Cps-Translation and the Correctness of Optimising Compilers

Geoffrey Burn* and Daniel Le Métayer†
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom.
{glb,dlm}@doc.ic.ac.uk

Submitted for publication. In the meantime, please refer to as Imperial College, Department of Computing Technical Report number DoC92/20.

Abstract

We show that compiler optimisations based on strictness analysis can be expressed formally in the functional framework using continuations. This formal presentation has two benefits: it allows us to give a rigorous correctness proof of the optimised compiler; and it exposes the various optimisations made possible by a strictness analysis. These benefits are especially significant in the presence of partially evaluated data structures.

1 Introduction

Realistic compilers for imperative or functional languages include a number of optimisations based on non-trivial global analyses. Proving the correctness of such optimising compilers should involve three steps:

1. proving the correctness of the original (unoptimised) compiler;
2. proving the correctness of the analysis; and
3. proving the correctness of the modifications of the simple-minded compiler to exploit the results of the analysis.

A substantial amount of work has been devoted to steps (1) and (2) but there has been surprisingly few attempts at tackling step (3). In this paper we propose a method to carry out this third step in the context of optimising compilers for functional languages which use the results of 'strictness' analysis.

There are two ways we might want to use strictness information in compiling lazy functional languages:

- evaluating an argument expression instead of passing it as an unevaluated closure; and
- compiling functions which know their arguments have been evaluated, so that the argument can be passed explicitly, rather than as a closure containing a value in the heap (i.e. 'unboxed' rather than 'boxed').

*This author is partially funded by ESPRIT BRA 3124 (Semantique) and SERC grant GR/H 17381 ("Using the Evaluation Transformer Model to make Lazy Functional Languages more Efficient").

†This author is on leave from INRIA/IRISA and is partially funded by the SERC Visiting Fellowship GR/H 19330.

Translating programs into continuation-passing style (cps) allows us to express both uses of strictness information because:

- a cps-translation captures the evaluation order of expressions; and
- a closure is essentially a value waiting for a continuation which uses it.

The main results of this paper are three cps-conversions which use strictness information, each of which has been proved correct. We start by showing how simple strictness information can be used to change the evaluation order (Section 2.1). This is then extended in two orthogonal ways: firstly we give a cps-conversion where functions can be compiled knowing that some of their arguments have been evaluated (Section 2.2); and secondly we express how the evaluation order can be changed in more complicated ways for structured data types such as lists (Section 3).

A consequence of the second cps-translation, described in Section 2.2, is that the translation of the types makes it explicit whether or not an (evaluated) argument is being passed in a closure in the heap (i.e. whether or not it is 'boxed'). This appears to be a natural alternative to that given in [JL91].

In the translation rules, we state what properties must hold in order to use particular rules. Safe approximations to these properties can be determined using established program analyses, as discussed in Section 4.

The cps-conversions we describe in this paper can be used in the context of [FM91], where a complete compiler is proved correct. We can therefore demonstrate the correctness of a complete compiler which uses strictness information.

A survey of related work can be found in Section 5, and Section 6 reviews the benefits of this approach and identifies areas of further research.

2 Using Simple Strictness Information

Figures 1 and 2 describe the syntax of our functional language and its semantics.

Our starting point is an adaptation of the compiler described in [FM91]. The key feature of this compiler is the fact that it is described entirely within the functional framework as a succession of transformations. This makes its correctness proof easier to establish.

We need only consider the first step of the compiler here, which is the call-by-name cps-transformation, given in Figure 3. The transformation captures the call-by-name computation rule because the translation of an application indicates that the argument is passed unevaluated to the function. The important point about $\mathcal{N}\ [\![E]\!]$ is that it has at most one redex outside the scope of a lambda, which means that call-by-value and call-by-name coincide for the translated term [Plo75]. Furthermore, this redex is always at the head of the expression [FM91], and the expression can be reduced without dynamic search for the next redex, just like machine code.

We have left the types off the translated terms for clarity. **Ans** is the type of answers. The result of translating an expression of type σ is an expression of type $\mathsf{B}\ [\![\sigma]\!] = \mathsf{C}\ [\![\sigma]\!] \rightarrow \mathbf{Ans}$. This can be stated formally by Theorem 2.2.

