Transforming Lazy Functions using Comportment Properties

Ross Paterson

University of North London

Abstract. This paper develops a source-to-source transformation that seeks to delay the creation of thunks as much as possible, often eliminating them altogether. The generated programs make use of simpler representations, including types in which all expressions terminate. The transformation is guided by an abstract interpretation, with the abstract values of basic comportment analysis concretized as representation types related by injections.
The transformation of functions is polyvariant, with functions mapped to natural transformations. This leads to a clean treatment of higher order functions.
Keywords: static analysis, abstract interpretation, automatic program transformation

1 Introduction

If a function is lazy, its argument must be passed as a suspended computation, or *thunk*. This object is typically created on the heap, or *boxed*, and the function using it must inspect the thunk to determine whether it has been evaluated. It has long been recognized that these thunks are a major obstacle to the application of conventional compiler technology to lazy functional languages. They have been attacked using strictness analysis [9], which determines that certain sub-expressions must be evaluated in order to produce the final output. Instead of being suspended, such expressions may be safely evaluated early, and their values passed in an unboxed form on the stack.

There is also a dual approach, known as termination analysis, which determines that certain sub-expressions will have been evaluated by a certain point. If we know that an argument of a function will have been evaluated by the time the function is called, we can pass the argument in unboxed form to a call-by-value variant of the function (even if the function is not strict).

A typical functional language incorporates lazy data structures, and strictness analysis techniques have been developed that produce information about latent demand [14]. Similarly termination analysis may be extended to produce statements about latent termination: though an expression might not terminate, it may be that if it does then its components will also have been evaluated.

The information supplied by these analyses has been used to improve compiler performance, but the changes to the code generator are seldom formally verified. Exceptions include the work of Danvy and Hatcliffe [3] and of Burn and Le Métayer [1], who describe modifications of the continuation-passing transform to incorporate simple strictness information. Paterson [11] defines a transformation based on projection-based strictness analysis. The transformation moves backwards to perform evaluation as early as possible, and as a side-benefit discovers unused components of data structures. The transformation of this paper is dual to that one; it moves forward, delaying boxing as much as possible, and as a side-benefit discovers unused alternatives.

We shall confine our attention to forwards analyses, which may infer that the value of an expression lies within a subset of values, called a *comportment property* [2]. It is customary to render the analysis tractable by using a finite set of properties, indexed by an abstract domain. For the purposes of simple strictness analysis, it suffices to use properties that are ideals, but in order to encompass other properties, including termination, totality and constancy, this restriction must be dropped.

This paper is concerned with exploiting the results of comportment analysis to transform programs into forms operating on reduced types. For this we require a source language whose type system allows us to express such distinctions as whether or not a type includes the bottom element $\perp$. This is routine in denotational semantics; we need only model our source language on the denotational meta-language [8]. An appropriately modified typed λ-calculus is discussed in Sect. 2.

Next we develop concrete representations of basic comportment properties over each type as subtypes, each with an embedding into the original type. As defined by Cousot and Cousot [2], the qualification "basic" indicates that the properties we use are not closed under disjunction (union). Thus we use an approximation of union in the analysis of conditionals, losing some information in comparison with the full-flown comportment analysis. Section 3 presents the easy case of non-functional types. The principal difficulty is with function types, discussed in Sect. 4. Since a function will be specialized in different ways for different input subtypes, we can represent it as a family of related functions, i.e. a natural transformation. The class of natural transformations is sufficiently constrained that it may be embedded in the original function type.

In Sect. 5 we define an abstract interpretation and accompanying source-to-source transformation, which transforms an expression to one of the reduced type. The abstract interpretation is a fairly standard comportment analysis (except for the treatment of fixed points). The novelty is the translation into forms exploiting the subtype representations to achieve greater efficiency. In particular, the creation of closures is delayed as long as possible, and often eliminated.

Our transformation exploits termination properties directly, producing improved representations of values known to terminate, and permitting eager evaluation where it does not harm termination. Strictness properties may be used indirectly, as sketched in Ex. 2 in the next section.

There are some similarities between our work and binding time analysis, which identifies subexpressions whose values will be statically known, given that part of the input will be static. Our abstract interpretation is particularly close to that of Hunt and Sands [7]. However, the transformations we consider are much less ambitious than those performed by specializers using binding time information. On the other hand we use an extended definition of staticness, using static values within dynamic structures. We are also able to transform the program without knowing the values of the static inputs.

2 A Source Language with Explicit Lifting

The denotational semantics of lazy functional programs often refers to a lifting domain constructor $A_\perp$, which adds an additional bottom element to the domain A. For example, a numeric type might be modelled as $N_\perp$ for some set N, and a sum type as $(A+B)_\perp$, where $+$ denotes disjoint union. The addition of the bottom element is required to model the possibility of nontermination. The key to thunk removal is a source language modelled on the meta-language, in which this lifting is explicit, and therefore susceptible to transformation [8]. Such a language might serve as a core language, to which a more comfortable source language is translated as a first step in compilation.

We assume a collection of base types A, B, etc, which correspond to flat sets, without a bottom element. Then the types of our language are described by the following grammar:

$$t ::= B \mid \varnothing \mid t \times t \mid t + t \mid t_\perp \mid t \to t$$

Here $\varnothing$ represents the empty type, and $\times$, $+$ and $\to$ are unlifted. Note that many types, including base types and sum types, have no bottom element. Any expression of such a type must terminate, and can thus be safely evaluated early instead of being suspended in a thunk. The aim then becomes to transform programs to equivalent programs using types with fewer liftings.

On the other hand, we may not define values by general recursion unless the type concerned is known to have a least element $\perp$ and thus support the Kleene definition of fixed points. We define a class of *pointed* types by stating that $t_\perp$ is always pointed, $s \times t$ is pointed if both s and t are, and $s \to t$ is pointed if t is.[1]

Our source language is a simply typed λ-calculus, similar to those of [6,8]. We assume the families of constants c^t listed in Fig. 1. We also assume a fixed set of constants $c^{B_1 \to \cdots B_n \to B}$ for base types $B_1, \ldots, B_n$ and B. Variables x^t are also typed, but we will routinely omit the type superscripts on variables and constants where they are irrelevant or may be deduced from the context.

