Total Correctness by Local Improvement in Program Transformation

David Sands

University of Copenhagen*
e-mail: dave@diku.dk

Abstract

The goal of program transformation is to improve efficiency while preserving meaning. One of the best known transformation techniques is Burstall and Darlington's unfold-fold method. Unfortunately the unfold-fold method itself guarantees neither improvement in efficiency nor total-correctness. The correctness problem for unfold-fold is an instance of a strictly more general problem: transformation by locally equivalence-preserving steps does not necessarily preserve (global) equivalence.

This paper presents a condition for the total correctness of transformations on recursive programs, which, for the first time, deals with higher-order functional languages (both strict and non-strict) including lazy data structures. The main technical result is an *improvement theorem* which says that if the local transformation steps are guided by certain optimisation concerns (a fairly natural condition for a transformation), then correctness of the transformation follows.

The improvement theorem makes essential use of a formalised improvement-theory; as a rather pleasing corollary it also guarantees that the transformed program is a formal improvement over the original. The theorem has immediate practical consequences:

- It is a powerful tool for proving the correctness of existing transformation methods for higher-order functional programs, without having to ignore crucial factors such as *memoization* or *folding*. We have applied the theorem to obtain a particularly simple proof of correctness for a higher-order variant of *deforestation*.
- It yields a simple syntactic method for guiding and constraining the unfold/fold method in the general case so that total correctness (and improvement) is always guaranteed.

*DIKU, Universitetsparken 1, 2100 København Ø, DENMARK.

To appear: The 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 23-25, 1995

1 Introduction

The context of this study is transformations on functional programs. Source-to-source transformation methods for recursive programs, such as *unfold-fold transformation*, *partial evaluation* and *deforestation* [BD77] [JGS93][Wad90], proceed by performing a sequence of equivalence preserving steps on the definitions in a given program.

The main goal of such methods is to improve the efficiency of programs, but not at the expense of their meaning. Program transformations should preserve the *extensional* meaning of programs in order to be of any practical value. In this case we say that the transformation is *correct*.

The Problem: "Equivalence-preserving steps that do not preserve equivalence"

The problem is that for many transformation methods which deal with recursive programs (including those methods mentioned above), correctness cannot be argued by simply showing that the basic transformation steps are meaning preserving. Yet this observation (clarified below) runs contrary to many informal (and some formal) arguments which are used in attempts to justify correctness of particular transformation methods.

The problem arises through transformation steps for which "equivalence" depends critically on (ie. is *local* to) the function being transformed. A typical example of such a local equivalence would be an *unfold* step which replaces a recursive call to f by the corresponding instance of the body, or its inverse, a *fold* step. As a result, the new definition may not be semantically equivalent to the original, in particular since it may introduce new recursive structure which leads to worse termination properties. To take a concrete (but contrived) example to illustrate this point, consider the following transformation (where $\triangleq$ denotes a function definition, and $\cong$ is semantic equivalence with respect to the current definition):

$$\boxed{f\ x \triangleq x + 42} \xrightarrow[\text{using } 42 \cong f\ 0]{\text{transform}} \boxed{f\ x \triangleq x + f\ 0}$$

The problem comes from the fact that the equivalence used in a transformation step (in this example, the replacement of 42 by the call $f\ 0$) is not necessarily an equivalence with respect to the *new* definition.

Here is another example, expressed in unfold-fold steps.

Given the natural law for conditionals:

$$\texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z')\ \texttt{else}\ z\ \cong\ \texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z$$

we have the following unfold-fold transformation:

$$\begin{aligned}
\texttt{f}\ x\ &\triangleq\ \texttt{if}\ x\ \texttt{then}\ (\texttt{f}\ x)\ \texttt{else}\ \texttt{true}\\
&\xrightarrow{unfold} \texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ x\ \texttt{then}\ (\texttt{f}\ x)\ \texttt{else}\ \texttt{true}\ )\\
&\qquad\qquad \texttt{else}\ \texttt{true}\\
&\xrightarrow{unfold} \texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ x\ \texttt{then}\ (\texttt{f}\ x)\ \texttt{else}\ \texttt{true}\ )\\
&\qquad\qquad\qquad\qquad \texttt{else}\ \texttt{true})\\
&\qquad\qquad \texttt{else}\ \texttt{true}\\
&\xrightarrow{law\times 2} \texttt{if}\ x\ \texttt{then}\ (\texttt{f}\ x)\ \texttt{else}\ \texttt{true}\\
&\xrightarrow{fold} \texttt{f}\ x
\end{aligned}$$

The transformed definition is thus $\texttt{f}\ x \triangleq \texttt{f}\ x$, and illustrates the well-known fact that unfold-fold transformations do not, in general, preserve total correctness. It also serves as a counter-example to folk-law in functional programming which says that "more unfolds than folds" is sufficient to guarantee the correctness of the unfold-fold method.

Consequences

This problem has important consequences:

(i) Many transformation methods have not been proved correct, and it seems that the correctness problem has received little attention because of an implicit (and incorrect) assumption that it is sufficient to argue the correctness at the level of the basic steps. In particular we believe this to be the case for more realistic forms of partial evaluation, deforestation and supercompilation of functional languages, where arguments of this form are unsatisfactory because the transformations perform *memoization* or *folding*[1].

(ii) Some transformation methods simply do not preserve correctness in general. It is well known that this is the case for unfold-fold transformations.

The Contribution of this Work

This paper presents a solution to the problem which deals with higher-order functional languages (both strict and non-strict) including lazy data structures.

- The main technical result is an *improvement theorem* which says that if the transformation steps are guided by certain optimisation concerns (a fairly natural condition for a transformation), then correctness of the transformation follows.

  The above notion of optimisation is based on a formal *improvement-theory*. Roughly speaking, an expression e is improved by e' if in all closing contexts C, if $C[e]$ evaluates to some answer, $C[e']$ can also evaluate to some answer, but requires no more evaluation steps than $C[e]$. The important property of improvement from the point of view of program transformation is that it is substitutive—an expression can be improved by improving a sub-expression. For reasoning about the improvement relation a more tractable formulation and some related proof techniques are used.

  The *improvement theorem* shows that if e is improved by e' (in addition to e being operationally equivalent to e') then a transformation which replaces e by e' is totally correct; in addition this also guarantees that the transformed program is a formal improvement over the original. (Notice that in the above example, replacement of 42 by the equivalent term $f\ 0$ is not an improvement since the latter requires evaluation of an additional function call).

- The significance of the improvement theorem is that it finds immediate practical application to the consequences of the problem, namely:

  (i) Total correctness proofs for automatic transformations based on a higher-order variant of the well-known deforestation method [Wad90]. With a new formulation of the deforestation algorithm (extended to deal with higher-order functions) the proof of correctness, including the crucial *folding* process, becomes strikingly simple.

  (ii) A simple syntactic method for restricting the general (incorrect) unfold-fold method is provided. The method is based on a single annotation (whose meaning and algebraic properties are given by the improvement theory) which effectively guides and constrains transformations to guarantee correctness and improvement.

In this paper we illustrate only the second of these two applications. The details of the first are found in [San94]. The remainder is organised as follows:

Section 2 deals with preliminaries including the syntax and operational semantics of a simple higher-order functional language. In **Section 3** we give the formal definition of a transformation, and some standard partial-correctness results. In **Section 4** the definition and properties of improvement are given, and the improvement theorem is stated. **Section 5** considers the application of the improvement theorem to the problem of ensuring correctness in unfold-fold transformations. **Section 6** considers related work, and concludes with a discussion of further work. An appendix contains some technical details, leading up to a proof of the improvement theorem.