Definition 2.1 *If ρ is a type environment, then its transformation $\mathcal{N}\ [\![\rho]\!]$ is defined by the rule:*

$$\frac{\rho \vdash x : \sigma}{\mathcal{N}\ [\![\rho]\!] \vdash x : \mathsf{B}\ [\![\sigma]\!]}$$

Theorem 2.2

$$\frac{\rho \vdash E : \sigma}{\mathcal{N}\ [\![\rho]\!] \vdash \mathcal{N}\ [\![E]\!] : \mathsf{B}\ [\![\sigma]\!]}$$

The set T of types is the least set defined by:

$$\{bool, int\} \subseteq T$$
$$\sigma, \tau \in T \Rightarrow (\sigma \rightarrow \tau) \in T$$
$$\sigma \in T \Rightarrow (list\ \sigma) \in T$$

The type system of Λ_T

$$(1)\quad x^\sigma\ :\ \sigma \qquad\qquad (2)\quad \mathbf{k}_\sigma\ :\ \sigma$$

$$(3)\quad \frac{E_1\ :\ \sigma\rightarrow\tau,\ E_2\ :\ \sigma}{(E_1\ E_2)\ :\ \tau} \qquad\qquad (4)\quad \frac{E\ :\ \tau}{(\lambda x^\sigma.E)\ :\ \sigma\rightarrow\tau}$$

$$(5)\quad \frac{E\ :\ \sigma\rightarrow\sigma}{\mathbf{fix}_\sigma\ E\ :\ \sigma}$$

Abstract Syntax of Λ_T

$\mathbf{true}_{bool}$	$\mathbf{false}_{bool}$	$\mathbf{if}_{bool\rightarrow\sigma\rightarrow\sigma\rightarrow\sigma}$
$\{\mathbf{0}_{int}, \mathbf{1}_{int}, \mathbf{2}_{int}, \ldots\}$	$\mathbf{plus}_{int\rightarrow int\rightarrow int}$	$\mathbf{head}_{list\ \sigma\rightarrow\sigma}$
$\mathbf{nil}_{list\ \sigma}$	$\mathbf{cons}_{\sigma\rightarrow list\ \sigma\rightarrow list\ \sigma}$	$\mathbf{tail}_{list\ \sigma\rightarrow list\ \sigma}$

The Constants of Λ_T

Figure 1: Definition of the Language Λ_T

$$\begin{array}{lcl} \mathbf{S}_B & = & \text{some domain for the base type } B \\ \mathbf{S}_{\sigma\rightarrow\tau} & = & \mathbf{S}_\sigma \rightarrow \mathbf{S}_\tau \\ \mathbf{S}_{(list\sigma)} & = & List\ \mathbf{S}_\sigma \end{array}$$

Semantics of the Types

$$\mathbf{S}\ [\![x^\sigma]\!]\ \rho^{\mathbf{S}} = \rho^{\mathbf{S}}\ x^\sigma$$
$$\mathbf{S}\ [\![\mathbf{k}_\sigma]\!]\ \rho^{\mathbf{S}} = \mathbf{K}^{\mathbf{S}}\ [\![\mathbf{k}_\sigma]\!]$$
$$\mathbf{S}\ [\![E_1\ E_2]\!]\ \rho^{\mathbf{S}} = (\mathbf{S}\ [\![E_1]\!]\ \rho^{\mathbf{S}})\ (\mathbf{S}\ [\![E_2]\!]\ \rho^{\mathbf{S}})$$
$$\mathbf{S}\ [\![\lambda x^\sigma.E]\!]\ \rho^{\mathbf{S}} = \lambda d \epsilon \mathbf{S}_\sigma.\mathbf{S}\ [\![E]\!]\ \rho^{\mathbf{S}}[d/x^\sigma]$$
$$\mathbf{S}\ [\![\mathbf{fix}_\sigma\ E]\!]\ \rho^{\mathbf{S}} = \bigsqcup_{i\geq 0} (\mathbf{S}\ [\![E]\!]\ \rho^{\mathbf{S}})^i\ \bot_{\mathbf{S}_\sigma}$$