Most of these constants are familiar, as long as one remembers that sums are unlifted. The main novelty lies in the constants *lift*, *fail* and *ext*, which manipulate lifted types, and satisfy the axioms

$$\mathit{ext}\ f\ \mathit{fail} = \perp \tag{1}$$

$$\mathit{ext}\ f\ (\mathit{lift}\ x) = f\ x \tag{2}$$

[1] If t is pointed, then $t + \varnothing$ and $\varnothing + t$ also have least elements, but we will not be using them here.

$\mathit{empty}^{\varnothing\to t}$		$\mathit{fix}^{(t\to t)\to t}$ for pointed t
$\mathit{fst}^{s\times t\to s}$	$\mathit{inl}^{s\to s+t}$	$\mathit{fail}^{t_\perp}$
$\mathit{snd}^{s\times t\to t}$	$\mathit{inr}^{t\to s+t}$	$\mathit{lift}^{t\to t_\perp}$
$\mathit{pair}^{s\to t\to s\times t}$	$\mathit{choose}^{(s\to u)\to(t\to u)\to s+t\to u}$	$\mathit{ext}^{(s\to t)\to s_\perp\to t}$ for pointed t

Fig. 1. Constants

Operationally, $\mathit{ext}\ f\ x$ forces the evaluation of x to weak head normal form, and if this terminates then (tail-) calls f with the result as argument.

We aim to delay the application of *lift* as long as possible, hoping to cancel it with a subsequent *ext*. There is some similarity to deforestation [4,13], but here we preserve the structure of the source program.

As the standard interpretation of our language is trivial, we shall proceed by proving equalities in the logic associated with our language. This logic comprises axioms including those above (see [6,8] for details), β- and η-equivalence and Scott induction.

Example 1. Suppose the familiar type `Int` is expanded as $\mathbb{Z}_\perp$, for the primitive flat type $\mathbb{Z}$ representing the integers or some finite approximation. The familiar constants `1`, `2`, etc. are similarly expanded as $\mathit{lift}(1)$, $\mathit{lift}(2)$ etc. for $1, 2, \ldots : \mathbb{Z}$. Similarly, functions like $* : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$ are expanded as

$$x * y \triangleq \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\ \mathit{lift}\ (u \star v))\ y)\ x$$

using a primitive function $\star : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, corresponding to a machine instruction. Similarly, the function $\lambda\,\texttt{x. 2 * x}$ might be transformed to

$$\begin{aligned}\lambda\,\texttt{x.2 * x} &= \lambda\, x.\ \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\ \mathit{lift}\ (u \star v))\ x)\ (\mathit{lift}\ 2)\\ &= \mathit{ext}\ (\lambda\, v.\ \mathit{lift}\ (2 \star v))\end{aligned}$$

Example 2. Suppose a function $f : s_\perp \to t$ is discovered to be strict by the analysis discussed here or any other. Then any occurrence of f may be replaced by the term $\mathit{ext}\ (f \circ \mathit{lift})$. Then the transformation of Sect. 5 will promote the *lift* through f, producing an improved translation. In this way we achieve similar effects to previous approaches in which strictness information had been used to enable local optimizations to remove thunks [5,12].

3 Embeddings of Non-functional Types

In this section and the next we define for each type t a lattice of subtypes of t. In the analysis, this lattice is represented by an abstract domain $\mathcal{D}^\#_t$, which is a finite lattice under the ordering $\subseteq^\#$. A concretion mapping $\mathsf{C}_t[\cdot]$ relates the two lattices. Types that do not contain $\to$ are easier to handle, and will be treated first.

Example 3. For the type $\mathbb{Z}_\perp$, the relevant subtypes are

$\varnothing_\perp$**,** which can be used for expressions that are known not to terminate,
$\mathbb{Z}$**,** which can be used for expressions that are known to terminate, and
$\mathbb{Z}_\perp$**,** which can be used for any expression.

To complete the lattice, we add the type $\varnothing$. These types may be arranged in a lattice, with the inclusions made explicit as embedding functions between subtypes, as on the right side of Fig. 2. The figure also shows the corresponding abstract domain $\mathcal{D}^\#_{\mathbb{Z}_\perp}$ and the concretion map.

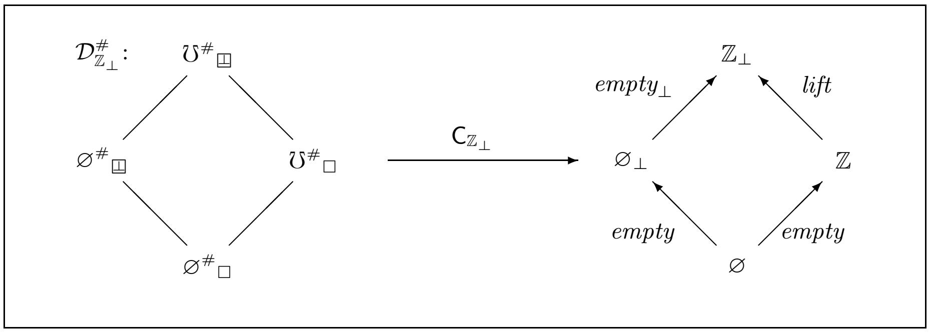

Fig. 2. Abstract domain $\mathcal{D}^\#_{\mathbb{Z}_\perp}$ and lattice of subtypes of $\mathbb{Z}_\perp$

The rather curious names of the abstract values on the left are explained by the definition of the abstract domains $\mathcal{D}^\#_t$ by induction over t:

