On the Termination of Higher-Order Positive Supercompilation

G.W. Hamilton

School of Computing and Lero
Dublin City University
Ireland
hamilton@computing.dcu.ie

Abstract

The verification of program transformation systems requires that we prove their termination. For positive supercompilation, ensuring termination requires the memoisation of expressions which are subsequently used to determine when to perform generalization and folding. For a first-order language, it is sufficient to memoise only those expressions immediately prior to a function unfolding step. However, for a higher-order language, this is not sufficient to ensure termination, so more expressions need to be memoised. Determining which additional expressions to memoise can greatly affect the results obtained. Memoising too many expressions requires a lot more expensive checking for the possibility of generalization or folding; more new functions will also be created and generalization will be performed more often, resulting in less improved residual programs. We would therefore like to memoise as few expressions as possible while still ensuring termination. In this paper, we describe a simple pre-processing step which can be applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there must be function unfolding. We prove, for programs that have been pre-processed in this way, that it is only necessary to memoise expressions immediately before function unfolding to ensure termination, and we demonstrate this on a number of tricky examples.

1 Introduction

Supercompilation is a program transformation technique for functional languages which can be used for program specialization and for the removal of intermediate data structures. Supercompilation was originally devised by Turchin in what was then the USSR in the early 1970s, but did not become widely known to the outside world until over a decade later. One reason for this delay was that the work was originally published in Russian in journals which were not accessible to the outside world; it was eventually published in mainstream journals much later [23, 24]. Another possible reason why supercompilation did not become more widely known much earlier is that it was originally formulated in the language Refal, which is rather unconventional in its use of a complex pattern matching algorithm. This meant that Refal programs were hard to understand, and describing transformations making use of this complex pattern matching algorithm made the descriptions quite inaccessible. This problem was overcome by the development of *positive supercompilation* [19, 22], which is defined over a more familiar functional language.

Ensuring the termination of positive supercompilation requires the memoisation of expressions, and then using these memoised expressions to determine when to perform generalization and folding. Positive supercompilation was originally formulated for a first-order language, so it was sufficient to memoise only the expressions immediately prior to function unfolding to ensure termination since in any potentially infinite sequence of transformation steps there must

be an unfolding. However, this is not sufficient to ensure termination when transforming a higher-order language. For example, consider the following program:

Example 1. $(\lambda x \to x\ x)\ (\lambda x \to x\ x)$

When this program is transformed there will be a potentially infinite sequence of transformation steps without any unfolding. Although this expression would not be accepted by most type checkers, there are also many examples of expressions which would be accepted by a type checker and which will have a potentially infinite sequence of transformation steps without any unfolding. For example, consider the following program:

Example 2. **data** $D = F\ (D \to D)$

$$(\lambda f \to f\ (F\ (\lambda x \to f\ x\ x))\ (F\ (\lambda x \to f\ x\ x)))\ (\lambda y \to \textbf{case}\ y\ \textbf{of}\ F\ g \to g)$$

This program will also produce a potentially infinite sequence of transformation steps without any unfolding when transformed.

To avoid this potential non-termination, some formulations of positive supercompilation for a higher-order language memoise *all* expressions [17, 2], or at least a substantial subset of them [9, 10, 11]. Memoising too many expressions requires a lot more expensive checking for the possibility of generalization or folding. Also, more new functions will be created and generalization will be performed more often, resulting in less improved residual programs.

In this paper, we describe a simple pre-processing step which can be applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there must be an unfolding. This involves introducing names for some anonymous functions (and possibly also performing λ-lifting [7]) to ensure that only memoising expressions immediately preceding an unfold step is sufficient to ensure termination of the transformation. This pre-processing step would transform the program in Example 1 to the following:

$$f\ f\ \textbf{where}\ f = \lambda x \to x\ x$$

Thus, any potentially infinite sequence of transformation steps would have to include the unfolding of f. Applying the pre-processing transformation to the program in Example 2 would give the following:

$$\begin{array}{l} f_1\ f_2 \\ \textbf{where} \\ f_1 \ = \ \lambda f \to f\ (F\ (f_3\ f))\ (F\ (f_3\ f)) \\ f_2 \ = \ \lambda y \to \textbf{case}\ y\ \textbf{of}\ F\ g \to g \\ f_3 \ = \ \lambda f \to \lambda x \to f\ x\ x \end{array}$$

Thus, any potentially infinite sequence of transformation steps would have to include the unfolding of f_2 and f_3. The new functions are introduced sparingly, so we argue that there will not be a large overhead required for the additional memoisation and comparison of expressions prior to the unfolding of these functions, and that better residual programs will be produced as a result.

The remainder of this paper is structured as follows. In Section 2, we describe the higher-order language over which the transformations are defined. In Section 3, we give our own formulation of the positive supercompilation algorithm on this language. In Section 4, we consider the different situations in which this transformation may not terminate and give examples. In Section 5, we present our pre-processing step to transform higher-order programs into a form for which positive supercompilation will be guaranteed to terminate and prove that this is the case. Section 6 concludes and considers related work.

