95b]; the method was used to obtain correct unfold-fold transformations, where sufficient aspects of the transformation "history" are built into the program itself, to obtain correct folding. This work is the most closely related to the goals of this paper, by virtue of the fact that it deals with correct transformations on higher-order functions. An advantage of the present approach is that it does away with all of the additional book-keeping (the "tick-algebra") from [San95b]. Compared to unfold-fold derivations using the tick algebra, expression procedures seem to handle transformations involving lazy data-structures in a smoother manner.

Acknowledgement and Further Work This paper grew out of an attempt to apply the improvement theorem from [San94] to verify the correctness of expression procedures in a higher-order language. However, the improvement theorem (as we noted in [San94]) is *not* able to prove the correctness of expression procedures, because of the (essential) abstraction rule. The idea of using a weighted improvement came from Torben Amtoft (Aarhus), (the use of "weights" also oc-

[4] Only partial correctness is preserved in a strict language since the domain of termination can be increased. Some of the conditions on Scherlis' rules for a strict language need to be relaxed to allow such transformations.

[5] For deforestation we do need the abstraction rule (in order to construct new definitions) but we never need to apply it in a "non-abstractable" context.

curs in Amtoft's study of unfold-fold transformations of logic programs [Amt93][Amt92]) when the author was explaining the details of the proof of the improvement theorem. At the time, we concluded that a "weighted" improvement theorem looked like an easy generalisation, but that there were no obvious applications.

A unifying topic for further work would be to show that the improvement theorem also holds for weighted improvement. We conjecture that it does, and that this is sufficient to give an alternative (but perhaps less direct) correctness proof for the transformation studied in this paper.

Another direction for further work is to consider languages with side-effects. Mason [Mas86][Ch. 7] extends Sherlis' transformation rules to a first-order Lisp with destructive operations, but leaves the correctness as a conjecture. It would be interesting to see if the techniques of this paper could be used in this context as well.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.

[Ale92] F. Alexandre. A technique for transforming logic programs by fold-unfold transformations. In *PLILP '92*, volume 631 of *LNCS*, pages 202–216. Springer-Verlag, 1992.

[Amt92] T. Amtoft. Unfold/fold transformations preserving termination properties. In *PLILP '92*, volume 631 of *LNCS*, pages 187–201. Springer-Verlag, 1992.

[Amt93] T. Amtoft. *Sharing of computations.* PhD thesis, DAIMI, Aahus University, 1993.

[BCE92] A. Bossi, N. Cocco, and S. Etalle. Transforming normal programs by replacement. In *Third Workshop on Meta-Programming in Logic, META 92*, 1992.

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24:44–67, January 1977.

[Deb88] S. Debray. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 297–307, 1988. SIGPLAN Notices 23(7).

[FFK87] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[FN88] Y. Futamura and K. Nogi. Generalised partial computation. In D. Bjørner, Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987.* North-Holland, 1988.

[GK93] R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In G.Filè P.Cousot, M.Falaschi and A.Rauzy, editors, *Static Analysis. Proceedings*, volume 724 of *LNCS*, pages 112–123. Springer-Verlag, 1993.

[How89] D. J. Howe. Equality in lazy computation systems. In *Fourth annual symposium on Logic In Computer Science*, pages 198–203. IEEE, 1989.

[Kot78] L. Kott. About transformation system: A theoretical study. In B. Robinet, editor, *Program Transformations*, pages 232–247. Dunod, 1978.

[Mas86] I. Mason. *The Semantics of Destructive Lisp.* Number 5 in CSLI Lecture Notes. CSLI, 1986.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4, 1977.

[MW79] Z. Manna and R. Waldinger. Synthesis: Dreams → programs. *Transactions on Programming Languages and Systems*, 5(4), 1979.

[Plo75] G. D. Plotkin. Call-by-name, Call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[RFJ89] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language: An approach to instantiation. In *Functional Programming, Glasgow 1989: Proceedings of the First Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer Verlag, August 1989.

[San91] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, pages 298–311, Skye, August 1991. Springer Workshop Series.

[San94] D. Sands. Total correctness and improvement in the transformation of functional programs. DIKU, University of Copenhagen, Unpublished (53 pages), May 1994.

[San95a] D. Sands. Correctness of recursion-based automatic program transformations. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE '95)*, LNCS. Springer-Verlag, 1995.

[San95b] D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1995.

[Sch80] W. Scherlis. *Expression Procedures and Program Derivation.* PhD thesis, Dept. of Computer Science, Stanford, 1980. Report No. STAN-CS-80-818.

[Sch81] W. L. Scherlis. Program improvement by internal specialisation. In *8'th Symposium on Principals of Programming Languages.* ACM, 1981.

[SGJ94] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP'94*. LNCS 788, Springer Verlag, 1994.

[Sør94] M H Sørensen. Turchin's supercompiler revisited: An operational theory of positive information propagation. Master's thesis, Department of Computer Science, University of Copenhagen, 1994.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S. Tarnlund, editor, *2nd International Logic Programming Conference*, pages 127–138, 1984.

[Tur86] V. F. Turchin. The concept of a supercompiler. *ToPLaS*, 8:292–325, July 1986.

[Wad90] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP 88, LNCS 300.

[Weg76] B. Wegbreit. Goal–directed program transformation. *IEEE Transactions on Sofware Engineering*, 2:69–80, June 1976.