- For a base type B (for example $\mathbb{Z}$), $\mathcal{D}^\#_B$ consists of the two values $\varnothing^\#_B$ and $\mho^\#_B$, with $\varnothing^\# \subseteq^\# \mho^\#$.
- The domain $\mathcal{D}^\#_\varnothing$ consists of a single value $\varnothing^\#_\varnothing$.
- For any type t, the abstract domain $\mathcal{D}^\#_{t_\perp}$ consists of values $\alpha_{\boxbot}$ and $\alpha_{\square}$, for $\alpha \in \mathcal{D}^\#_t$. (The significance of the notation is that the subtype denoted by $\alpha_{\boxbot}$ includes the bottom element of $t_\perp$, while the subtype denoted by $\alpha_{\square}$ does not.) The constructors $\cdot_{\boxbot}$ and $\cdot_{\square}$ are monotonic, and $\alpha_{\square} \subseteq^\# \alpha_{\boxbot}$ for each α.
- For types s and t, the abstract domain $\mathcal{D}^\#_{s\times t}$ consists of pairs $\alpha \boxtimes \beta$ for $\alpha \in \mathcal{D}^\#_s$ and $\beta \in \mathcal{D}^\#_t$, such that either both α and β are the least elements of their respective lattices, or neither is (i.e. a smash product). The pairs are ordered pointwise.
- For types s and t, the abstract domain $\mathcal{D}^\#_{s+t}$ consists of pairs $\alpha \boxplus \beta$ for $\alpha \in \mathcal{D}^\#_s$ and $\beta \in \mathcal{D}^\#_t$, ordered pointwise.

The concretion maps are also defined by induction on the structure of t, as follows:

$$
\begin{array}{ll}
\mathsf{C}_B[\mho^\#_B] = B & \mathsf{C}_{t_\perp}[\alpha_{\boxplus}] = (\mathsf{C}_t[\alpha])_\perp \\
\mathsf{C}_B[\varnothing^\#_B] = \varnothing & \mathsf{C}_{t_\perp}[\alpha_{\Box}] = \mathsf{C}_t[\alpha] \\
\mathsf{C}_\varnothing[\varnothing^\#_\varnothing] = \varnothing & \mathsf{C}_{s\times t}[(\alpha \boxtimes \beta)] = \mathsf{C}_s[\alpha] \times \mathsf{C}_t[\beta] \\
 & \mathsf{C}_{s+t}[(\alpha \boxplus \beta)] = \mathsf{C}_s[\alpha] + \mathsf{C}_t[\beta]
\end{array}
$$

Each abstract domain $\mathcal{D}^\#_t$ has a least element $\varnothing^\#_t$, indicating no values, whose concrete value is a type that is equivalent to $\varnothing$. At first glance that this might seem useless, since the abstract interpretation can hardly assign this value to any expression. However $\varnothing^\#_t$ is useful in compound abstract values. For example, *inl* produces only values in the left part of a sum type, so the type of its result could be simplified to $s + \varnothing$. Similarly, *fail* produces only the bottom element of a lifted type; the type of its result could be simplified to $\varnothing_\perp$, and the type being lifted could be simplified to $\varnothing$. The abstract version of the product domain is formed using the smash product because we do not wish to have distinct points representing $\varnothing \times \varnothing$ and $\varnothing \times t$.

Each abstract domain $\mathcal{D}^\#_t$ also has a greatest element $\mho^\#_t$. It is easy to establish that if t does not include $\to$, then $\mathsf{C}_t[\mho^\#_t] = t$, though this will not hold for types including $\to$. For each type t, the concretion map C_t maps the abstract domain $\mathcal{D}^\#_t$ to a lattice of subtypes of this type $\mathsf{C}_t[\mho^\#_t]$. To make this lattice explicit, we define for each $\alpha \subseteq^\# \alpha'$ in $\mathcal{D}^\#_t$, a function $\mathsf{C}_t[\alpha \subseteq^\# \alpha'] : \mathsf{C}_t[\alpha] \to \mathsf{C}_t[\alpha']$. These functions are also defined by induction over t, as in Fig. 3.

$$
\mathsf{C}_t[\alpha \subseteq^\# \alpha'] \;:\; \mathsf{C}_t[\alpha] \to \mathsf{C}_t[\alpha']
$$

$$
\begin{aligned}
\mathsf{C}_B[\varnothing^\#_B \subseteq^\# \varnothing^\#_B] &= \mathit{empty} \\
\mathsf{C}_B[\mho^\#_B \subseteq^\# \mho^\#_B] &= \mathit{id} \\
\mathsf{C}_B[\varnothing^\#_B \subseteq^\# \mho^\#_B] &= \mathit{empty} \\
\mathsf{C}_\varnothing[\varnothing^\#_\varnothing \subseteq^\# \varnothing^\#_\varnothing] &= \mathit{empty} \\
\mathsf{C}_{s\times t}[\alpha \boxtimes \beta \subseteq^\# \alpha' \boxtimes \beta'] &= \mathsf{C}_s[\alpha \subseteq^\# \alpha'] \times \mathsf{C}_t[\beta \subseteq^\# \beta'] \\
\mathsf{C}_{s+t}[\alpha \boxplus \beta \subseteq^\# \alpha' \boxplus \beta'] &= \mathsf{C}_s[\alpha \subseteq^\# \alpha'] + \mathsf{C}_t[\beta \subseteq^\# \beta'] \\
\mathsf{C}_{t_\perp}[\alpha_{\boxplus} \subseteq^\# \alpha'_{\boxplus}] &= (\mathsf{C}_t[\alpha \subseteq^\# \alpha'])_\perp \\
\mathsf{C}_{t_\perp}[\alpha_{\Box} \subseteq^\# \alpha'_{\boxplus}] &= \mathit{lift} \circ \mathsf{C}_t[\alpha \subseteq^\# \alpha'] \\
\mathsf{C}_{t_\perp}[\alpha_{\Box} \subseteq^\# \alpha'_{\Box}] &= \mathsf{C}_t[\alpha \subseteq^\# \alpha']
\end{aligned}
$$

Fig. 3. Injections between types $\mathsf{C}_t[\alpha]$

Proposition 4. *For each type t and $\alpha \subseteq^{\#} \beta \in \mathcal{D}_t^{\#}$, the function $\mathsf{C}_t[\alpha \subseteq^{\#} \beta]$ is bimonotone.*[2]

That is, these functions are subspace injections.