2 Language

In this section, we describe the higher-order functional language which will be used throughout this paper. The syntax of this language is given in Fig. 1.

prog	::=	e_0 **where** $f_1 = e_1 \ldots f_n = e_n$	Program
e	::=	x	Variable
	|	$c\ e_1 \ldots e_n$	Constructor
	|	$\lambda x\ \rightarrow e$	λ-Abstraction
	|	f	Function Call
	|	$e_0\ e_1$	Application
	|	**case** e_0 **of** $p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$	Case Expression
	|	**let** $x = e_0$ **in** e_1	Let Expression
p	::=	$c\ x_1 \ldots x_n$	Pattern

Figure 1: Language Syntax

The intended operational semantics of the language is normal order reduction. Programs in the language consist of an expression to evaluate and a set of function definitions. An expression can be a variable, constructor application, λ-abstraction, function call, application, **case** or **let**. Variables introduced by λ-abstraction, **let** or **case** patterns are *bound*; all other variables are *free*. We use $fv(e)$ and $bv(e)$ to denote the free and bound variables respectively of expression e. We write $e_1 \equiv e_2$ if e_1 and e_2 differ only in the names of bound variables.

It is assumed that the input program contains no **let** expressions; these are only introduced during transformation. Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. It is assumed that the language is typed using the Hindley-Milner polymorphic typing system [16, 3] so erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x\ \rightarrow e)$ **of** $p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$ cannot occur.

Definition 2.1 (Substitution)**.** $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ denotes a *substitution*. If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 2.2 (Renaming)**.** $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$, where σ is a bijective mapping, denotes a *renaming*. If e is an expression, then $e\sigma = e\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ is the result of simultaneously replacing the variables $x_1, \ldots, x_n$ with the corresponding variables $x'_1, \ldots, x'_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 2.3 (Shallow Reduction Context)**.** A shallow reduction context $\mathcal{R}$ is an expression containing a single hole $\bullet$ in the place of the redex, which can have one of the two following possible forms:

$$\mathcal{R} ::= \bullet\ e \mid (\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1 \mid \ \ldots\ \mid p_n \rightarrow e_n)$$

Definition 2.4 (Evaluation Context)**.** An evaluation context $\mathcal{E}$ is represented as a sequence of shallow reduction contexts (known as a *zipper* [6]), representing the nesting of these contexts from innermost to outermost within which the expression redex is contained. An evaluation context can therefore have one of the two following possible forms:

$$\mathcal{E} ::= \langle\rangle \mid \langle \mathcal{R} : \mathcal{E} \rangle$$

Definition 2.5 (Insertion into Redex)**.** The insertion of an expression e into the redex of an evaluation context κ, denoted by $\kappa\bullet e$, is defined as follows:

$$\begin{array}{lcl} \langle\rangle\bullet e & = & e \\ \langle(\bullet\ e') : \kappa\rangle\bullet e & = & \kappa\bullet(e\ e') \\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n) : \kappa\rangle\bullet e & & \\ \qquad = & \kappa\bullet(\textbf{case}\ e\ \textbf{of}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n) \end{array}$$

Free variables within the expression e may become bound within $\kappa\bullet e$; if $\kappa\bullet e$ is closed then we call κ a *closing context* for e.

3 Positive Supercompilation

In this section, we give our own higher-order formulation of the positive supercompilation algorithm [22]. At the heart of the positive supercompilation algorithm are a number of *driving* rules which reduce a term (possibly containing free variables) using normal-order reduction. Function unfolding is performed as a part of this reduction process, and folding is performed upon encountering a renaming of a memoised expression. To ensure the termination of the transformation, generalization is performed when an expression is encountered which is a *homeomorphic embedding* of a memoised expression, denoted by $\lesssim$.

The homeomorphic embedding relation was derived from results by Higman [5] and Kruskal [12] and was defined within term rewriting systems [4] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [21, 20], partial evaluation [14] and partial deduction [1, 13].

Definition 3.1 (Well-Quasi Order)**.** A well-quasi order on a set S is a reflexive, transitive relation $\lesssim$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \lesssim s_j$.

This ensures that in any infinite sequence of expressions $e_0, e_1, \ldots$ there definitely exists some $i < j$ where $e_i \lesssim e_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely.

Definition 3.2 (Embedding of Expressions)**.** To define our homeomorphic embedding relation on expressions $\lesssim$, we first define a relation $\trianglelefteq$ as shown in Figure 2, where $e_1 \trianglelefteq e_2$ if e_1 is embedded in e_2 and all of the free variables within e_1 and e_2 also match up.

An expression is embedded within another by this relation if either *diving* (denoted by $\trianglelefteq_d$) or *coupling* (denoted by $\trianglelefteq_c$) can be performed. Diving occurs when an expression is embedded in a sub-expression of another expression, and coupling occurs when two expressions have the same top-level construct and all the corresponding sub-expressions of the two constructs are embedded. Our version of this embedding relation extends previous versions to handle λ-abstractions and **case** expressions that contain bound variables. In these instances, the bound variables within the two expressions must also match up.

$$\frac{e_1 \trianglelefteq_c e_2}{e_1 \trianglelefteq e_2} \qquad \frac{e_1 \trianglelefteq_d e_2}{e_1 \trianglelefteq e_2}$$

$$x \trianglelefteq_c x \qquad f \trianglelefteq_c f$$

$$\frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq e'_i}{(c\ e_1 \ldots e_n) \trianglelefteq_c (c\ e'_1 \ldots e'_n)} \qquad \frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq_d (c\ e_1 \ldots e_n)}$$

$$\frac{e \trianglelefteq (e'\{x' \mapsto x\})}{\lambda x.e \trianglelefteq_c \lambda x'.e'} \qquad \frac{e \trianglelefteq e'}{e \trianglelefteq_d \lambda x.e'}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad e_1 \trianglelefteq e'_1}{(e_0\ e_1) \trianglelefteq_c (e'_0\ e'_1)} \qquad \frac{\exists i \in \{0, 1\}.e \trianglelefteq e_i}{e \trianglelefteq_d (e_0\ e_1)}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad \forall i \in \{1 \ldots n\}.\exists \sigma.p_i \equiv (p'_i\ \sigma) \wedge e_i \trianglelefteq (e'_i\ \sigma)}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n) \trianglelefteq_c (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 \rightarrow e'_1 | \ldots | p'_n \rightarrow e'_n)}$$

$$\frac{\exists i \in \{0 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq_d (\textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n)}$$