2 Preliminaries

We summarise some of the notation used in specifying the language and its operational semantics. The subject of this study will be an untyped higher-order non-strict functional language with lazy data-constructors. Our technical results will be specific to this language (and its call-by-name operational semantics), but the inclusion of a strict application operator and arbitrary strict primitive functions (which could include constructors and destructors for strict data structures) should be sufficient to convince the reader that similar results carry over to call-by-value languages.

We assume a flat set of mutually recursive function definitions of the form $\texttt{f}\ x_1 \ldots x_{\alpha_{\texttt{f}}} \triangleq e_{\texttt{f}}$ where $\alpha_{\texttt{f}}$, the arity of function $\texttt{f}$, is greater than zero. (For an indexed set of

[1] A number of rigorous studies of correctness in partial evaluation [Gom92][Pal93][Wan93] [CK93] ignore the memoization aspects and deal with the orthogonal issue of the correctness of *binding time analysis*, which controls *where* transformation occurs in a program.

functions we will sometimes refer to the arity by index, $\alpha_{\mathtt{i}}$, rather than function name.) $\mathtt{f}$, $\mathtt{g}$, $\mathtt{h}$ ..., 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:

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

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 $e\{\vec{e}'/\vec{x}\}$ 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. The term $\mathrm{FV}(e)$ will denote the free variables of expression e. Sometimes we will (informally) write substitutions of the form $\{\vec{e}/\vec{\mathtt{g}}\}$ to represent the replacement of occurrences of function symbols $\vec{\mathtt{g}}$ by expressions $\vec{e}$[2].

A *context*, ranged over by C, C_1, etc. is an expression with zero or more occurrences of a "hole", [ ], in the place of some subexpressions; $C[e]$ is the expression produced 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. A context is called *open* if it contains free variables.

2.1 Operational Semantics

The details of the operational semantics of the language are given in the appendix; the important point is that evaluation relation (a partial function) $\Downarrow$ is defined. If $e \Downarrow w$ for some closed expression e then we say that e *evaluates to weak head normal form* w.

The weak head normal forms, $w, w_1, w_2, \ldots \in \mathrm{WHNF}$ are just the constructor-expressions $c(\vec{e})$, and the partially applied functions, $\mathtt{f} e_1 \ldots e_k$, $0 \le k < \alpha_{\mathtt{f}}$.

For a given closed e, if $e \Downarrow w$ for some w then we say that e *converges*, and sometimes write $e \Downarrow$. Otherwise we say that e *diverges*. We make no finer distinctions between divergent expressions, so "errors" and "loops" are identified. The operational semantics is a standard call-by-name one, and $\Downarrow$ is defined in terms of a one-step evaluation relation.

2.2 Approximation and Equivalence

Here we define operational ordering on expressions, $\lesssim$ and its associated equivalence $\cong$. 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].

Observational 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:

Definition 2.1

(i) e observationally approximates e', $e \lesssim e'$, *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* $e \lesssim e'$ *and* $e' \lesssim e$.

NB. For this language, if we choose to observe more than just convergence, such as the actual constructor produced, the observational approximation and equivalence relations will be unaffected. Note that we can "observe" the difference between "$\bot$" and "$\lambda x.\bot$", which is the natural choice in an untyped theory of equivalence (for both strict and lazy languages). The distinction remains even if we impose a type discipline on our language, and choose only to observe programs of non-function type (cf. [Plo75]). This is due to the inclusion of a strict version of application (@).

3 Transformation and Basic Correctness Results

The essence of the definition is that a transformation consists of equivalence-preserving modifications to the bodies of a set of definitions. This is sufficiently general to describe unfold-fold transformations and many variants.

3.1 Transformation as Definition Construction

For the purposes of the formal definitions and results, transformation is viewed as the introduction of some *new* functions from a given set of definitions; so the transformation from a program consisting of a single function $\mathtt{f}\, x \triangleq e$, to a new version $\mathtt{f}\, x \triangleq e'$ will be formally represented by the derivation of a new function $\mathtt{g}\, x \triangleq e'\{\mathtt{g}/\mathtt{f}\}$, rather than by modification of the definition of $\mathtt{f}$. Correctness of the transformation now corresponds to validity the statement: $\mathtt{f} \cong \mathtt{g}$.

The reason why we adopt this representation of a transformation is because observational approximation and equivalence should, strictly speaking, be parameterised by the intended set of function definitions. Such explicit parameterisation is unwieldy, but we are able to avoid it by using a certain open-endendness property of the language, viz. that extending the language with new function symbols (ie. new definitions) *conservatively extends* operational approximation and equivalence. We can similarly "garbage collect" unreachable functions without affecting equivalences which do not pertain to those functions.

The implication of these properties is that we will be able to keep the set of definitions *implicit* in the theory of operational approximation and equivalence, by making sure that we never *change* the definition of a given function; transformation just adds new functions which extend the language.

3.1.1 Transformation

Sometimes it is also appropriate to consider weaker transformations, in which the transformation steps can increase the definedness of terms. Examples are unrestricted *unfolding*

[2] This will only be done informally because the various equivalences and preorderings will only be closed under proper substitutions.

in a strict language, or optimizations for strict versions of data structures such as $\mathtt{head}(\mathtt{cons}(x,y)) \sqsubseteq_{\sim} x$. Transformations using inequalities in this direction will be called *weak* transformations. Such transformations are common in strict languages eg. in Turchin's *supercompiler*[Tur86].

The following definition of transformation (weak transformation) will enable us to formulate the correctness problem and state some relatively standard partial correctness results.

Definition 3.1 (Transformation) (Weak Transformation)
A transformation (weak transformation, respectively) of a set of function definitions,

$$\{\mathtt{f}_i\, x_1 \ldots x_{\alpha i} \triangleq e_{\mathtt{f}_i}\}_{i\in I}$$

is given by a set of expressions

$$\{e_i\}_{i\in I}\,, \quad \mathrm{FV}(e_i) \subseteq \{x_1 \ldots x_{\alpha_i}\}\,,$$

such that $e_{\mathtt{f}_i} \cong e_i$ (respectively, $e_{\mathtt{f}_i} \sqsubseteq_{\sim} e_i$), together with a set of new functions (the transformed program)

$$\{\mathtt{g}_i\, x_1 \ldots x_{\alpha_i} \triangleq e_i\{\vec{\mathtt{g}}/\vec{\mathtt{f}}\}\}_{i\in I}$$

We say that there is a transformation (resp. weak transformation) from $\mathtt{f}_i$ to $\mathtt{g}_i$, and it will be taken that the function names $\mathtt{g}_i$ are fresh.

The form of the above definition of a transformation emphasises the transformational derivation as the two phase operation of: a local transformation (of the bodies of the functions) followed by definition construction. We reason about transformations which introduce new auxiliary functions by considering that all auxiliaries are introduced at the beginning of the transformation (cf. *virtual transformation sequences* [TS84]).

A higher-order variant of the classic Unfold-Fold transformation [BD77] can easily be recast as a transformation according to the above definition. Burstall and Darlington's original definition was for a first order functional language where functions are defined by pattern matching, and a call-by-name evaluation is assumed. The example is considered in detail in Section 5, where conditions guaranteeing total correctness are studied.