Proposition 5. *For each type t and $\alpha \subseteq^{\#} \beta \subseteq^{\#} \gamma \in \mathcal{D}_t^{\#}$,*

$$\mathsf{C}_t[\alpha \subseteq^{\#} \alpha] = id_{\mathsf{C}_t[\alpha]} \tag{3}$$

$$\mathsf{C}_t[\beta \subseteq^{\#} \gamma] \circ \mathsf{C}_t[\alpha \subseteq^{\#} \beta] = \mathsf{C}_t[\alpha \subseteq^{\#} \gamma] \tag{4}$$

That is, C_t is a functor from $\mathcal{D}_t^{\#}$ to the category of our source language.

3.1 A Note on the Computational Pre-order

In addition to the set-based inclusion ordering (our $\subseteq^{\#}$), previous accounts of comportment analysis [2,10] make use of the computational or Egli-Milner ordering for the calculation of fixed points. The requirement that functions also preserve this ordering has the additional benefit of reducing the size of the abstract function domains, making the analysis more tractable.

There is a complication, however: the computational relation between properties is not anti-symmetric. To turn it into an ordering, one must deal with equivalence classes of properties, eliminating the distinction between a property and its convex closure. However we require this distinction for the description of the latent termination properties mentioned in Sect. 1. For example, it may be that we are not sure whether a list-valued expression will terminate, but we know that if it does then the head will not be $\perp$. The use of the computational ordering would identify this property with the set of all lists. Hence we require a different definition of fixed points; we shall use the inclusion ordering.

We can however define a computational *pre-order* $\sqsubseteq^{\#}$, using it to restrict the size of the abstract function domains. Since this pre-order has such a minor role, we only sketch it here. In each abstract domain $\mathcal{D}_t^{\#}$, the point $\varnothing^{\#}$ is isolated in the $\sqsubseteq^{\#}$ pre-order. The domains $\mathcal{D}_{s\times t}^{\#}$ and $\mathcal{D}_{s+t}^{\#}$ inherit the pre-order pointwise. The pre-order $\sqsubseteq^{\#}$ on $\mathcal{D}_{t_\perp}^{\#}$ is defined by the following rules:

$$\begin{gathered}
\alpha \neq \varnothing^{\#} \Rightarrow \alpha_{\boxed{\perp}} \sqsubseteq^{\#} \alpha_{\square} \\
\alpha \sqsubseteq^{\#} \beta \Rightarrow \alpha_{\square} \sqsubseteq^{\#} \beta_{\square} \\
\alpha \sqsubseteq^{\#} \beta \Rightarrow \alpha_{\boxed{\perp}} \sqsubseteq^{\#} \beta_{\boxed{\perp}} \\
\alpha \subseteq^{\#} \beta \Rightarrow \alpha_{\boxed{\perp}} \sqsubseteq^{\#} \beta_{\boxed{\perp}}
\end{gathered}$$

4 Embeddings of Higher Types

So far, our abstract values describe the sets of values that an expression might produce. This cannot be directly extended to functions: while strict functions

[2] A function f is bimonotone if $x \leq y \Leftrightarrow f\ x \leq f\ y$.

$\mathbb{Z}_\perp \to \mathbb{Z}_\perp$ are in one-to-one correspondence with elements of the type $\mathbb{Z} \to \mathbb{Z}_\perp$, and strict total functions may be represented by $\mathbb{Z} \to \mathbb{Z}$, no simple representation is available for functions that are merely known to be total.

In analysing a function, we want to know what set of outputs can be produced from a given set of inputs. In all cases the empty set of inputs should produce the empty set of outputs. We also expect that larger sets of inputs will produce larger sets of outputs, and that more defined inputs will produce more defined outputs. Hence we define $\mathcal{D}^{\#}_{s\to t}$ as the set of functions $\mathcal{D}^{\#}_s \to \mathcal{D}^{\#}_t$ that preserve $\varnothing^{\#}$ and are monotonic with respect to both $\subseteq^{\#}$ and $\sqsubseteq^{\#}$. Both the order $\subseteq^{\#}$ and the pre-order $\sqsubseteq^{\#}$ are extended pointwise to functions.

Example 6. The domain $\mathcal{D}^{\#}_{\mathbb{Z}_\perp\to\mathbb{Z}_\perp}$ has 7 elements, exactly those listed as basic comportments by Cousot and Cousot [2]. We define the abstract functions by their actions on non-$\varnothing^{\#}$ elements:

$$\begin{array}{ll}
\lambda\,\alpha.\,\mho^{\#}{}_{\boxed{\scriptstyle\perp}} & \text{representing all functions} \\
\lambda\,\alpha.\,\mho^{\#}{}_{\square} & \text{constant, total functions} \\
\lambda\,\alpha.\,\varnothing^{\#}{}_{\boxed{\scriptstyle\perp}} & \text{divergent functions} \\
\lambda\,\alpha.\,\alpha & \text{strict, total functions} \\
\lambda\,\alpha.\,\alpha \cup^{\#} \mho^{\#}{}_{\square} & \text{total functions} \\
\lambda\,\alpha.\,\alpha \cup^{\#} \varnothing^{\#}{}_{\boxed{\scriptstyle\perp}} & \text{strict functions} \\
\lambda\,\alpha.\,\varnothing^{\#}{}_{\square} & \text{empty}
\end{array}$$

The orderings are exactly as in [2].

Suppose that the abstract interpretation of a function $f : s \to t$ yields a function $\varphi \in \mathcal{D}^{\#}_s \to \mathcal{D}^{\#}_t$. For each abstract input α, there will be a version $f_\alpha : \mathsf{C}_s[\alpha] \to \mathsf{C}_t[\varphi\ \alpha]$. That is, the transformation is polyvariant.

Example 7. Abstract interpretation of $\mathit{ext}\,(\lambda\,u.\,\mathit{lift}\,(2 \star u)) : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ yields the identity function on $\mathcal{D}^{\#}_{\mathbb{Z}_\perp}$, indicating that the function is both strict and total. Transformation yields four versions mapping between corresponding subtypes, as shown in Fig. 4. In this case, the transformed functions connect corresponding types in the two lattices because the abstract function was the identity. Other abstract functions will produce different patterns.