Semantics of the Language Terms

Figure 2: The Semantics of Λ_T

$$
\begin{array}{lcll}
\mathsf{U}\,[\![int]\!] & = & int & \text{— unboxed values}\\
\mathsf{U}\,[\![bool]\!] & = & bool & \\
\mathsf{U}\,[\![\sigma \to \tau]\!] & = & \mathsf{C}\,[\![\tau]\!] \to \mathsf{B}\,[\![\sigma]\!] \to \mathbf{Ans} & \\
\mathsf{U}\,[\![(list\ \sigma)]\!] & = & (\mathsf{B}\,[\![\sigma]\!] \times \mathsf{B}\,[\![(list\ \sigma)]\!]) + nil & \\
\mathsf{C}\,[\![\sigma]\!] & = & \mathsf{U}\,[\![\sigma]\!] \to \mathbf{Ans} & \text{— continuations}\\
\mathsf{B}\,[\![\sigma]\!] & = & \mathsf{C}\,[\![\sigma]\!] \to \mathbf{Ans} & \text{— boxed values}
\end{array}
$$

Translation of Types

$$
\begin{array}{lcl}
\mathcal{N}\,[\![x]\!] & = & x\\
\mathcal{N}\,[\![0]\!] & = & \lambda c.c\ 0 \qquad \text{— and similarly for other integers and booleans}\\
\mathcal{N}\,[\![\mathbf{plus}]\!] & = & \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.\mathbf{plus_c}\ c_2\ m\ n))))\\
\mathcal{N}\,[\![\mathbf{if}\ E_1\ E_2\ E_3]\!] & = & \lambda c.\mathcal{N}\,[\![E_1]\!]\ (\mathbf{if_c}\ (\mathcal{N}\,[\![E_2]\!]\ c)\ (\mathcal{N}\,[\![E_3]\!]\ c))\\
\mathcal{N}\,[\![\mathbf{nil}]\!] & = & \lambda c.c\ \mathbf{nil}\\
\mathcal{N}\,[\![\mathbf{cons}]\!] & = & \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.\mathbf{cons_c}\ c_2\ x\ y))\\
\mathcal{N}\,[\![\mathbf{head}]\!] & = & \lambda c.c\ (\lambda c_1.\lambda x.x\ (\lambda v.\mathbf{head}\ v\ c_1))\\
\mathcal{N}\,[\![E_1\ E_2]\!] & = & \lambda c.\mathcal{N}\,[\![E_1]\!]\ (\lambda f.f\ c\ (\mathcal{N}\,[\![E_2]\!]))\\
\mathcal{N}\,[\![\lambda x.E]\!] & = & \lambda c.c\ (\lambda c.\lambda x.\mathcal{N}\,[\![E]\!]\ c)\\
\mathcal{N}\,[\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] & = & \mathbf{fix}_{\mathsf{B}[\![\sigma]\!]}\ (\lambda x.\mathcal{N}\,[\![E]\!])
\end{array}
$$

Translation of Terms

$$
\begin{array}{lcl}
\mathbf{plus_c}\ c\ m\ n & = & c\ (\mathbf{plus}\ m\ n)\\
\mathbf{if_c}\ E_1\ E_2 & = & \lambda v.\mathbf{if}\ v\ E_1\ E_2\\
\mathbf{cons_c}\ c\ E_1\ E_2 & = & c\ (\mathbf{cons}\ E_1\ E_2)
\end{array}
$$

Figure 3: The Call-by-Name Cps-conversion

Expressions of type $\mathsf{C}\,[\![\sigma]\!]$ are continuations: they take the result of evaluating an expression of type $\mathsf{U}\,[\![\sigma]\!]$ into an answer. Meyer and Wand first showed that the type of the cps-translation of an expression could be derived from the type of the original expression [MW85].

The rule for application differs slightly from the usual presentation of the cps-translation [Rey74, Plo75]: the continuation is passed as the first argument to a function rather than as the second argument (as was done in [Fis72]). This change is motivated by the subsequent steps of the compiler. We stress that the work of this paper is independent of this change of argument order, but has been presented in this way so that our results can be fed into those of [FM91] and so have an optimised compiler whose correctness has been proved.