Definition 3.2 (Correctness)
A transformation (weak transformation) from $\mathtt{f}_i$ to $\mathtt{g}_i$ is correct (weakly correct) if $\mathtt{f}_i \cong \mathtt{g}_i$ ($\mathtt{f}_i \sqsubseteq_{\sim} \mathtt{g}_i$.)

An alternative way of expressing a transformation would be to say that there is a transformation from $\mathtt{f}\vec{x} \triangleq e_{\mathtt{f}}$ to new definition $\mathtt{g}\vec{x} \triangleq e_{\mathtt{g}}$ if and only if $e_{\mathtt{f}} \cong e_{\mathtt{g}}\{\mathtt{f}/\mathtt{g}\}$.

The definition of transformation is "complete" with respect to equivalence of definitions (cf. *second order replacement* [Kot80]), since if $\mathtt{f}\vec{x} \triangleq e$ is equivalent to some new[3] function $\mathtt{g}\vec{x} \triangleq e'$ it follows that $e \cong e'$, and that $e \cong e'\{\mathtt{f}/\mathtt{g}\}$; this latter equivalence defines a transformation from $\mathtt{f}$ to $\mathtt{g}$.

This shows that unfold-fold transformations are only a special case (of transformation), since the unfold-fold method is known to be incomplete in this sense (see [BK83][Kot80]-[Zhu94]). Of course, the problem we address in this paper is that both the general transformation, and the special case of unfold-fold are not *sound*.

[3] ie. f does not depend on g

3.2 Basic Correctness Results

Here we summarise some basic correctness and partial correctness results which relate to transformations. To summarise:

- Transformation is *partially* correct: the derived functions are less than the originals ($\mathtt{g}_i \sqsubseteq_{\sim} \mathtt{f}_i$).
- Weak transformations are not partially correct: the derived program can be unrelated (with respect to $\sqsubseteq_{\sim}$) to the original.
- As a corollary of the partial correctness property, "reversible" and "non-recursive" transformations are correct.

The first point follows easily from a least-fixed point property of the recursive equations, and is a fairly standard result: if $\mathtt{f}$ is transformed to $\mathtt{g}$ then it is easily verified that $\mathtt{f}$ "satisfies" $\mathtt{g}$'s defining equation. So $\mathtt{f}$ is a fixed-point of $\mathtt{g}$'s definition. But $\mathtt{g}$ is the *least* fixed-point, and hence $\mathtt{g} \sqsubseteq_{\sim} \mathtt{f}$. This corresponds to the (informal) partial correctness argument given by Burstall and Darlington [BD77] for unfold-fold transformation. A version of this result appears in [Cou79] where it is formulated for recursive program schemes (first order functional programs) but in a more general form.

The second point, a less well-known fact, is illustrated with a simple example:

Example 3.3 Let $\mathtt{f}\, x \triangleq \mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{f}\ \mathtt{false})$. Then since $\mathtt{f}\ \mathtt{true} \cong 1$ and $\mathtt{f}\ \mathtt{false} \sqsubseteq_{\sim} 2$ then

$$\mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{f}\ \mathtt{false}) \sqsubseteq_{\sim} \mathtt{if}\ x\ \mathtt{then}\ (\mathtt{f}\ \mathtt{true})\ \mathtt{else}\ 2$$

so there is a weak transformation from $\mathtt{f}$ to $\mathtt{g}$ where $\mathtt{g}\, x \triangleq \mathtt{if}\ x\ \mathtt{then}\ (\mathtt{g}\ \mathtt{true})\ \mathtt{else}\ 2$, but $\mathtt{f}$ and $\mathtt{g}$ are incomparable.

4 Correctness by Improvement

This section presents the main technical result of the paper. It says, roughly speaking, that a transformation from $\mathtt{f}\, x \triangleq e$, via equivalence $e \cong e'$ to $\mathtt{g}\, x \triangleq e'\{\mathtt{g}/\mathtt{f}\}$ is totally correct if e' is an *improvement* over e. The improvement relation is expressed in terms of the number of non-primitive function calls; e is improved by e' (written $e \rhd_{\sim} e'$) if in all closing contexts, $C[e]$ requires evaluation of no fewer non-primitive functions than $C[e']$.

4.1 Improvement

An intuition behind the use of improvement to obtain total correctness (at least for the case when the program computes a constant) is that improvement $e \rhd_{\sim} e'$ represents some "progress" towards convergence, and in the transformation to $\mathtt{g}\, x \triangleq e'\{\mathtt{g}/\mathtt{f}\}$, this progress is enjoyed on every recursive call.

The main problem with formulating an appropriate notion of improvement is that the language has a non-discrete data domain (ie lazy data and higher-order functions) as the results of programs. For this reason, as for operational approximation and equivalence, it is natural to define this as a contextual (pre)congruence.

Definition 4.1 *Closed expression e converges in n-steps to weak head normal form w, $e{\Downarrow^n}w$ if $e{\Downarrow}w$, and this computation requires n reductions of non-primitive functions.*

The operational semantics in the appendix makes this definition precise. A *reduction* of a non primitive function corresponds to replacing a call instance $\mathtt{f}\ e_1 \ldots e_{\alpha_\mathtt{f}}$ with the corresponding instance of the body of $\mathtt{f}$, $e_\mathtt{f}\{e_1 \cdots e_{\alpha_\mathtt{f}}/x_1 \ldots x_{\alpha\mathtt{f}}\}$. It will be convenient to adopt the following abbreviations:

- $e{\Downarrow^n} \stackrel{\text{def}}{=} \exists w.\, e{\Downarrow^n}w$
- $e{\Downarrow^{n\leq m}} \stackrel{\text{def}}{=} e{\Downarrow^n}\ \&\ n \leq m$
- $e{\Downarrow^{\leq m}} \stackrel{\text{def}}{=} \exists n.\, e{\Downarrow^{n\leq m}}$

Now improvement is defined in an analogous way to observational approximation:

Definition 4.2 *(Improvement) e* is improved by *e', $e \underset{\sim}{\rhd} e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed, if $C[e]{\Downarrow^n}$ then $C[e']{\Downarrow^{\leq n}}$.*

Some properties (laws) of the improvement relation are stated at the end of the section.

4.2 The Improvement Theorem

We are now able to state the main theorem, the proof of which is outlined in the appendix.

Theorem 4.3 *Given a set of function definitions,*

$$\{\mathtt{f}_i\, x_1 \ldots x_{\alpha i} \triangleq e_i\{\vec{\mathtt{f}}/\vec{y}\}\}_{i\in I}$$

and a set of expressions

$$\left\{e'_i\right\}_{i\in I} \quad \textit{where } \mathrm{FV}(e'_i) \subseteq \{x_1 \ldots x_{\alpha i}\} \cup \vec{y}$$

if $e_i\{\vec{\mathtt{f}}/\vec{y}\} \underset{\sim}{\rhd} e'_i\{\vec{\mathtt{f}}/\vec{y}\}$ then $\mathtt{f}_i \underset{\sim}{\rhd} \mathtt{g}_i$ where

$$\{\mathtt{g}_i\, x_1 \ldots x_{\alpha i} \triangleq e'_i\{\vec{\mathtt{g}}/\vec{y}\}\}_{i\in I}$$

Since improvement implies observational approximation, as a direct consequence of the Theorem we have a condition for correctness and weak correctness:

Corollary 4.4 *Given a transformation (resp. weak transformation) from $\{\mathtt{f}_i\, x_1 \ldots x_{\alpha i} \triangleq e_{\mathtt{f}_i}\}_{i\in I}$ to $\{\mathtt{g}_i\, x_1 \ldots x_{\alpha i} \triangleq e_i\{\vec{\mathtt{g}}/\vec{\mathtt{f}}\}\}_{i\in I}$, if $e_{\mathtt{f}_i} \underset{\sim}{\rhd} e_i$ then the transformation is correct (weakly correct) and moreover, $\mathtt{f}_i \underset{\sim}{\rhd} \mathtt{g}_i$.*

4.3 The Theory of Improvement

As is the case for operational approximation, to facilitate reasoning about the improvement relation it is essential to obtain a more tractable characterisation (than that provided by Definition 4.2)—in particular one which does not quantify over all contexts.