The commuting diagram of Fig. 4 is typical of the general situation, for the families of versions will be natural transformations:

$$\mathsf{C}_{s\to t}[\varphi] = \mathsf{C}_s \stackrel{.}{\to} \mathsf{C}_t \circ \varphi$$

That is, the type $\mathsf{C}_{s\to t}[\varphi]$ consists of families of functions $f_\alpha : \mathsf{C}_s[\alpha] \to \mathsf{C}_t[\varphi\ \alpha]$ for each $\alpha \in \mathcal{D}^{\#}_s$, satisfying

$$f_{\alpha'} \circ \mathsf{C}_s[\alpha \subseteq^{\#} \alpha'] = \mathsf{C}_t[\varphi\ \alpha \subseteq^{\#} \varphi\ \alpha'] \circ f_\alpha \tag{5}$$

However, we stipulate that $\mathsf{C}_{s\to t}[\varnothing^{\#}_{s\to t}]$ is empty.

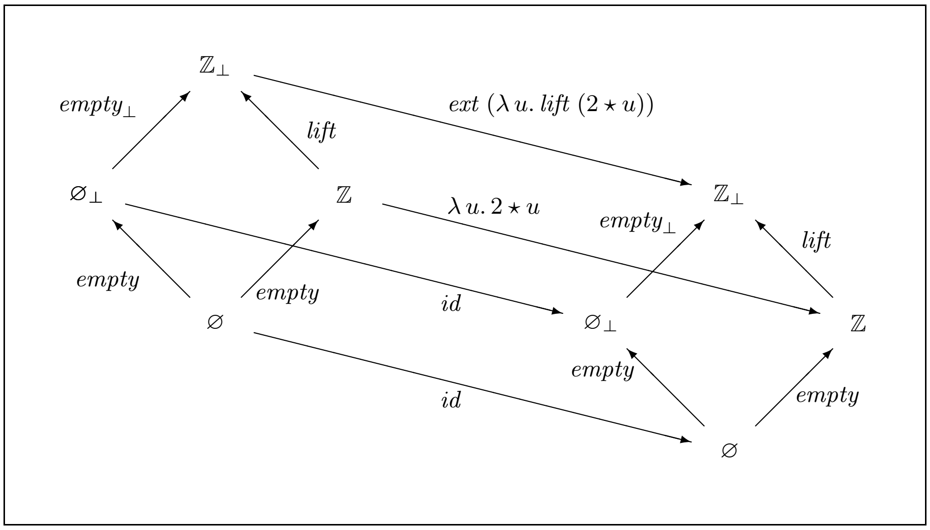

Fig. 4. Transformed versions of $ext\ (\lambda\, u.\, lift\ (2 \star u)) : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$

When such a natural transformation f exists, the instance $f_{\mho^\#}$ determines all the others, because each function $\mathsf{C}_t[\varphi\ \alpha \subseteq^\# \varphi\ \alpha']$ is injective. However the existence of $f_{\mho^\#}$ does not guarantee the existence of the natural transformation; our algorithm will construct the whole transformation.

These families will be implemented by iterated products of function types. It is important to note that the abstract domains $\mathcal{D}_t^\#$ will not appear in the transformed program. Selection of a component f_α will be represented by extraction of a known position in the tuple of functions. Unless f is a variable, this extraction can be done statically.

If $\varphi \subseteq^\# \varphi'$ in the pointwise ordering, then there is a corresponding injection $\mathsf{C}_{s\to t}[\varphi \subseteq^\# \varphi']$ between the natural transformations, defined by:

$$(\mathsf{C}_{s\to t}[\varphi \subseteq^\# \varphi']\ f)_\alpha = \mathsf{C}_t[\varphi\ \alpha \subseteq^\# \varphi'\ \alpha] \circ f_\alpha$$

It is straightforward to verify that

- $\mathsf{C}_{s\to t}[\varphi \subseteq^\# \varphi']\ f$ is natural,
- $\mathsf{C}_{s\to t}[\varphi \subseteq^\# \varphi']$ is bimonotone, and
- $\mathsf{C}_{s\to t}$ is a functor.

With the introduction of function types, the type $\mathsf{C}_t[\mho_t^\#]$ is no longer identical to t. However, it is isomorphic to t.

Proposition 8. *For each type t, there is an isomorphism $i_t : \mathsf{C}_t[\mho_t^\#] \cong t$.*

Proof. The required isomorphism is defined by induction over t. This is straightforward except for the case $s \to t$, for which we define

$$i_{s\to t}\ g = i_t \circ g_{\mho^\#} \circ i_s^{-1}$$

$$(i_{s\to t}^{-1}\ f)_\alpha = \begin{cases} \mathit{empty}_{s\to t} & \text{if } \alpha = \varnothing_s^\# \\ i_t^{-1} \circ f \circ i_s \circ \mathsf{C}_s[\alpha \subseteq^\# \mho_s^\#] & \text{otherwise} \end{cases}$$

where $\mathit{empty}_{s\to t}$ is the unique element of $\mathsf{C}_s[\varnothing_s^\#] \to \mathsf{C}_t[\varnothing_t^\#]$.

Then for $g : \mathsf{C}_{s\to t}[\mho_{s\to t}^\#]$ and $\alpha \neq \varnothing^\#$ (the empty case is trivial),

$$\begin{aligned} (i_{s\to t}^{-1}\ (i_{s\to t}\ g))_\alpha &= i_t^{-1} \circ i_t \circ g_{\mho^\#} \circ \mathsf{C}_s[\alpha \subseteq^\# \mho_s^\#] \circ i_s^{-1} \circ i_s \\ &= g_{\mho^\#} \circ \mathsf{C}_s[\alpha \subseteq^\# \mho_s^\#] && \text{by assumption} \\ &= \mathsf{C}_t[\mho_{s\to t}^\#\ \alpha \subseteq^\# \mho_{s\to t}^\#\ \mho_s^\#] \circ g_\alpha && \text{by (5)} \\ &= \mathsf{C}_t[\mho_t^\# \subseteq^\# \mho_t^\#] \circ g_\alpha \\ &= g_\alpha \end{aligned}$$

The reverse equality is demonstrated by instantiation. □

Thus for each abstract value $\alpha \in \mathcal{D}_t^\#$ there is an embedding $\alpha^{\mathrm{E}} : \mathsf{C}_t[\alpha] \to t$ defined by $\alpha^{\mathrm{E}} = i_t \circ \mathsf{C}_t[\alpha \subseteq^\# \mho_t^\#]$. The image of this function α^{E} is the property used in the conventional interpretation.