Figure 2: Homeomorphic Embedding Relation

The homeomorphic embedding relation $\lesssim$ can now be defined as follows:

$$e_1 \lesssim e_2 \text{ iff } \exists \sigma.e_1\sigma \trianglelefteq_c e_2$$

Within this relation the two expressions must be coupled but, since σ is a renaming, there is no longer a requirement that all of the free variables within the two expressions match up. Generalizing only when two expressions are coupled ensures that the result is not a variable, and there is no need for a *split* operation as used in [21].

Example 3. Some examples of homeomorphic embedding are as follows:

1.	$f\ (g\ x) \lesssim f\ (g\ y)$	6.	$f\ (g\ x) \not\lesssim g\ (f\ y)$
2.	$f\ (h\ x) \lesssim f\ (g\ (h\ y))$	7.	$g\ (h\ x) \not\lesssim f\ (g\ (h\ y))$
3.	$f\ x\ y \lesssim f\ z\ z$	8.	$f\ z\ z \not\lesssim f\ x\ y$
4.	$f\ x\ x \lesssim f\ (g\ y)\ (h\ y)$	9.	$f\ (g\ y)\ (h\ y) \not\lesssim f\ x\ x$
5.	$\lambda x.x \lesssim \lambda y.y$	10.	$\lambda x.x \not\lesssim \lambda y.x$

Theorem 3.3. *The homeomorphic embedding relation $\lesssim$ is a well-quasi-order.*

Proof. The proof is identical to that in [10]. It involves showing that there are a finite number of functors (function names and constructors) in the language. Applications of different arities are replaced with separate constructors; we prove that arities are bounded so there are a finite number of these. We also replace case expressions with constructors. Since our homeomorphic embedding relation requires that the bound variables in expressions match up, bound variables are defined using de Bruijn indices, and each of these are replaced with separate constructors; we also prove that de Bruijn indices are bounded. The overall number of functors is therefore finite, so Kruskal's tree theorem can then be applied to show that $\lesssim$ is a well-quasi-order. □

Definition 3.4 (Generalization)**.** The generalization of two expressions e_1 and e_2 is a triple $(e_g, \theta_1, \theta_2)$ where θ_1 and θ_2 are substitutions such that $e_g\theta_1 \equiv e_1$ and $e_g\theta_2 \equiv e_2$.

The generalization we define for expressions e_1 and e_2 is the *most specific generalization*, denoted by $e_1 \sqcap e_2$, as defined in term algebra [4]. When an expression is generalized, sub-expressions within it are replaced with variables, which implies a loss of knowledge about the expression. The most specific generalization therefore entails the least possible loss of knowledge.

Definition 3.5 (Most Specific Generalization)**.** A most specific generalization of expressions e_1 and e_2 is a generalization $(e_g, \theta_1, \theta_2)$ such that for every other generalization $(e'_g, \theta'_1, \theta'_2)$ of e_1 and e_2, e_g is an instance of e'_g.

Definition 3.6 (The Generalization Operator $\sqcap$)**.** The most specific generalization of two expressions e_1 and e_2, denoted by $e_1 \sqcap e_2$, is defined as shown in Figure 3.

$$x \sqcap x = x$$

$$f \sqcap f = f$$

$$\begin{array}{ll}(c\ e_1 \ldots e_n) \sqcap (c\ e'_1 \ldots e'_n) = & (c\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i) \\ & \text{where} \\ & \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i\end{array}$$

$$\begin{array}{ll}(\lambda x.e_0) \sqcap (\lambda x'.e'_0) = & (\lambda x.e^g_0, \theta_0, \theta'_0) \\ & \text{where} \\ & (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap (e'_0\{x' \mapsto x\})\end{array}$$

$$\begin{array}{ll}(e_0\ e_1) \sqcap (e'_0\ e'_1) = & (e^g_0\ e^g_1, \theta_0 \cup \theta_1, \theta'_0 \cup \theta'_1) \\ & \text{where} \\ & (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \\ & (e^g_1, \theta_1, \theta'_1) = e_1 \sqcap e'_1\end{array}$$

$$\begin{array}{l}(\textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1 | \ldots | p_n \to e_n) \sqcap (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 \to e'_1 | \ldots | p'_n \to e'_n) = \\ \quad (\textbf{case}\ e^g_0\ \textbf{of}\ p_1 \to e^g_1 | \ldots | p_n \to e^g_n, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta'_i) \\ \quad \text{where} \\ \quad (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \\ \quad \forall i \in \{1 \ldots n\}.\exists \sigma.p_i \equiv (p'_i\ \sigma) \wedge (e^g_i, \theta_i, \theta'_i) = e_i \sqcap (e'_i\ \sigma)\end{array}$$

$$e \sqcap e' = (x, \{x \mapsto e\}, \{x \mapsto e'\}) \qquad \text{in all other cases } (x \text{ is fresh})$$