It turns out that $\underset{\sim}{\rhd}$ is an instance of the improvement theories previously investigated by the author [San91] (but quite independently of the transformation problem addressed in this paper). This earlier study provides some crucial technical background for "improvement" orderings in functional languages. Most importantly, [San91] provides a definition of *improvement simulation* as a generalisation of Abramsky's *applicative bisimulation*, and some results directly extending those of Howe [How89] which helps characterise when these relations are contextual congruences. We have used this to show that improvement can be expressed as an improvement simulation.

In the remainder of this section we describe this characterisation, and outline the associated proof technique (for improvement) which it provides.

Definition 4.5 *A relation $\mathcal{IR}$ on closed expressions is an* improvement simulation *if for all e, e', whenever $e\ \mathcal{IR}\ e'$, if $e{\Downarrow^n}w_1$ then $e'{\Downarrow^{\leq n}}w_2$ for 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) $w_1 \in$ *Closures*[4], $w_2 \in$ *Closures, 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, and yields a "similar" result, who's "components" are related by that improvement-simulation.

The key to reasoning about the improvement relation (and in particular, the key to proving the correctness of the improvement theorem) is the fact that $\underset{\sim}{\rhd}$, restricted to closed expressions, is itself an improvement simulation (and is in fact the *maximal* improvement simulation). Furthermore, improvement on open expressions can be characterised in terms of improvement on all closed instances. This is summarised in the following:

Lemma 4.6 (Improvement Context-Lemma) *For all e, e', $e \underset{\sim}{\rhd} e'$ if and only if there exists an improvement simulation $\mathcal{IR}$ 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 $e \underset{\sim}{\rhd} e'$ it is sufficient to find an improvement-simulation containing each closed instance of the pair.

In the appendix we also use a useful variant of this proof technique which follows the idea of "(bi)simulation up to" from CCS [Mil89].

Here are some example improvement laws which follow either directly from its definition, or from the improvement context lemma (by exhibiting appropriate improvement simulations):

Proposition 4.7

(i) $e \underset{\sim}{\rhd} e' \Rightarrow e \underset{\sim}{\sqsubset} e'$

(ii) $e \underset{\sim}{\rhd} e' \Rightarrow C[e] \underset{\sim}{\rhd} C[e']$

(iii) $e{\Downarrow}e' \Rightarrow e \underset{\sim}{\rhd} e'$

(iv) $\mathtt{f}\ x \underset{\sim}{\rhd} e$ *if* $\mathtt{f}\ x \triangleq e$

(v) $e \not\underset{\sim}{\rhd} \mathtt{f}\ x$ *if* $\mathtt{f}\ x \triangleq e$ *and* $\exists e'.\, \mathtt{f}\ e'{\Downarrow}$

[4] *Closures* is the set of function-valued results, ie. partially applied functions.

$$\textit{(vi)}^{5}\ R[\ \texttt{case}\ x\ \texttt{of}\ \ c_1(\vec{y}_1) : e_1\ \ \cdots\ \ c_n(\vec{y}_n) : e_n] \ \trianglerighteq\ \texttt{case}\ x\ \texttt{of}\ \ c_1(\vec{y}_1) : R[e_1]\ \ \cdots\ \ c_n(\vec{y}_n) : R[e_n]$$

5 Application to Unfold-Fold Transformations

The improvement theorem is directly applicable to the verification of the correctness of some transformation methods; as mentioned in the introduction, we have applied it to variants of the deforestation and supercompilation transformations (see [San94]).

In this section we consider a different kind of problem: transformation. The problem is different because, in general, unfold-fold is not correct. Our task is to use the improvement theorem to design a simple method for constraining the transformation process such that correctness is ensured.

Burstall and Darlington's original definition was for a first order functional language where functions are defined by pattern matching, and a call-by-name evaluation is assumed [BD77]. The following six rules were introduced for transforming recursion equations:

(i) *Definition* introduces a new recursion equation whose left-hand side is not an instance of the left-hand side of a previous definition;

(ii) *Instantiation* introduces a substitution instance of an existing equation;

(iii) *Unfolding* replaces an instance of a function-call by the appropriate instance of the body of the function;

(iv) *Folding* replaces an instance of a body of a function by a corresponding call;

(v) *Abstraction* replaces occurrences of some expressions by a variable bound by a *where*-clause;

(vi) *Laws* rewrite according to laws about primitive functions.

An important point, that lends significant power to the method, is that folding can make use of any definition of the function being folded — and this is typically an older definition. The original definition from [BD77] also allows unfolding using any earlier definition, but in practice this is never useful and will not be considered here.

Unfold-fold transformation carries over to higher-order functional languages essentially unchanged. To suit our particular language but without significant loss of generality we simplify the rules we will consider and the style in which they are applied.

- The definition rule above may be applied freely since it just introduces new functions (and conservatively extends the theories of operational approximation and improvement).
- Instantiation will not be used explicitly; the role of instantiation will be played by certain distribution laws for case-branches. In any case, unrestricted instantiation is problematic because it is not even locally equivalence preserving, since it can force premature evaluation of expressions (a problem noted in [Bir84], and addressed in some detail in [RFJ89]) and is better suited to a typed language in which one can ensure that the set of instances is exhaustive.
- Abstraction will not be used explicitly, since this can easily be represented by laws using case-expressions to represent the binding (which has the advantage of enabling us to deal implicitly with possible problems with bound variables which do not arise in Burstall and Darlington's language), or by using standard "lambda-lifting" techniques (equivalent to definition + folding).

[5] R ranges over *reduction contexts*; these are single-holed contexts whose hole is in a part of the term which will be evaluated next according to the operational semantics (see appendix for details).

5.1 Unfold-Fold Transformation Step

In our setting, an *unfold-fold transformation step* will consist of adding new definitions and (repeatedly) transforming the right-hand sides of some set of definitions $\{\mathtt{f}_i\vec{x}_i \triangleq e_i\}_{i\in I}$, according to unfolding, folding and rewriting laws, to obtain a set of expressions $\{e'_i\}_{i\in I}$. Using these expressions a set of new function definitions is constructed, namely

$$\{\mathtt{f}'_i\vec{x}_i \triangleq e'_i\{\vec{\mathtt{f}}'/\vec{\mathtt{f}}\}\}_{i\in I}.$$

Henceforth, the terms *unfolding* and *folding* refer to local transformations on expressions, and not to operations which in themselves give rise to new definitions. (It is the *unfold-fold transformation step* which constructs new definitions.)

Given definitions $\mathtt{f}_i\vec{x} \triangleq e_i$, the unfold and fold rewrites are just instances of the congruences $\mathtt{f}_i\vec{x} \cong e_i$ (although this law does not hold in a call-by-value language under arbitrary substitutions, so for strict languages further restrictions must be applied). We take the laws to be operational equivalences by definition. So we see that an unfold-fold transformation step is now a transformation according to definition 3.1, and we obtain the standard partial correctness result that the new program may be less defined than the original (see [Kot78,Kot85][Cou79]).