5 Abstract Interpretation and Transformation

Abstract interpretation of a closed expression $e : t$ will yield a value $e^\# \in \mathcal{D}_t^\#$, representing the type of the transformed expression. Transformation of e will yield an expression $e^\star : \mathsf{C}_t[e^\#]$. The transformation will be considered correct if

$$e = (e^\#)^{\mathrm{E}}\ (e^\star) \tag{6}$$

so that the behaviour of the original expression e can be recovered from the transformed version. The correctness condition is extended to expressions with free variables in Sect. 5.2.

The following lemma is used frequently in the proofs.

Lemma 9. *The condition $f = \varphi^{\mathrm{E}}\ f'$ is equivalent to*

$$\forall\, \alpha \in \mathcal{D}_s^\#.\ f \circ \alpha^{\mathrm{E}} = (\varphi\ \alpha)^{\mathrm{E}} \circ {f'}_\alpha$$

The forward implication is established by expanding the definitions and simplifying. The reverse direction follows by instantiating α to $\mho_s^\#$.

We consider first the abstract interpretation and transformation of the constants.

5.1 Translation of Constants

For each constant $c : t$, we shall define an abstract constant $c^\# \in \mathcal{D}_t^\#$ and a translated constant $c^\star : \mathsf{C}_t[c^\#]$ satisfying

$$c = (c^\#)^{\mathrm{E}}\ (c^\star) \tag{7}$$

In most cases this property is easily verified.

The definitions of the abstract constants $c^{\#} \in \mathcal{D}^{\#}_t$ are mostly standard (e.g. in [7]). The constant *empty* is new, but its transformation is straightforward:

$$\begin{aligned} \mathit{empty}^{\#}\ \alpha &= \varnothing^{\#}_t \\ \mathit{empty}^{\star}_{\alpha} &= \mathit{empty} \end{aligned}$$

The product constants are translated to themselves:

$$\begin{array}{rclrclrcl} \mathit{fst}^{\#}(\alpha \boxtimes \beta) &=& \alpha & \mathit{snd}^{\#}(\alpha \boxtimes \beta) &=& \beta & \mathit{pair}^{\#}\ \alpha\ \beta &=& \alpha \boxtimes \beta \\ \mathit{fst}^{\star}_{\alpha\boxtimes\beta} &=& \mathit{fst} & \mathit{snd}^{\star}_{\alpha\boxtimes\beta} &=& \mathit{snd} & (\mathit{pair}^{\star}_{\alpha}\ x)_{\beta}\ y &=& \mathit{pair}\ x\ y \end{array}$$

Injections into sums can only produce values in one arm of the sum:

$$\begin{array}{rclrcl} \mathit{inl}^{\#}\ \alpha &=& \alpha \boxplus \varnothing^{\#} & \qquad \mathit{inr}^{\#}\ \beta &=& \varnothing^{\#} \boxplus \beta \\ \mathit{inl}^{\star}_{\alpha} &=& \mathit{inl} & \mathit{inr}^{\star}_{\beta} &=& \mathit{inr} \end{array}$$

We must combine the results of the two arms of a *choose*:

$$\begin{aligned} &\mathit{choose}^{\#}\ \varphi\ \psi\ (\alpha \boxplus \beta) = \varphi\ \alpha \cup^{\#} \psi\ \beta \\ &\left((\mathit{choose}^{\star}_{\varphi}\ f)_{\psi}\ g\right)_{\alpha\boxplus\beta} = \mathit{choose}\ (\mathsf{C}_t[\varphi\ \alpha \subseteq^{\#} \gamma] \circ f_{\alpha})\ (\mathsf{C}_t[\psi\ \beta \subseteq^{\#} \gamma] \circ g_{\beta}) \\ &\qquad\qquad\qquad\qquad \text{where } \gamma = \varphi\ \alpha \cup^{\#} \psi\ \beta \end{aligned}$$

Note that if α is $\varnothing^{\#}$, then since φ is strict $\gamma = \psi\ \beta$ and the left branch is transformed to $\lambda\, x^{\varnothing}.\, e = \mathit{empty}$. The program will then contain types $\varnothing + t$. We assume that a subsequent pass will reduce these, eliminating associated injections and *choose*s.

Failure can only produce the bottom element of lifting:

$$\begin{aligned} \mathit{fail}^{\#}\ \alpha &= {\varnothing^{\#}}_{\boxed{\bot}} \\ \mathit{fail}^{\star}_{\alpha} &= \mathit{empty}_{\bot} \end{aligned}$$

Applications of *lift* are delayed:

$$\begin{aligned} \mathit{lift}^{\#}\ \alpha &= \alpha_{\Box} \\ \mathit{lift}^{\star}_{\alpha} &= \mathit{id} \end{aligned}$$

For evaluation, there are two possibilities. If the expression to be evaluated is already unboxed, no evaluation is required:

$$\begin{aligned} \mathit{ext}^{\#}\ \varphi\ \alpha_{\Box} &= \varphi\ \alpha \\ (\mathit{ext}^{\star}_{\varphi}\ f)_{\alpha_{\Box}} &= f_{\alpha} \end{aligned}$$

If the argument is boxed, the evaluation remains, and we must ensure that its result type is pointed:

$$\begin{aligned} \mathit{ext}^{\#}\ \varphi\ \alpha_{\boxed{\bot}} &= \varphi\ \alpha \cup^{\#} \bot^{\#} \\ (\mathit{ext}^{\star}_{\varphi}\ f)_{\alpha_{\boxed{\bot}}} &= \mathit{ext}\ (\mathsf{C}_t[\varphi\ \alpha \subseteq^{\#} \gamma] \circ f_{\alpha})\ \text{where } \gamma = \varphi\ \alpha \cup^{\#} \bot^{\#} \end{aligned}$$

where for each pointed type t, $\perp^\#_t$ is the least element of $\mathcal{D}^\#_t$ such that the image of ${\perp^\#_t}^{\mathrm{E}}$ contains the bottom element of t. We can define $\perp^\#_t$ by the following rules:

$$
\begin{aligned}
\perp^\#_{s\times t} &= \perp^\#_s \boxtimes \perp^\#_t \\
\perp^\#_{t_\perp} &= {\varnothing^\#}_{\boxed{\scriptstyle\perp}} \\
\perp^\#_{s\to t}\ \alpha &= \perp^\#_t \text{ for } \alpha \neq \varnothing^\#_s
\end{aligned}
$$

For example, $\perp^\#_{\mathbb{Z}_\perp} = {\varnothing^\#}_{\boxed{\scriptstyle\perp}}$, whose concretion is the one point type $\varnothing_\perp$. In general, for any pointed type t, $\perp^\#_t \in \mathcal{D}^\#_t$ abstracts a one point type, that point being mapped by ${\perp^\#_t}^{\mathrm{E}}$ to the bottom element of t.