Let us take a small example to illustrate this transformation and expose the potential sources of inefficiency:

$$
\begin{array}{c}
F = \lambda x.(\mathbf{plus}\ x\ 1)\\
E = F\ (\mathbf{plus}\ 2\ 7).
\end{array}
$$

Applying the translation rules from Figure 3 yields, after reducing "administrative redexes" (terminology due to [Plo75]) introduced by continuations:

$$
\begin{array}{lcl}
\mathcal{N}\,[\![F]\!] & = & \lambda c.c\ F_1\\
F_1 & = & \lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1)\\
\mathcal{N}\,[\![E]\!] & = & \lambda c.F_1\ c\ (\lambda c.(\mathbf{plus_c}\ c\ 2\ 7)).
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S}\ [\![x]\!] & = & x \\
\mathcal{S}\ [\![\mathbf{k}_\sigma]\!] & = & \mathcal{N}\ [\![\mathbf{k}_\sigma]\!] \\
\mathcal{S}\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] & = & \lambda c.\mathcal{S}\ [\![E_1]\!]\ (\mathbf{if}_{\mathsf{c}}\ (\mathcal{S}\ [\![E_2]\!]\ c)\ (\mathcal{S}\ [\![E_3]\!]\ c)) \\
\mathcal{S}\ [\![E_1\ E_2]\!] & = & \lambda c.\mathcal{S}\ [\![E_2]\!]\ (\lambda v.\mathcal{S}\ [\![E_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
 & & \text{if } \forall \rho^{\mathsf{S}} : \mathbf{S}\ [\![E_1]\!]\ \rho^{\mathsf{S}}\ \perp = \perp \\
 & = & \lambda c.\mathcal{S}\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathcal{S}\ [\![E_2]\!])) \\
 & & \text{otherwise} \\
\mathcal{S}\ [\![\lambda x.E]\!] & = & \lambda c.c\ (\lambda c.\lambda x.\mathcal{S}\ [\![E]\!]\ c) \\
\mathcal{S}\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] & = & \mathbf{fix}_{\mathsf{B}[\![\sigma]\!]}\ (\lambda x.\mathcal{S}\ [\![E]\!])
\end{array}
$$

Translation of Terms

Figure 4: The Cps-conversion Using Simple Strictness Information

We note that F_1 is passed the unevaluated argument $(\lambda c.(\mathbf{plus}_{\mathsf{c}}\ c\ 2\ 7))$ which will be immediately evaluated in the body of F_1. This cost of passing an unevaluated argument may be significant (in terms of execution time as well as space consumption). A more efficient computation rule would be to evaluate the argument of the function before the call, provided this does not change the semantics of the program. This can be achieved in the cps-translation by first translating the argument expression and then the function, so that the translation of E becomes: $\lambda c.\mathbf{plus}_{\mathsf{c}}\ (\lambda v.F_1\ c\ (\lambda c.c\ v))\ 2\ 7$. This version is more efficient in terms of space consumption because the closure which is passed as an argument to F_1 now represents an evaluated argument. There is still room for improvement however because we have not exploited the fact that F_1 will be passed an evaluated closure in the compilation of F. Using this property we can replace F_1 in the body of the translation of F by $F_2 = \lambda c.\lambda x.\mathbf{plus}_{\mathsf{c}}\ c\ x\ 1$, and E by $\lambda c.\mathbf{plus}_{\mathsf{c}}\ (F_2\ c)\ 2\ 7$, which has the effect of passing the value 9 to F rather than the evaluated closure $(\lambda c.c\ 9)$.