5.2 Correctness by Improvement in Unfold-Fold

If we restrict the application of the laws so that they are improvement steps (as they usually are in practice), then since the unfolding step is an improvement, transformation steps not involving folds are totally correct, and the results are improvements. This follows directly from Corollary 4.4. (In fact it is not too difficult to show that any transformation without folding is correct, providing the laws only involve primitive functions ([Cou86][Zhu94].)

The problem is that no fold step, viewed in isolation, is an improvement[6]. The key to guaranteeing correctness is to ensure that we pay for each fold step at some point, thus maintaining overall improvement.

[6] Except for the trivial case when the function being folded-against is everywhere undefined.

5.3 The Tick Algebra

To enable the required improvement condition to be maintained step-wise through the transformation, we will extend the unfold-fold process with a single annotation, $\checkmark$, "tick". A tick represents a single computation step, so we can think of it as just an identity function $\checkmark x \triangleq x$. From the point of view of observational equivalence it is just an annotation, since $\checkmark e \cong e$; but from the point of view of improvement it is more significant. In particular, $\checkmark e \mathrel{\underset{\sim}{\rhd}} e$ but $e \not\mathrel{\underset{\sim}{\rhd}} \checkmark e$ (except if all closed instances of e diverge).

The technique for ensuring correctness of unfold-fold will be to use a tick to pay for a fold step, paid for by a "nearby" unfold. With respect to a definition $\mathtt{f}x \triangleq e$, the unfolding and folding steps correspond to instances of the law $\mathtt{f}\,x \cong e$. In terms of improvement, we have the following laws (which follow easily from the improvement context-lemma) relating a function and its body:

$$\begin{array}{lcl} \mathtt{f}\,x & \underset{\sim}{\rhd} & \checkmark e \\ \checkmark e & \underset{\sim}{\rhd} & \mathtt{f}\,x \end{array}$$

The idea will be to use (substitution instances of) these laws, in place of unfolding and folding, respectively.

The *tick algebra* is a collection of laws for propagating ticks around an expression whilst preserving or increasing improvement. Figure 1 gives some basic laws for $\checkmark$ which will augment the transformation rules. Let $\mathrel{\underset{\sim}{\lhd\rhd}}$ denote the "cost"-equivalence associated to improvement:

$$e_1 \mathrel{\underset{\sim}{\lhd\rhd}} e_2 \stackrel{\text{def}}{=} e_1 \mathrel{\underset{\sim}{\rhd}} e_2 \;\&\; e_2 \mathrel{\underset{\sim}{\rhd}} e_1.$$

We also need a distribution law for nested case-expressions (a symmetric version of Proposition 4.7(vi). The the basic reduction steps of the operational semantics (see Appendix, Figure 3) are also included in the improvement relation, and are therefore useful. In the laws, R ranges over (possibly open) reduction contexts. These are single-holed contexts whose hole is in a part of the term which will be evaluated next according to the operational semantics (see appendix for details).

The laws are straightforward to prove using the improvement context-lemma (the details are omitted). But it is not intended (or expected) that one should (need to) prove tick-laws "on the fly". The method is intended to be viewed as completely syntactic—given a "reasonable" set of tick laws.

5.4 "Improved" Unfold-Fold Transformations

We show by two small examples how the tick algebra can be used to maintain improvement throughout the steps of a transformation, thereby guaranteeing total correctness and improvement.

Example 5.1 In Figure 2 we give a standard unfold-fold example, but now locally maintaining the improvement relation at each step of the transformation by introducing ticks at unfold steps, and propagating these, via the tick laws, to the fold site. The example ensures, by construction, that the derived program $\mathtt{sumsq}'$ is an improvement over the original (in addition to being equivalent). Note that in this particular case, since the improvement steps above are all represented by cost-equivalences, we get a good picture of the *degree* of improvement: namely, one function call is saved for each call to $\mathtt{sumsqs}'$, plus one more when the argument is nil.

Example 5.2 To take a smaller (but less standard) example, consider the usual Y-combinator of the lambda-calculus (in head normal form) given by

$$\lambda h.h((\lambda x.h(xx))\lambda x.h(xx)).$$

A direct translation of this term into our language would be the expression $\mathtt{Y}$, where

$$\begin{array}{rcl} \mathtt{Y}\,h & \triangleq & h\,(\mathtt{D}\,h\,(\mathtt{D}\,h)) \\ \mathtt{D}\,h\,x & \triangleq & h\,(x\,x) \end{array}$$

Now we transform the definition of $\mathtt{Y}$ to obtain a direct recursive version, by unfolding the call to $\mathtt{D}$ then immediately folding against $\mathtt{Y}$:

$$\begin{array}{rcll} \mathtt{Y}\,h & \triangleq & h\,(\mathtt{D}\,h\,(\mathtt{D}\,h)) & \\ & \underset{\sim}{\rhd} & h\,\checkmark(h\,(\mathtt{D}\,h(\mathtt{D}\,h))) & \text{(unfold } \mathtt{D}\text{)} \\ & \underset{\sim}{\rhd} & h\,(\mathtt{Y}\,h) & \text{(fold with } \mathtt{Y}\text{)} \end{array}$$

thus obtaining a new definition $\mathtt{Y}'h \triangleq h\,(\mathtt{Y}'h)$. This shows that $\mathtt{Y}$ is equivalent to $\mathtt{Y}'$, and $\mathtt{Y} \mathrel{\underset{\sim}{\rhd}} \mathtt{Y}'$. (In fact since the steps of the unfold fold are all cost equivalences, a small extension of the main theorem gives us that $\mathtt{Y} \mathrel{\underset{\sim}{\lhd\rhd}} \mathtt{Y}'$.)

Space does not permit an exposition of the pragmatics of the method but a few remarks concerning extensions and limitations are appropriate:

- Transformations in which folds steps occur before any unfold steps can be allowed by "borrowing" ticks and "paying them back" later. This corresponds to use of the law $\checkmark e_1 \mathrel{\underset{\sim}{\rhd}} \checkmark e_2 \Rightarrow e_1 \mathrel{\underset{\sim}{\rhd}} e_2$. We can incorporate this idea into a stepwise transformation (which uses only axioms, congruence properties and transitivity) by introducing *negative* ticks.
- Derived functions are improvements on the originals, so we can allow folding against functions obtained from previous transformation steps. For example, if a transformation step from $\mathtt{f}$ derives a function $\mathtt{f}'$ then $\mathtt{f} \mathrel{\underset{\sim}{\rhd}} \mathtt{f}'$. So a subsequent transformation step can include a generalised fold, in which an instance of the body of $\mathtt{f}$ (suitably "ticked") can be replaced by a call to $\mathtt{f}'$.
- Some transformations involving lazy data-structures need to be phrased slightly differently (from traditional unfold-fold derivations) for the method to work. Problems can arise because ticks cannot propagate across lazy constructors. These cases are easily handled by postponing the introduction of new definitions until the point at which a potential fold is discovered. This is analogous to the way transformations are performed using *expression procedures* [Sch81] (discussed in the next section).