Figure 3: Generalization Rules

Within these rules, if both expressions have the same top-level construct, this is made the top-level construct of the resulting generalized expression, and the corresponding sub-expressions within the construct are then generalized. Otherwise, both expressions are replaced by the same fresh variable. It is assumed that the new variables introduced are all different and distinct from the original program variables. The following rewrite rule is exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions that were previously given different names:

$$(e, \theta \cup \{x \mapsto e', x' \mapsto e'\}, \theta' \cup \{x \mapsto e'', x' \mapsto e''\}) \Rightarrow (e\{x \mapsto x'\}, \theta \cup \{x' \mapsto e'\}, \theta \cup \{x' \mapsto e''\})$$

The results of applying this most specific generalization to items 1-5 in Example 3 are as follows:

1. $(f\ g\ v, \{v \mapsto x\}, \{v \mapsto y\})$
2. $(f\ v, \{v \mapsto h\ x\}, \{v \mapsto g\ (h\ y)\})$
3. $(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto y\}, \{v_1 \mapsto z, v_2 \mapsto z\})$
4. $(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto x\}, \{v_1 \mapsto g\ y, v_2 \mapsto h\ y\})$
5. $(\lambda x.x, \{\}, \{\})$

During transformation, **let** expressions are introduced to represent the results of generalization; note that there were no **let** expressions in the original program and these are only introduced as a result of generalization. We define an abstraction operation on expressions that extracts the sub-terms resulting from generalization.

Definition 3.7 (Abstraction Operation)**.**

$$\begin{array}{l} \mathit{abstract}(e, e') = \textbf{let}\ x_1\ =\ e_1, \ldots, x_n\ =\ e_n\ \textbf{in}\ e_0 \\ \text{where}\quad e \sqcap e' = (e_0, \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}, \theta) \end{array}$$

Positive supercompilation effectively performs a normal-order reduction of the input program. Previously encountered terms are memoised and if the current term is a renaming of a memoised one, then folding is performed, and the transformation is complete. If the current term has a memoised term embedded, then generalization is performed, and the sub-terms of the generalization are further transformed. Generalization ensures that a renaming of a memoised term is always eventually encountered, and that the transformation therefore terminates. The rules for our formulation of positive supercompilation are as shown in Figure 4.

The rules $\mathcal{T}$ are defined on an expression and its surrounding context, denoted by κ. Only those expressions that have a function in the redex position (immediately prior to unfolding) are memoised in rule (6). These expressions are replaced by a new function call, and this new function call is associated with the expression it replaced in the set ρ. On encountering a renaming of a memoised expression contained in ρ, it is also replaced by a corresponding call of its associated new function. The parameter Δ contains the set of function definitions in the original program.

The rules $\mathcal{T}'$ are defined on an expression and its surrounding context, also denoted by κ. These rules are applied when the normal-order reduction of the input program becomes 'stuck' as a result of encountering a variable in the redex position. The expression will already have been transformed and so is not transformed any further, but the surrounding context is further transformed. In rule (12), if the context surrounding a variable redex is a **case**, then information is propagated to each branch of the **case** to indicate that this variable has the value of the corresponding branch pattern.

We can see that there are no trivial loops in the rules $\mathcal{T}$ and $\mathcal{T}'$. In the rules $\mathcal{T}$, a reduction step is performed in rules (3), (5) and (6), and sub-expressions of the redex are transformed in rules (2), (4), (7), (8) and (9). In rule (1), the rules $\mathcal{T}'$ are invoked, and in each of these rules, sub-expressions of the context are further transformed using the rules $\mathcal{T}$. Non-termination can therefore only occur if the terms encountered in the transformation rules grow uncontrollably.