It is also important to note that the types of the transformed terms give us significant information. The type of F_1 is

$$\mathsf{C}\ [\![int]\!] \to \mathsf{B}\ [\![int]\!] \to \mathbf{Ans}$$

$(= \mathsf{U}\ [\![int \to int]\!])$, whilst the type of F_2 is

$$\mathsf{C}\ [\![int]\!] \to \mathsf{U}\ [\![int]\!] \to \mathbf{Ans}.$$

In implementation terms, a value of type $\mathsf{B}\ [\![\sigma]\!]$ must be represented in the heap and accessed indirectly through the stack, whereas a term of type $\mathsf{U}\ [\![\sigma]\!]$ can be represented directly on the stack if σ is a basic type. This distinction has been called *boxed* versus *unboxed* representation in [JL91]. In our framework $\mathsf{B}\ [\![\sigma]\!]$ denotes a boxed implementation of σ and $\mathsf{U}\ [\![\sigma]\!]$ is an unboxed representation of σ, so that the 'boxedness' of a value can be determined from its type.

These optimisations are presented more formally in the next two subsections.

2.1 Changing the Evaluation Order

An improved cps-translation using simple strictness information is presented in Figure 4. We make the following observations about the rules:

- $\mathcal{N}\ [\![E]\!]$ and $\mathcal{S}\ [\![E]\!]$ have the same type, and a similar theorem to Theorem 2.2 can easily be proved.

- The key rule is the translation of application. There are two cases to consider:
  - when the functional expression is strict (the first rule), then the argument can be evaluated before the functional expression. In cps-conversion, this is expressed by putting the translation of the argument expression at the front of the converted expression. The continuation in this case picks up the value, wraps it into a closure $(\lambda c.c\ v)$ (i.e. boxes the value), and then proceeds to evaluate the functional expression as before.
  - when the functional expression is not strict (second rule), the translation has the same structure as the call-by-name cps conversion, but uses the $\mathcal{S}$ conversion scheme so that strictness information can be used in translating subexpressions.
- Apart from **if**, the conversion rules for the constants are identical to the call-by-name cps conversion, because the change of evaluation order that is allowed by these constants is captured by the general rule for function application. An exception is made for **if** because the evaluation context of the chosen alternative is the same as the evaluation context of the application of **if**, but the strictness information about **if** says that neither its second nor its third argument need to be evaluated.

The correctness of this translation is expressed by the following theorem. We do not prove it because it follows as a corollary of the more general translation presented in Section 3.

Theorem 2.3 *For all terms E:* $\mathsf{S}\ [\![\mathcal{S}\ [\![E]\!]]\!] = \mathsf{S}\ [\![\mathcal{N}\ [\![E]\!]]\!]$.

2.2 Unboxed Values

Looking at the two rules for application in Figure 4 we can see that E_1 is compiled in the same way in both cases. This is because the code for E_1 expects a closure as its argument; when the argument is evaluated before the call and returns the value v, a closure $\lambda c.c\ v$ has to be built to encapsulate this value. This could obviously be omitted provided that E_1 is compiled with the extra assumption that its argument is already evaluated. The rules in Figure 5 achieve this optimisation; two extra arguments are passed to the compilation function. I is the set of indices representing the evaluated arguments of the expression and V is the set of evaluated variables of the expression. Notice that each function may be compiled in several different ways, depending on the calling context. In particular, the operator **plus** can be compiled in four different ways.

Some functions are compiled when their application context is not known (for example, functions which are passed as arguments to another function, or functions in a list), but they may be applied in a context where their argument has been evaluated. When such a function is applied (either because it is the closure bound to a variable, or because it is the result of applying **head** to a list of functions), it has to be converted to take an unboxed argument. This is accomplished by the function $conv_I$, whose definition has been omitted for the sake of brevity.

The rule given for the fixed point operator in Figure 5 is the most precise translation possible, but is not effective in general, as it can lead to the production of infinite code. We prefer to separate the issues of exploiting strictness information in a systematic way from the necessary engineering decisions to be made for limiting the size of the code produced from a real compiler. We consider this second aspect in Section 4.

The correctness of this conversion follows from Theorem 2.3 (correctness of the $\mathcal{S}$ translation scheme) and Theorem 2.5.