Recursion also requires a pointed type:

$$
\begin{aligned}
\mathit{fix}^\#\ \varphi &= \bigcup^{\#}_{n\in\omega} {\varphi'}^n(\varnothing^\#) \text{ where } \varphi'\ \alpha = \varphi\ \alpha \cup^\# \perp^\# \\
\mathit{fix}^\star_\varphi\ f &= \mathit{fix}\ (\mathsf{C}_t[\varphi\ \gamma \subseteq^\# \gamma] \circ f_\gamma) \text{ where } \gamma = \mathit{fix}^\# \varphi
\end{aligned}
$$

By Lemma 9, the instance of (7) for fix is equivalent to

$$
\begin{aligned}
\forall\, \varphi \in \mathcal{D}^\#_{t\to t}.\, \forall\, f : \mathsf{C}_{t\to t}[\varphi].\, \mathit{fix}\ (\varphi^{\mathrm{E}}\ f) &= (\mathit{fix}^\#\ \varphi)^{\mathrm{E}}\ (\mathit{fix}^\star_\varphi\ f) \\
&= \gamma^{\mathrm{E}}\ (\mathit{fix}\ (\mathsf{C}_t[\varphi\ \gamma \subseteq^\# \gamma] \circ f_\gamma))
\end{aligned} \tag{8}
$$

Since $\perp^\# \subseteq^\# \gamma$, we know that γ^{E} is strict, and hence (8) follows from

$$
f \circ \gamma^{\mathrm{E}} = (\varphi\ \gamma)^{\mathrm{E}} \circ {f'}_\gamma = \gamma^{\mathrm{E}} \circ \mathsf{C}_t[\varphi\ \gamma \subseteq^\# \gamma] \circ {f'}_\gamma
$$

As noted in Sect. 3.1, our definition of $\mathit{fix}^\#$ differs from that of the Cousots [2], who use the sequence $\varphi^n(\perp^\#)$, which is a chain under the computational ordering. In contrast, we seek the smallest strict property for recursive functions.

The translation of a base constant $c^{B_1\to\cdots B_n\to B}$ is

$$
\begin{aligned}
(c^{B_1\to\cdots B_n\to B})^\#\ \alpha_1\ \ldots\ \alpha_n &= \mho^\#_{B_1\to\cdots B_n\to B} \\
(\ldots((c^{B_1\to\cdots B_n\to B})^\star_{\alpha_1}\ x_1)\ \ldots)_{\alpha_n}\ x_n &= c^{B_1\to\cdots B_n\to B}\ ({\alpha_1}^{\mathrm{E}}\ x_1)\ \ldots\ ({\alpha_n}^{\mathrm{E}}\ x_n)
\end{aligned}
$$

Here each α_i is either $\mho^\#_{B_i}$, representing B_i, or $\varnothing^\#_{B_i}$, representing $\varnothing$. Hence each ${\alpha_i}^{\mathrm{E}}$ is either *id* or *empty*. The application of the constant will only produce a result if all of its arguments exist.

5.2 Transformation of Expressions

An expression $e : t$ with free variables $x^{t_1}, \ldots, x^{t_n}$ may be considered as a function of type $t_1 \times \cdots \times t_n \to t$. Hence we define the set $\mathsf{Env}^\#$ of abstract environments as the domain of mappings of variables x^t to abstract values from $\mathcal{D}^\#_t$, such that if any variable is mapped to $\varnothing^\#$, they all are.

For any variable x^t, let $x^\star$ denote a family of variables indexed by $\mathsf{Env}^\#$, such that for each $\eta \in \mathsf{Env}^\#$, $x^\star$ has type $\mathsf{C}_t[\eta\ x^t]$.

Definition 10. If $e : t$, we write $e \overset{e^\#}{\Longrightarrow} e^\star$ to mean

1. $e^\# : \mathsf{Env}^\# \to \mathcal{D}_t^\#$ maps the empty environment to $\varnothing^\#$ and is monotonic with respect to both $\subseteq^\#$ and $\sqsubseteq^\#$. (That is, $e^\#$ is an abstract interpretation of e.)
2. $e^\star$ is a family of expressions $e^\star_\eta : \mathsf{C}_t[e^\#\ \eta]$ indexed by environments $\eta \in \mathsf{Env}^\#$, such that if $\eta \subseteq^\# \eta' \in \mathsf{Env}^\#$, then
$$\bigwedge_{x^s} \left(x^\star_{\eta'} = \mathsf{C}_s[\eta_x \subseteq^\# \eta'_x](x^\star_\eta)\right) \Rightarrow e^\star_{\eta'} = \mathsf{C}_t[e^\#\eta \subseteq^\# e^\#\eta']\ (e^\star_\eta)$$
(That is, $e^\star$ is a natural transformation.)
3. For each $\eta \in \mathsf{Env}^\#$,
$$\bigwedge_{x^s} \left(x = {\eta_x}^{\mathrm{E}}\ (x^\star_\eta)\right) \Rightarrow e = (e^\#\ \eta)^{\mathrm{E}}\ (e^\star_\eta)$$
(That is, the combination of abstract interpretation and transformation is correct.)

From these definitions it is easy to prove the following result, which enables us to transform any λ-expression.

Theorem 11.