6 Related Work

Although the topics of program transformation, including partial evaluation, have been active research topics in functional programming for the last decade and more, the problem of correctness has received surprisingly little attention.

$$\bullet\ {}^{\surd}e \mathrel{\underset{\sim}{\rhd}} e \qquad \bullet\ \frac{{}^{\surd}e_1 \mathrel{\underset{\sim}{\rhd}} {}^{\surd}e_2}{e_1 \mathrel{\underset{\sim}{\rhd}} e_2} \qquad \bullet\ R[{}^{\surd}e] \mathrel{\underset{\sim}{\lhd\rhd}} {}^{\surd}R[e]$$

$$\bullet\ \frac{\mathtt{f}\,\vec{x} \triangleq e}{\mathtt{f}\,\vec{x} \mathrel{\underset{\sim}{\lhd\rhd}} {}^{\surd}e} \qquad \bullet\ {}^{\surd}p(e_1 \ldots e_n) \mathrel{\underset{\sim}{\lhd\rhd}} p(e_1 \ldots {}^{\surd}e_i \ldots e_n)$$

$$\bullet\ {}^{\surd}\,\mathtt{case}\ e\ \mathtt{of}\ c_1(\vec{x}_1) : e_1 \ldots c_n(\vec{x}_n) : e_n \mathrel{\underset{\sim}{\lhd\rhd}} \mathtt{case}\ e\ \mathtt{of}\ c_1(\vec{x}_1) : {}^{\surd}e_1 \ldots c_n(\vec{x}_n) : {}^{\surd}e_n$$

Figure 1: Tick Laws

Consider the following definitions:

$$\begin{aligned}
\mathtt{sum}\,xs &\triangleq \mathtt{case}\ xs\ \mathtt{of}\\
&\quad\ nil : \mathtt{0}\\
&\quad\ cons(y, ys) : y + \mathtt{sum}\,ys\\
\mathtt{sq}\,x &\triangleq x * x\\
\mathtt{map}\,f\,xs &\triangleq \mathtt{case}\ xs\ \mathtt{of}\\
&\quad\ nil : \mathtt{nil}\\
&\quad\ cons(y, ys) : cons((f\ y), \mathtt{map}\,f\ ys)
\end{aligned}$$

Now consider the following transformation of the function which takes the sum of the squares of a list:

$$\mathtt{sumsqs}\ xs \triangleq \mathtt{sum}(\mathtt{map}\ \mathtt{sq}\ xs).$$

Transforming the right-hand side we obtain:

$$\begin{aligned}
\mathtt{sum}(\mathtt{map}\,\mathtt{sq}\,xs) &\mathrel{\underset{\sim}{\lhd\rhd}} \mathtt{sum}^{\surd}\left(\begin{array}{l}\mathtt{case}\ xs\ \mathtt{of}\\ \quad nil : \mathtt{nil}\\ \quad cons(y, ys) : \mathtt{cons}((\mathtt{sq}\,y), \mathtt{map}\,\mathtt{sq}\,ys)\end{array}\right) && \text{(unf. } \mathtt{map}\text{)}\\
&\mathrel{\underset{\sim}{\lhd\rhd}} {}^{\surd}\,\mathtt{case}\ {}^{\surd}\left(\begin{array}{l}\mathtt{case}\ xs\ \mathtt{of}\\ \quad nil : \mathtt{nil}\\ \quad cons(y, ys) : \mathtt{cons}((\mathtt{sq}\,y), \mathtt{map}\,\mathtt{sq}\,ys)\end{array}\right)\ \mathtt{of} && \text{(unf. } \mathtt{sum}\text{)}\\
&\qquad nil : 0\\
&\qquad cons(x, xs) : x + \mathtt{sum}\,xs\\
&\mathrel{\underset{\sim}{\lhd\rhd}} {}^{\surd\surd}\,\mathtt{case}\ xs\ \mathtt{of} && (^{\surd}\text{/case-laws})\\
&\qquad nil : 0\\
&\qquad cons(y, ys) : (\mathtt{sq}\,y) + \mathtt{sum}(\mathtt{map}\,\mathtt{sq}\,ys)\\
&\mathrel{\underset{\sim}{\lhd\rhd}} {}^{\surd}\,\mathtt{case}\ xs\ \mathtt{of} && (^{\surd}\text{-laws})\\
&\qquad nil : {}^{\surd}0\\
&\qquad cons(y, ys) : (\mathtt{sq}\,y) + {}^{\surd}\mathtt{sum}(\mathtt{map}\,\mathtt{sq}\,ys)\\
&\mathrel{\underset{\sim}{\lhd\rhd}} {}^{\surd}\,\mathtt{case}\ xs\ \mathtt{of} && \text{(fold } \mathtt{sumsqs}\text{)}\\
&\qquad nil : {}^{\surd}0\\
&\qquad cons(y, ys) : (\mathtt{sq}\,y) + \mathtt{sumsq}\ ys
\end{aligned}$$

Finally, after eliminating the ticks we construct the new definition:

$$\begin{aligned}
\mathtt{sumsq}'\ xs \triangleq\ & \mathtt{case}\ xs\ \mathtt{of}\\
& nil : \mathtt{0}\\
& cons(y, ys) : (\mathtt{sq}\,y) + \mathtt{sumsq}'\,ys
\end{aligned}$$

Figure 2: Improved `sumsqs` transformation

Our definition of transformation and its partial (but not total) correctness property is what Kott named "second order replacement" [Kot80] (see also [Sch80][Theorem 2.7]). Kott's motivation for the definition was that the less-general unfold-fold method, even ignoring the correctness issue, has strictly limited power (see also [BK83] [Zhu94]). He did not however consider solutions to the correctness problem for this more general class of transformations. Some principles for obtaining total correctness follow from the standard partial correctness results; for example if one can prove the new program is *total* [MW79], or more generally if the new equations have a "unique" solution [Cou79]. Correctness also follows (from partial correctness) if the transformation is *reversible*. Courcelle [Cou86] proposed simple constraints on unfold-fold transformations to guarantee reversibility. This method is rather limited in power since it cannot introduce new recursive structure. Reversible transformations have also been studied in logic programming (see eg. [Mah89] and [GS91]).

Expression Procedures Considering more specific transformation methods, of particular note is Scherlis' transformation method based on *expression procedures* [Sch80]. This method is less general (ie. can be simulated by) the unfold-fold method, but has the distinction that it preserves total correctness without need for any constraints. The method is proved correct (for a strict first order language) using a notion of *progressiveness*, based on reduction orderings (related ideas are used in [Red89] for the synthesis of Noetherian rewrite rules from equational specifications). There are some connections between progressiveness and improvement [San94] which suggest there is potential for applying the improvement methods developed here to the problem of generalising Scherlis' transformation and proving it correct for lazy and higher order languages.

Kott: Unfold-Fold With respect to total correctness of unfold-fold transformations on first-order programs, Kott [Kot78] [Kot85] gives some technical results relating the number of unfolding steps to the number of folds in restricted form of transformation of the body of a single recursive function, using a sequence of unfoldings, laws and foldings (in that order). It is often stated, incorrectly, that the main result of the paper implies that: *"... an unfold-fold derivation is correct if there are no more folds than there are unfolds at any stage of a transformation."* Our solution based on the tick-algebra certainly has this flavour, but without further qualification this statement is false (see the example in the introduction). In fact, Kott's main results qualify a statement of this form by modifying the very notion of correctness in a way which is dependent on each transformation history. From a given transformation, Kott effectively constructs an additional "strictness" law which must hold for the above correctness statement to be true. In practice this is unsatisfactory because the extra law is often *not* true in the intended model, even when the transformation is correct.