$$
\begin{array}{ll}
(1) & \mathcal{T}[\![x]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}'[\![x]\!]\ \kappa\ \rho\ \Delta \\
(2) & \mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \Delta = c\ (\mathcal{T}[\![e_1]\!]\ \langle\rangle\ \rho\ \Delta) \ldots (\mathcal{T}[\![e_n]\!]\ \langle\rangle\ \rho\ \Delta) \\
(3) & \mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1' \mid \cdots \mid p_k \rightarrow e_k') : \kappa\rangle\ \rho\ \Delta = \\
& \qquad \mathcal{T}[\![e_i'\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa\ \rho\ \Delta\ (p_i = c\ x_1 \ldots x_n) \\
(4) & \mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle\rangle\ \rho\ \Delta = \lambda x \rightarrow (\mathcal{T}[\![e]\!]\ \langle\rangle\ \rho\ \Delta) \\
(5) & \mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta = \mathcal{T}[\![e\{x \mapsto e'\}]\!]\ \kappa\ \rho\ \Delta \\
(6) & \mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta = \begin{cases} e\sigma & \text{if } \exists (e = e') \in \rho, \sigma.e'\sigma \equiv \kappa\bullet f \\ \mathcal{T}[\![abstract(\kappa\bullet f, e')]\!]\ \langle\rangle\ \rho\ \Delta & \text{if } \exists (e = e') \in \rho.e' \lesssim \kappa\bullet f \\ f'\ x_1 \ldots x_n & \text{otherwise} \\ \textbf{where} & (f' \text{ is fresh}, \{x_1 \ldots x_n\} = fv(\kappa\bullet f)) \\ f' = \lambda x_1 \ldots x_n \rightarrow (\mathcal{T}[\![\Delta(f)]\!]\ \kappa\ (\rho \cup \{f'\ x_1 \ldots x_n = \kappa\bullet f\})\ \Delta) & \end{cases} \\
(7) & \mathcal{T}[\![e\ e']\!]\ \kappa\ \rho\ \Delta = \mathcal{T}[\![e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta \\
(8) & \mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n]\!]\ \kappa\ \rho\ \Delta = \\
& \qquad \mathcal{T}[\![e_0]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta \\
(9) & \mathcal{T}[\![\textbf{let}\ x = e_0\ \textbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta = \textbf{let}\ x = (\mathcal{T}[\![e_0]\!]\ \langle\rangle\ \rho\ \Delta)\ \textbf{in}\ (\mathcal{T}[\![e_1]\!]\ \kappa\ \rho\ \Delta) \\
\\
(10) & \mathcal{T}'[\![e]\!]\ \langle\rangle\ \rho\ \Delta = e \\
(11) & \mathcal{T}'[\![e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta = \mathcal{T}'[\![e\ (\mathcal{T}[\![e']\!]\ \langle\rangle\ \rho\ \Delta)]\!]\ \kappa\ \rho\ \Delta \\
(12) & \mathcal{T}'[\![x]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta = \\
& \quad \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow (\mathcal{T}[\![(\kappa\bullet e_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \Delta) \mid \cdots \mid p_n \rightarrow (\mathcal{T}[\![(\kappa\bullet e_n)\{x \mapsto p_n\}]\!]\ \langle\rangle\ \rho\ \Delta) \\
(13) & \mathcal{T}'[\![e]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta = \\
& \qquad \textbf{case}\ e\ \textbf{of}\ p_1 \rightarrow (\mathcal{T}[\![e_1]\!]\ \kappa\ \rho\ \Delta) \mid \cdots \mid p_n \rightarrow (\mathcal{T}[\![e_n]\!]\ \kappa\ \rho\ \Delta)
\end{array}
$$

Figure 4: Positive Supercompilation Transformation Rules

4 Termination

In [19], three different possible causes of non-termination of positive supercompilation when applied to a first-order functional language were identified: *obstructing function calls*, *accumulating parameters* and *accumulating narrowing*[1]. A further possible cause of non-termination has also been identified in [18] for the deforestation of higher-order functional languages, which also applies to positive supercompilation: *accumulating spines*. We now give examples of each of these causes of non-termination.

$$
\begin{array}{l}
nrev\ xs \\
\textbf{where} \\
nrev\ =\ \lambda xs\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil\ \rightarrow Nil \\
\qquad\qquad\qquad\quad \mid Cons\ x'\ xs' \rightarrow app\ (nrev\ xs')\ (Cons\ x'\ Nil) \\
app\ =\ \lambda xs\ \rightarrow \lambda ys\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad Nil\ \rightarrow ys \\
\qquad\qquad\qquad\qquad \mid Cons\ x'\ xs' \rightarrow Cons\ x'\ (app\ xs'\ ys)
\end{array}
$$

Figure 5: Example Obstructing Function Call

[1]This was originally called *accumulating side-effects*, but since functional languages do not admit side-effects, we prefer to call this possible cause of non-termination *accumulating narrowing*.

Example 4 (Obstructing Function Call)**.** Consider the program shown in Figure 5. During the transformation of this program, we encounter the progressively larger terms: $nrev\ xs$, **case** $(nrev\ xs)$ **of** $\cdots$, **case** (**case** $(nrev\ xs)$ **of** $\cdots$) **of** $\cdots$, etc. The call to $nrev$ thus prevents the surrounding context from being reduced, so this context continues to grow. This call to $nrev$ is therefore an obstructing function call.

Example 5 (Accumulating Parameter)**.** Consider the program shown in Figure 6.

$$
\begin{array}{l}
arev\ xs \\
\textbf{where} \\
arev \;=\; \lambda xs \rightarrow arev'\ xs\ Nil \\
arev' \;=\; \lambda xs \rightarrow \lambda ys \rightarrow \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Nil \rightarrow ys \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid Cons\ x'\ xs' \rightarrow arev'\ xs'\ (Cons\ x'\ ys)
\end{array}
$$

Figure 6: Example Accumulating Parameter

During transformation of this program, we encounter the progressively larger terms: $arev'\ xs\ Nil$, $arev'\ xs'\ (Cons\ x'\ Nil)$, $arev'\ xs''\ (Cons\ x''\ (Cons\ x'\ Nil))$, etc. The second parameter in each recursive call to $arev'$ therefore accumulates a progressively larger term.

Example 6 (Accumulating Narrowing)**.** Consider the program shown in Figure 7.

$$
\begin{array}{l}
app\ xs\ xs \\
\textbf{where} \\
app \;=\; \lambda xs \rightarrow \lambda ys \rightarrow \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Nil \rightarrow ys \\
\qquad\qquad\qquad\qquad\qquad \mid Cons\ x\ xs \rightarrow Cons\ x\ (app\ xs\ ys)
\end{array}
$$

Figure 7: Example Accumulating Narrowing

During transformation of this program, we encounter the progressively larger terms: $app\ xs\ xs$, $app\ xs'\ (Cons\ x'\ xs')$, $app\ xs''\ (Cons\ x'\ (Cons\ x''\ xs''))$, etc. The second parameter in each recursive call to app therefore also accumulates a progressively larger term, but in this case the accumulation is caused by unification-based information propagation (narrowing).