1. *If a constant c satisfies $c = (c^\#)^E\ (c^\star)$, then $c \overset{\overline{c^\#}}{\Longrightarrow} \overline{c^\star}$ where $\overline{c^\#}\ \eta = c^\#$ and $\overline{c^\star}_\eta = c^\star$.*
2. *$x \overset{x^\#}{\Longrightarrow} x^\star$ where $x^\#\ \eta = \eta_x$*
3. *If $e \overset{e^\#}{\Longrightarrow} e^\star$, then $\lambda\, x.\, e \overset{f^\#}{\Longrightarrow} f^\star$, where*
$$\begin{aligned} f^\#\ \eta\ \alpha &= e^\#\ \eta[x \mapsto \alpha] \\ (f^\star_\eta)_\alpha &= \lambda\, x^\star_\eta.\, e^\star_{\eta[x\mapsto\alpha]} \end{aligned}$$
4. *If $e_1 : s \to t$, $e_2 : s$, $e_1 \overset{e_1^\#}{\Longrightarrow} e_1^\star$ and $e_2 \overset{e_2^\#}{\Longrightarrow} e_2^\star$, then $e_1\ e_2 \overset{e^\#}{\Longrightarrow} e^\star$, where*
$$\begin{aligned} e^\#\ \eta &= (e_1^\#\ \eta)\ (e_2^\#\ \eta) \\ e^\star_\eta &= ({e_1^\star}_\eta)_{e_1^\#\,\eta}\ ({e_2^\star}_\eta) \end{aligned}$$

Example 12. For the function
$$\begin{aligned} &\mathit{thrice} : (t \to t) \to t \to t \\ &\mathit{thrice}\ f\ x \triangleq f\ (f\ (f\ x)) \end{aligned}$$
the transformed version is
$$\begin{aligned} \mathit{thrice}^\#\ \varphi\ \alpha\ &= \varphi\ (\varphi\ (\varphi\ \alpha)) \\ (\mathit{thrice}^\star_\varphi\ f)_\alpha\ x &\triangleq f_{\varphi(\varphi\ \alpha)}\ (f_{\varphi\ \alpha}\ (f_\alpha\ x)) \end{aligned}$$
Note the use of three (possibly) different transformations of f.

Sometimes we will prefer weaker translations than those defined above. For example, if the above translation is applied to the term $(\lambda\, x.\, (x, x)) \circ \mathit{lift}$, the *lift* will be duplicated. If these *lift*s are not subsequently consumed by the transformation, the resulting program might generate two thunks where formerly there was one. The detection of such situations requires a linearity analysis (as with deforestation in similar situations [13]). Such an analysis is beyond the scope of this paper, but we can offer the following result, which guarantees the correctness of such weakenings.

Proposition 13. *If* $e : t$, $e \overset{e^\#}{\Longrightarrow} e^\star$ *and* $e^\# \subseteq^\# e^{\#\prime} \in \mathsf{Env}^\# \to \mathcal{D}_t^\#$, *then* $e \overset{e^{\#\prime}}{\Longrightarrow} e^{\star\prime}$, *where*

$$(e^{\star\prime})_\eta = \mathsf{C}_t[e^\# \ \eta \subseteq^\# e^{\#\prime} \ \eta] \ (e^\star_\eta)$$

Example 14. Suppose $e : s_\perp \times t$ and we wish to force the transformed version to use a lifted type for the first component. Then we can use this proposition with

$$e^{\#\prime} \ \eta = e^\# \ \eta \cup^\# \perp^\# \boxtimes \mho^\#$$

6 Conclusion

We have proved the correctness of a source-to-source transformation intended to remove thunks from the source program. The transformation of functions is polyvariant, with functions mapped to natural transformations, and is able to deal with higher order functions in a clean way. Such transformations are made possible by a source language in which lifting is explicit, and not all types need have a bottom element [8].

There are several areas in which this work could be extended.

Polymorphism. Our transformation assumed a monotyped language. It appears that many polymorphic functions, such as *thrice* above, could be handled by applying our rules to symbolic expressions representing abstract values.

Efficiency of the analysis. The abstract domains of even basic comportment analysis are rather large, but many of the points they contain contribute little to the quality of the transformation. It would be useful to identify useful subdomains. Similarly, one might like to cut down the sizes of the families of function variants used.

Quality of the generated program. We have said nothing about pragmatic issues, not even a guarantee that the transformation will improve the program. However we have given a framework (Prop. 13) within which such choices may be made while maintaining the correctness of the resulting transformation.

Recursive types. It is relatively straightforward to extend this work to (purely positive) retractive types, like the type of finite, partial and infinite lists. Adding initial types, like finite lists, would provide opportunities for further optimizations. However some care would be required: sometimes evaluating the large data structure early (rather than on demand) may be wasteful of space.

References

1. Geoffrey L. Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on a global program analysis. In *Fifth International Symposium on Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 346–364, Tallinn, Estonia, August 1993. Springer.
2. Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *International Conference on Computer Languages*, pages 95–112. IEEE Computer Society Press, 1994.
3. Olivier Danvy and John Hatcliff. CPS-transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1992.
4. G.W. Hamilton. Higher order deforestation. In *Programming Languages: Implementations, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 1996.
5. A. Reza Haydarlou and Pieter H. Hartel. Thunk lifting: Reducing heap usage in an implementation of a lazy functional language. *Journal of Functional and Logic Programming*, 1995.
6. Brian T. Howard. Inductive, projective, and pointed types. In *ACM International Conference on Functional Programming*, pages 102–109, Philadelphia, May 1996.
7. Sebastian Hunt and David Sands. Binding time analysis: A new PERspective. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 154–165. ACM SIGPLAN Notices 26(9), September 1991.
8. John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 204–218, Linköping, Sweden, April 1996. Springer.
9. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In B. Robinet, editor, *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 1980.
10. Alan Mycroft and Kirsten Lackner Solberg. Uniform pers and comportment analysis. In *Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 1995.
11. Ross Paterson. Compiling laziness using projections. In *Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 255–269, Aachen, Germany, September 1996. Springer.
12. Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, Cambridge, MA, 1991.
13. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
14. Philip Wadler and John Hughes. Projections for strictness analysis. In *Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407, Portland, OR, 1987. Springer.