In comparison, in addition to the fact that we can handle higher-order functions and much less restricted transformations, our approach to obtaining unfold-fold transformations is technically simpler, and more practical since it can guide the transformation rather than being a post-hoc verification. Given that there are obvious relationships between strictness laws and tick-propagation laws, we conjecture that it will be possible to prove that our method allows us to verify the correctness of strictly more transformations than Kott's (as we have observed in practice).

Replacement and Unfold-fold in Logic Programs In logic programming there are analogous problems with the operation of *replacement* of some subgoals in a clause by a logical equivalent, see eg. [BCE92b]. In Bossi *et al*'s work, a model-theoretic definition of *semantic delay*, somewhat analogous to our improvement relation, is used as part of a replacement condition guaranteeing correctness for a variety of models, although to our knowledge it has not been used to construct simple syntactic methods to constrain unfold-fold transformation (although the method is able to justify some fold steps "in isolation" [BCE92a]) or to justify other automatic transformation methods.

For the most part, the operation of replacement has been studied only as a subsidiary tool in unfold-fold transformation [TS84,GS91,KF86,PP93]. Methods for constraining unfold-fold transformations (of logic programs) to guarantee correctness under various semantic interpretations have been widely studied – see Tamaki and Sato's work [TS84] and its many derivatives eg. [Sek93][Sat90][PP91]. See [Amt92] for a technical framework in which many of these conditions can be expressed, and [PP93] for an overview of the methods. There are (informal) similarities between our method of constraining unfold-fold using ticks and the method of *counters* of Kanamouri and Fujita[7] [KF86] (a direct generalisation of Tamaki and Sato's basic "foldable"-annotations).

A more technical comparison and criticism of related work is contained in the full version of the paper [San94].

Further Work

- Application of the improvement theorem to justify other transformation methods (eg. more realistic forms of partial evaluation), and in particular use of the corresponding call-by-value improvement theorem for call-by-value transformations.
- Investigating the use of correct unfold-fold transformations as a proof technique.
- Investigate the completeness of the improvement theorem with respect to improvement.

Finally, a problem left open in [San91]: is there a tractable *call-by-need* theory of improvement? A "reasonable" call-by-need theory is a prerequisite to answering the following natural question: does the improvement theorem hold for the corresponding improvement theory?

Acknowledgements

This work began while the author was funded by the DART project (Danish Research Council), and the Department of Computer Science at Copenhagen University. The author is partially supported by ESPRIT BRA "Coordination". The

[7] Given the way we use the general improvement theorem constrain the unfold-fold method, and given the above analogies, it seems reasonable to ask whether Bossi *et al*'s replacement conditions can be used to justify a method for constraining unfold-fold such as Kanamouri and Fujita's.

author gratefully acknowledges Torben Amtoft for discussions on the problem early in the development of this work, and his help in obtaining some of the references. Thanks to the "Topps" group at DIKU, and in particular to Robert Glück, Morten Heine Sørensen, Neil Jones, Kristian Nielsen, Bob Paige and Tom Reps for numerous discussions on the subject of program transformation, and comments on earlier drafts. Thanks to Phil Wadler for some helpful comments, and to the referees for suggesting a number clarifications.

References

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

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

[BCE92a] A. Bossi, N. Cocco, and S. Etalle. On safe folding. In *PLILP '92*, volume 631 of *LNCS*, pages 172–186. Springer-Verlag, 1992.

[BCE92b] 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.

[Bir84] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[BK83] G. Boudol and L. Kott. Recursion induction principle revisited. *Theoretical Computer Science*, 22:135–173, 1983.

[CK93] C. Consel and S. Khoo. On-line and off-line partial evaluation: Semantic specification and correctness proofs. Technical report, Yale University, April 1993.

[Cou79] B. Courcell. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[Cou86] B. Courcelle. Equivalences and transformations of regular systems—applications to recursive program schemes and grammars. *Theoretical Computer Science*, 42:1–122, 1986.

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

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

[GS91] P. Gardner and J. Shepherdson. Unfold/fold transformations of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. 1991.

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

[JGS93] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[KF86] T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. Technical Report ICOT Tech Report TR-179, Mitsubishi Electric Corp., 1986.

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

[Kot80] L. Kott. A system for proving equivalences of recursive programs. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 63–69. Springer-Verlag, 1980.

[Kot85] L. Kott. Unfold/fold transformations. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 412–433. CUP, 1985.

[Mah89] M. Maher. Correctness of a logic program transformation system. Technical report, IBM - Watson Research Center, 1987 (revised 1989).

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

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

[Pal93] J. Palsberg. Correctness of binding time analysis. *Journal of Functional Programming*, 3(3), 1993.

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

[PP91] M Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of the symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM press, SIGPLAN notices, 26(9), September 1991.

[PP93] A. Pettorossi and M Proietti. Transformation of logic programs: Foundations and techniques. Technical Report R 369, CNR Istituto di Analisi dei Sistemi ed Informatica, Rome, November 1993.

[Red89] U. Reddy. Rewriting techniques for program synthesis. In *Rewriting Techniques and Applications*, number 355 in LNCS, pages 388–403. Springer-Verlag, 1989.

[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.

[San93] D. Sands. A naïve time analysis and its theory of cost equivalence. TOPPS report D-173, DIKU, 1993. To appear: Journal of Logic and Computation.

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

[Sat90] T. Sato. An equivalence preserving first order unfold/fold transformation system. In *2nd International Conference on Algebraic and Logic Programming*, volume 462 of *LNCS*, pages 175–188. Springer-Verlag, 1990.

[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.

[Sek93] H. Seki. Unfold/fold transformation of general logic programs for the well-founded semantics. *J. Logic Programming*, 16:5–23, 1993.

[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.

[Wan93] M. Wand. Specifying the correctness of binding time analysis. *Journal of Functional Programming*, 3(3), 1993.

[Zhu94] Hong Zhu. How powerful are folding/unfolding transformations? *J. Functional Programing*, 4(1):89–112, January 1994.

A Appendix

In this appendix we provide some details not included in the main text, leading to an outline of the proof of the improvement theorem. Omitted proofs can be found in [San94].

Operational Semantics

The semantics for the language defined in section 2 is presented in terms of a single step reduction relation using the notion of a reduction context [FFK87]. Reduction contexts, ranged over by R, are contexts containing a single hole which is used to identify the next expression to be evaluated (reduced).

Definition A.1 *A reduction context R is given inductively by the following grammar*

$$\begin{array}{rcl} R & = & [\,] \mid R\ e \mid R@e \mid w@R \\ & \mid & \texttt{case}\ R\ \texttt{of}\ c_1(\vec{x}_1) : e_1 \ldots c_n(\vec{x}_n) : e_n \\ & \mid & p(\vec{c}, R, \vec{e}) \end{array}$$

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 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 A.2 *One-step reduction $\mapsto$ is the least relation on closed expressions satisfying the rules given in Figure 3.*

In each rule of the form $R[e] \mapsto R[e']$ in Figure 3, the expression e is referred to as a *redex*. 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 R and a redex e' such that $e_1 = R[e']$. Recall that the weak head normal forms are constructor expressions and partially applied functions $\texttt{f}\ e_1 \ldots e_k$, $k < \alpha(\texttt{f})$.