Example 7 (Accumulating Spine)**.** Consider the program shown in Figure 8.

$$
\begin{array}{l}
f\ x \\
\textbf{where} \\
f \;=\; \lambda x \rightarrow f\ x\ x
\end{array}
$$

Figure 8: Example Accumulating Spine

During transformation of this program, we encounter the progressively larger terms: $f\ x$, $f\ x\ x$, $f\ x\ x\ x$, etc. Each recursive call to f therefore accumulates an additional parameter. We should also note that this type of function definition is prohibited in most typing schemes.

In all of the above examples, a previously encountered term becomes embedded within a subsequent one. Since the sequence of transformation steps in each example must always include function unfolding, this embedding will be detected by our positive supercompilation algorithm and generalization will be performed, thus ensuring termination. However, not all recursion has to take place through named functions; this will also occur if there is a λ-term which is not strongly normalizing, as is the case for the programs given in examples 1 and 2. To ensure termination, we therefore need to make sure that all recursive functions are named.

5 Ensuring Termination

In this section, we show how to ensure the termination of our formulation of positive supercompilation. As shown in the previous section, non-termination can occur even in the absence of named functions if we have a λ-term which is not strongly normalizing. The sequence of terms obtained will require no function unfolding, and therefore will not be checked for possible folding or generalization. To avoid this possibility, we require that programs are in λ-*prefix* form, in which the only λ-abstractions occur in the prefix of the program expression or the prefix of function bodies. This λ-prefix form is defined as shown in Figure 9.

$prog$	$::= pf_0$ **where** $f_1 = pf_1 \ldots f_n = pf_n$	Program
pf	$::= \lambda x \ \rightarrow pf$	λ-Abstraction
	$\mid pf'$	λ-Free Expression
pf'	$::= x$	Variable
	$\mid c \ pf'_1 \ldots pf'_n$	Constructor
	$\mid f$	Function Call
	$\mid pf'_0 \ pf'_1$	Application
	$\mid$ **let** $x = pf'_0$ **in** pf'_1	Let Expression
	$\mid$ **case** pf'_0 **of** $p_1 \rightarrow pf'_1 \mid \cdots \mid p_n \rightarrow pf'_n$	Case Expression
p	$::= c \ x_1 \ldots x_n$	Pattern

Figure 9: λ-Prefix Form

It is quite straightforward to convert any program into this form; simply replace any λ-abstractions which are not in the prefix of the program expression or the prefix of a function body with a freshly named function. λ-lifting [7] is also performed to abstract over any of the free variables in the λ-abstraction, as named functions in our language cannot contain free variables. If the λ-abstraction matches one which has already been replaced by a function call, then it is replaced with a call to the same function as previously, thus minimizing the number of new function definitions which are introduced.

Example 8. Consider the program given in Example 2. This contains four λ-abstractions which are not in its prefix. The abstraction over f is made the body of the freshly named function f_1 and the abstraction over y is made the body of the freshly named function f_2. The other two abstractions over x are identical, so this abstraction is made the body of the freshly named function f_3; however, since the variable f appears free in this expression, λ-lifting is performed to abstract over f, and this extra parameter is added to the two calls of f_3.

We now prove that our simple pre-processing step is sufficient to ensure the termination of our formulation of the positive supercompilation algorithm. We do this by showing that if the original input to our positive supercompilation algorithm is in λ-prefix form, then all of the terms subsequently encountered must be in a particular form. We then show that any potentially infinite sequence of transformation steps in which the expressions are in this form must include function unfolding, so the transformation is guaranteed to terminate.

Lemma 5.1 (On The Form of Terms Encountered by Positive Supercompilation)**.** If the input to our positive supercompilation algorithm is in λ-prefix form, then all of the terms subsequently encountered must have the following form in which the only λ-abstractions are in the prefix of the redex:

$$\begin{array}{rl} sf ::= & \textbf{case } sf \textbf{ of } p_1 \rightarrow pf'_1 \mid \cdots \mid p_n \rightarrow pf'_n \\ \mid & sf\ pf' \\ \mid & pf \end{array}$$

where pf and pf' are as defined in figure 9.

Proof. The interesting cases are where substitution is performed in rules (3) and (5), and where generalisation is performed in rule (6). Since the only λ-abstractions can be in the redex, the terms which are substituted in rules (3) and (5) cannot contain any λ-abstractions, so the resulting term must also be in the above form. Generalization is performed in rule (6) when the redex is a function name, so the generalized term cannot contain any λ-abstractions and neither can the extracted sub-expressions. Details of the proof are given in Appendix A. □

Lemma 5.2. If the input to our positive supercompilation algorithm is in λ-prefix form, then every infinite sequence of transformation steps must include function unfolding.

Proof. Every infinite sequence of transformation steps must include either function unfolding or λ-application. If the input term is in λ-prefix form, then by Lemma 5.1, the only λ-abstractions in the terms subsequently encountered will be in the prefix of the redex, so transformation rule (4) or (5) will be continually applied until there are no λ-abstractions remaining in the current term. Thus, the only way in which new λ-abstractions can be introduced is by function unfolding. Thus every infinite sequence of transformation steps must include function unfolding. □

We are now able to prove our desired result.

Theorem 5.3. If the input to our positive supercompilation algorithm is in λ-prefix form, then it is guaranteed to terminate.

Proof. The proof is by contradiction. If our positive supercompilation algorithm did not terminate then the set of memoised expressions ρ must be infinite, since by Lemma 5.2 every infinite sequence of transformation steps must include function unfolding. Every new expression which is added to ρ cannot have any of the previous expressions in ρ embedded within it by the homeomorphic embedding relation $\lesssim$, since folding or generalization would have been performed instead. However, this contradicts the fact that $\lesssim$ is a well-quasi-order (Theorem 3.3). □

6 Conclusion and Related Work

In this paper, we have described a simple pre-processing step which can be applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there must be an unfolding. This involves introducing names for some anonymous functions to ensure that only memoising expressions immediately preceding an unfold step is sufficient to ensure termination of the transformation. The original positive supercompilation algorithm [19, 22] was only formulated for a first-order language, so it was sufficient to only memoise the expressions immediately prior to an unfolding step. In the higher-order formulations of positive supercompilation given by Mitchell [17] and Bolingbroke [2], *all* expressions are memoised. We argue that the extra work required for the additional checking for generalization and folding is too computationally expensive, particularly since the homeomorphic embedding check is very time consuming; this has been borne out by the experimental results obtained using this approach. It should also be pointed out that the implementation of positive supercompilation in [17] will not terminate on programs such as that given in Example 2. This is because the simplification rules that are applied to terms prior to transformation by positive supercompilation will not terminate for such programs which use *contravariant* (*negative*) data types. It is argued in [17] that this problem only occurs for contrived programs, and it is also a problem for GHC, which will not terminate when compiling this example program. However, this seems unsatisfactory. It is noted in [17] that this non-termination problem could be avoided by not performing simplification on negative data types. A similar approach was also adopted by Jonsson [8] and Mendel-Gleason [15] by requiring that all types in the input program are *positive*. This also seems unsatisfactory since such typing schemes are not used in mainstream functional languages.

Rather than memoising all expressions, the approach taken in the higher-order supercompiler HOSC [9, 10, 11] is to restrict this to only those expressions which are considered to be *non-trivial*. In HOSC 1.0 [9], an expression is considered to be non-trivial if it either has a function in the redex or an irreducible expression in the selector of a **case** expression (corresponding to the left hand side of our rules (6), (12) and (13)). However, it was subsequently discovered [10] that this was not sufficient to ensure the termination of the supercompiler, because it will not terminate for programs which encode recursion using a data type such as that given in Example 2. In HOSC 1.1 [10], an expression for which the next transformation step involves a substitution (corresponding to the left hand side of our rules (3) and (5)) is considered to be non-trivial if it satisfies a size constraint in which the expression resulting from the substitution is no smaller than the expression before substitution. However, it was subsequently discovered [11] that memoising every expression for which the next transformation step involves a β-reduction produces poor residual programs. In HOSC 1.5 [11], expressions for which the next transformation step involves a β-reduction are not memoised, but all applications and **case** expressions are (corresponding to the left hand side of our rules (7) and (8)), thus ensuring that in any potentially infinite sequence of transformation steps expressions will still be memoised. However, we argue that this approach still requires a lot of additional work when checking for generalization and folding, and can still produce poor residual programs. Using our approach, after the pre-processing step, only those expressions encountered immediately prior to an unfolding step (rule (6)) need to be memoised and checked for generalization and folding.

Using our approach, we are not able to prove that the programs resulting from transformation are an improvement over the original programs, since not all new functions are introduced in conjunction with the unfolding of an old function. However, this is a problem for all of the

described algorithms for higher-order positive supercompilation. However, using our approach, less new functions will be created and generalization will be performed less often, resulting in more improved residual programs. We have implemented the techniques described in this paper and preliminary experiments show that they make the resulting supercompiler more efficient and that they produce more improved residual programs in some cases.

Acknowledgements

Many thanks to the anonymous referees who provided very useful comments and feedback on an earlier version of this paper. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

References

[1] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1–2):25–46, 1993.

[2] Max Bolingbroke and Simon Peyton Jones. Supercompilation by Evaluation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, pages 135–146 , 2010.

[3] L. Damas and R. Milner. Principal Type Schemes for Functional Programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.

[5] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

[6] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[7] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the Workshop on Implementation of Functional Languages*, pages 165–180, February 1985.

[8] Peter Jonsson. *Time- and Size-Efficient Supercompilation.* PhD thesis, Dept. of Computer Science and Electrical Engineering, Lulea University of Technology, 2011.

[9] Ilya Klyuchnikov. Supercompiler HOSC 1.0: Under the Hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.

[10] Ilya Klyuchnikov. Supercompiler HOSC 1.1: Proof of Termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[11] Ilya Klyuchnikov. Supercompiler HOSC 1.5: Homeomorphic Embedding and Generalization in a Higher-Order Setting. Preprint 62, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[12] J.B. Kruskal. Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[13] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of the International Static Analysis Symposium, Pisa, Italy*, pages 230–245, 1998.

[14] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle.* PhD thesis, Université de Nice - Sophia Antipolis, 1994.

[15] Gavin Mendel-Gleason. *Types and Verification for Infinite State Systems.* PhD thesis, School of Computing, Dublin City University, 2012.

[16] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[17] Neil Mitchell. Rethinking Supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 309–320 , 2010.

[18] H. Seidl and M. H. Sørensen. Constraints to Stop Higher-Order Deforestation. *Proceedings of the Twelfth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 400–413, 1997.

[19] M. H. Sørensen. Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[20] M. H. Sørensen. Convergence of Program Transformers in the Metric Space of Trees. *Lecture Notes in Computer Science*, 1422:315–337, 1998.

[21] M. H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *Lecture Notes in Computer Science*, 787:335–351, 1994.