Definition A.3 *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 $\mapsto$). Further, if this evaluation requires n rewrites using function-application rule $(*)$ then we write $e{\Downarrow^n}w$.*

A Characterisation of Improvement

We give a more convenient definition of improvement simulation in terms of the following operation:

Definition A.4 *Given a relation R on closed expressions, we define $R^\dagger$ to be the least relation on weak head normal forms such that*

- $c(e_1 \ldots e_n)\ R^\dagger\ c(e'_1 \ldots e'_n)$ *if* $e_i\ R\ e'_i$, $i \in \{1 \ldots n\}$
- $(\texttt{f}\ e_1 \ldots e_i)\ R^\dagger\ (\texttt{g}\ e'_1 \ldots e'_j)$ *if for all closed e,* $(\texttt{f}\ e_1 \ldots e_i\ e)\ R\ (\texttt{g}\ e'_1 \ldots e'_j\ e)$

Definition A.5 *A relation $\mathcal{IR}$ on closed expressions is an* improvement simulation *if for all e_1, e_2, whenever $e_1\ \mathcal{IR}\ e_2$, if $e_1{\Downarrow^n}w_1$ then $e_2{\Downarrow^{\leq n}}w_2$ for some w_2 such that $w_1\ \mathcal{IR}^\dagger\ w_2$.*

Lemma A.6 *(Lemma 4.6, restated)*
For all e, e', $e \gtrsim e'$ if and only if there exists an improvement simulation $\mathcal{IR}$ such that for all closing substitutions σ, $e\sigma\ \mathcal{IR}\ e'\sigma$.

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

Figure 3: One-step reduction rules

PROOF. The main part of the proof ($\Leftarrow$) follows from [San91] (Theorem 2.14), after recasting the language into the appropriate syntactic form. The ($\Rightarrow$) direction follows from the fact that the language has sufficiently many destructors – although the details are somewhat tedious (see the corresponding proof about a similar relation in [San93][Appendix B]). □

Definition A.7 *A relation*[8] $\mathcal{IR}$ *on closed expressions is an* improvement simulation up to improvement *(abbreviated to an* i-simulation*) if for all* e_1, e_2, *whenever* $e_1\ \mathcal{IR}\ e_2$, *if* $e_1 \Downarrow^n w_1$ *then* $e_2 \Downarrow^{\leq n} w_2$ *for some* w_2 *such that* $w_1\ \underset{\sim}{\rhd}; R^\dagger; \underset{\sim}{\rhd}\ w_2$.

Proposition A.8 *To prove* $e_1\ \underset{\sim}{\rhd}\ e_2$ *it is sufficient to find an i-simulation which contains each closed instance of the pair.*

Proof of the Improvement Theorem

Now we can sketch the correctness proof of the main theorem. In what follows let $\vec{\mathtt{f}} \equiv \mathtt{f}_1 \ldots \mathtt{f}_n$ and $\vec{\mathtt{g}} \equiv \mathtt{g}_1 \ldots \mathtt{g}_n$ be as in the statement of Theorem 4.3. Under the conditions of the theorem we must show that $\vec{\mathtt{f}}\ \underset{\sim}{\rhd}\ \vec{\mathtt{g}}$. We do this by constructing a relation that contains $(\mathtt{f}_i, \mathtt{g}_i)$, and which we show to be an i-simulation up-to.

Definition A.9 *Define relation* $\mathcal{IS}$ *on closed expressions by*

$$\mathcal{IS} = \left\{ (e\{\vec{\mathtt{f}}/\vec{y}\}, e\{\vec{\mathtt{g}}/\vec{y}\}) \mid \mathrm{FV}(e) \subseteq \vec{y} \right\}$$

To show that $\mathcal{IS}$ is an improvement simulation up-to improvement we will need some technical results connecting $\mathcal{IS}$ and the one step reduction relation $\mapsto$.

Recall that improvement is measured in terms of the number of steps of the function application rule. If $e \mapsto e'$ according to this rule then we write $e \overset{\beta}{\mapsto} e'$. If $e \mapsto e'$ by any other rule (the redexes are disjoint, so only one rule can apply) then we write $e \overset{\delta}{\mapsto} e'$. In what follows let $\mapsto\!\!\!\!\mapsto$ denote the transitive reflexive closure of $\overset{\delta}{\mapsto}$. The following proposition details the interactions between the one-step reduction and the relation $\mathcal{IS}$.

Proposition A.10 *(i) If* $e_1\ \mathcal{IS}\ e_2$ *then* $e_1 \in$ WHNF *if and only if* $e_2 \in$ WHNF.

(ii) If $e_1\ \mathcal{IS}\ e_2$ *and* $e_1 \overset{\delta}{\mapsto} e_1'$ *then* $e_2 \overset{\delta}{\mapsto} e_2'$ *and* $e_1'\ \mathcal{IS}\ e_2'$.

(iii) If $e_1\ \mathcal{IS}\ e_2$ *and* $e_1 \overset{\beta}{\mapsto} e_1'$ *then* $e_2 \overset{\beta}{\mapsto} e_2'$ *and* $e_1'(\underset{\sim}{\rhd}; \mathcal{IS})e_2'$.

[8] If S and T are relations then $S; T$ denotes the relational-composition given by: $a\ (S; T)\ b$ if and only if $a\ S\ a'$ and $a'\ T\ b$ for some a'.

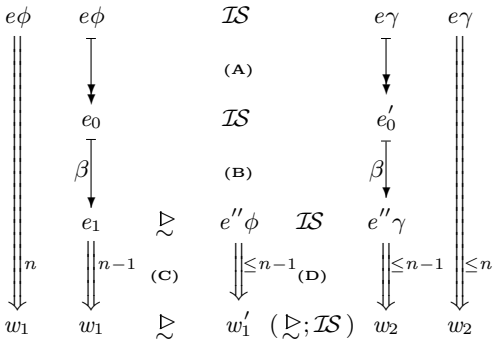
Now we can outline the proof of Theorem 4.3:

PROOF. **(Theorem 4.3)** Assume $e_1\ \mathcal{IS}\ e_2$ and $e_1 \Downarrow^n w_1$. We show by complete induction on n that $e_2 \Downarrow^{\leq n} w_2$ for some w_2 such that $w_1(\underset{\sim}{\rhd}; \mathcal{IS})w_2$. By simple properties of $\underset{\sim}{\rhd}$ and $\mathcal{IS}$ we can see that this is sufficient to show that $\mathcal{IS}$ is an i-simulation up-to, and hence that $\mathtt{f}_i\ \underset{\sim}{\rhd}\ \mathtt{g}_i$.

Let $\phi = \{\vec{\mathtt{f}}/\vec{y}\}$ and $\gamma = \{\vec{\mathtt{g}}/\vec{y}\}$. We have, by definition of $\mathcal{IS}$, an expression e such that $e\phi \equiv e_1$ and $e\gamma \equiv e_2$.

Base ($n = 0$)**:** Then $e\phi \mapsto\!\!\!\!\mapsto w_1$. By Proposition A.10(ii), $e\phi \mapsto\!\!\!\!\mapsto e'$ for some e' such that $w_1\ \mathcal{IS}\ e'$. By Proposition A.10(i) it follows that $e' \in$ WHNF and hence $e' \equiv w_2$ and we are done.

Induction ($n \geq 1$)**:** Since $e\phi \Downarrow^{n \geq 1}$ then for some e_0, e_1, $e\phi \mapsto\!\!\!\!\mapsto e_0$, $e_0 \overset{\beta}{\mapsto} e_1$ and $e_1 \Downarrow^{n-1} w_1$. We summarise the remaining argument with the following commuting diagram, in which we complete the squares (A)–(D) from top left to bottom right:



(A) by Proposition A.10(ii);

(B) by Proposition A.10(iii);

(C) since improvement is an improvement simulation;

(D) by the induction hypothesis.

□