[22] M. H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[23] V.F. Turchin. Program Transformation by Supercompilation. *Lecture Notes in Computer Science*, 217:257–281, 1985.

[24] V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.

A Proof of Lemma 5.1

We need to prove that for each of the transformation rules given in Figure 4, if $\mathcal{T}[\![e]\!]\ \kappa\ \rho\ \Delta = \ldots \mathcal{T}[\![e_1]\!]\ \kappa_1\ \rho_1\ \Delta \ldots \mathcal{T}[\![e_n]\!]\ \kappa_n\ \rho_n\ \Delta \ldots$ and $\kappa \bullet e \in sf$, then $\forall i \in \{1 \ldots n\}.\kappa_i \bullet e_i \in sf$.

Case (1): $\mathcal{T}[\![x]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}'[\![x]\!]\ \kappa\ \rho\ \Delta$

All further applications of $\mathcal{T}$ arising from this application of $\mathcal{T}'$ are applied to sub-expressions from the context κ. Since $\kappa \bullet x \in sf$, then for all sub-expressions e_i in κ, $e_i \in pf'$, so $e_i \in sf$

Case (2): $\mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \Delta = c\ (\mathcal{T}[\![e_1]\!]\ \langle\rangle\ \rho\ \Delta) \ldots (\mathcal{T}[\![e_n]\!]\ \langle\rangle\ \rho\ \Delta)$

Since $c\ e_1 \ldots e_n \in sf$, then $\forall i \in \{1 \ldots n\}.e_i \in pf'$, so $\forall i \in \{1 \ldots n\}.e_i \in sf$
Case (3): $\mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1'\ |\cdots|\ p_k \rightarrow e_k') : \kappa\rangle\ \rho\ \Delta =$
$\mathcal{T}[\![e_i'\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa\ \rho\ \Delta\ (p_i = c\ x_1 \ldots x_n)$

Since $\langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1'\ |\cdots|\ p_k \rightarrow e_k') : \kappa\rangle \bullet (c\ e_1 \ldots e_n) \in sf$, then $\forall i \in \{1 \ldots n\}.e_i \in pf' \wedge \forall i \in \{1 \ldots k\}.\kappa \bullet e_i{}' \in pf'$, so $\kappa \bullet e_i'\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\} \in sf$

Case (4): $\mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle\rangle\ \rho\ \Delta = \lambda x \rightarrow (\mathcal{T}[\![e]\!]\ \langle\rangle\ \rho\ \Delta)$

Since $\lambda x\ \rightarrow e \in sf$, then $e \in pf$, so $e \in sf$

Case (5): $\mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta = \mathcal{T}[\![e\{x \mapsto e'\}]\!]\ \kappa\ \rho\ \Delta$

Since $\langle(\bullet\ e') : \kappa\rangle \bullet (\lambda x\ \rightarrow e) \in sf$, then $\kappa \bullet e \in pf \wedge e' \in pf'$, so $\kappa \bullet e\{x \mapsto e'\} \in sf$

Case (6a): $\mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta = e\sigma$ if $\exists (e = e') \in \rho, \sigma.e'\sigma \equiv \kappa \bullet f$

No further transformation is performed.

Case (6b): $\mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}[\![abstract(\kappa\bullet f, e')]\!]\ \langle\rangle\ \rho\ \Delta$ if $\exists(e = e') \in \rho.e' \lesssim \kappa\bullet f$

Since $\kappa\bullet f \in sf$, then $\kappa\bullet f \in pf'$, so $abstract(\kappa\bullet f, e') \in sf$

Case (6c): $\mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta =$ $f'\ x_1 \ldots x_n$ otherwise
where
$f' = \lambda x_1 \ldots x_n \rightarrow (\mathcal{T}[\![\Delta(f)]\!]\ \kappa\ (\rho \cup \{f'\ x_1 \ldots x_n = \kappa\bullet f\})\ \Delta)$
(f' is fresh, $\{x_1 \ldots x_n\} = fv(\kappa\bullet f)$)

Since $\kappa\bullet f \in sf \wedge \Delta(f) \in pf$, then $\kappa\bullet\Delta(f) \in sf$

Case (7): $\mathcal{T}[\![e\ e']\!]\ \kappa\ \rho\ \Delta = \mathcal{T}[\![e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta$

Since $\kappa\bullet(e\ e') \in sf$, then $\langle(\bullet\ e') : \kappa\rangle\bullet e \in sf$

Case (8): $\mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1\ |\cdots|\ p_n \rightarrow e_n]\!]\ \kappa\ \rho\ \Delta =$
$\mathcal{T}[\![e_0]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1\ |\cdots|\ p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta$

Since $\kappa\bullet(\textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1\ |\cdots|\ p_n \rightarrow e_n) \in sf$, then $\langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1\ |\cdots|\ p_n \rightarrow e_n) :$
$\kappa\rangle\bullet e_0 \in sf$

Case (9): $\mathcal{T}[\![\textbf{let}\ x = e_0\ \textbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta = \textbf{let}\ x = (\mathcal{T}[\![e_0]\!]\ \langle\rangle\ \rho\ \Delta)\ \textbf{in}\ (\mathcal{T}[\![e_1]\!]\ \kappa\ \rho\ \Delta)$

Since $\kappa\bullet(\textbf{let}\ x = e_0\ \textbf{in}\ e_1) \in sf$, then $e_0 \in pf' \wedge \kappa\bullet e_1 \in pf'$, so $e_0 \in sf \wedge \kappa\bullet e_1 \in sf$