Definition 2.4 *If ρ is a type environment, then its transformation $\mathcal{S}'_V\ [\![\rho]\!]$ is defined by the rule:*

$$\frac{\rho \vdash x : \sigma \quad x \in V}{\mathcal{S}'_V\ [\![\rho]\!] \vdash x : \mathsf{U}\ [\![\sigma]\!]} \quad \frac{\rho \vdash x : \sigma \quad x \notin V}{\mathcal{S}'_V\ [\![\rho]\!] \vdash x : \mathsf{B}\ [\![\sigma]\!]}$$

$$
\begin{array}{lcl}
\mathsf{U'}_I\ [\![int]\!] & = & int \\
\mathsf{U'}_I\ [\![bool]\!] & = & bool \\
\mathsf{U'}_I\ [\![\sigma \to \tau]\!] & = & \mathsf{C'}_{(dec\ I)}\ [\![\tau]\!] \to \mathsf{U}\ [\![\sigma]\!] \to \mathbf{Ans} \quad \text{if } 1 \in I \\
 & = & \mathsf{C'}_{(dec\ I)}\ [\![\tau]\!] \to \mathsf{B}\ [\![\sigma]\!] \to \mathbf{Ans} \quad \text{if } 1 \notin I \\
\mathsf{U'}_I\ [\![list\ \sigma]\!] & = & \mathsf{U}\ [\![list\ \sigma]\!] \\
\mathsf{C'}_I\ [\![\sigma]\!] & = & \mathsf{U'}_I\ [\![\sigma]\!] \to \mathbf{Ans} \\
\mathsf{B'}_I\ [\![\sigma]\!] & = & \mathsf{C'}_I\ [\![\sigma]\!] \to \mathbf{Ans}
\end{array}
$$

Translation of Types

$$
\begin{array}{lcl}
inc\ I & = & \{i+1 | i \in I\} \\
dec\ I & = & \{i-1 | i \in I \wedge i > 1\} \\
conv_I & : & \mathsf{B}\ [\![\sigma]\!] \to \mathsf{B'}_I\ [\![\sigma]\!]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S}'\ I\ V\ [\![x]\!] & = & conv_I\ (\lambda c.c\ x) \quad \text{if } x \in V \\
 & = & conv_I\ x \quad \text{if } x \notin V \\
\mathcal{S}'\ I\ V\ [\![\mathbf{0}]\!] & = & \mathcal{N}\ [\![\mathbf{0}]\!] \quad \text{— and similarly for other basic values} \\
\mathcal{S}'\ I\ V\ [\![\mathbf{plus}]\!] & = & \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.\mathbf{plus_c}\ c_2\ x\ y)) \\
 & & \text{if } 1 \in I \wedge 2 \in I \\
 & = & \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.\mathbf{plus_c}\ c_2\ m\ y))) \\
 & & \text{if } 1 \notin I \wedge 2 \in I \\
 & = & \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.y\ (\lambda m.\mathbf{plus_c}\ c_2\ x\ m))) \\
 & & \text{if } 1 \in I \wedge 2 \notin I \\
 & = & \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.\mathbf{plus_c}\ c_2\ m\ n)))) \\
 & & \text{if } 1 \notin I \wedge 2 \notin I \\
\mathcal{S}'\ I\ V\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] & = & \lambda c.\mathcal{S}'\ \emptyset\ V\ [\![E_1]\!]\ (\mathbf{if_c}\ (\mathcal{S}'\ I\ V\ [\![E_2]\!]\ c)\ (\mathcal{S}'\ I\ V\ [\![E_3]\!]\ c)) \\
\mathcal{S}'\ I\ V\ [\![\mathbf{cons}]\!] & = & \mathcal{N}\ [\![\mathbf{cons}]\!] \\
\mathcal{S}'\ I\ V\ [\![\mathbf{head}]\!] & = & \lambda c.c\ (\lambda c_1.\lambda x.(conv_{(decI)}\ (\mathbf{head}\ x))\ c_1) \\
 & & \text{if } 1 \in I \\
\mathcal{S}'\ I\ V\ [\![\mathbf{head}]\!] & = & \lambda c.c\ (\lambda c_1.\lambda x.x(\lambda v.(conv_{(decI)}\ (\mathbf{head}\ v))\ c_1)) \\
 & & \text{if } 1 \notin I \\
\mathcal{S}'\ I\ V\ [\![E_1\ E_2]\!] & = & \lambda c.\mathcal{S}'\ \emptyset\ V\ [\![E_2]\!]\ (\lambda v.\mathcal{S}'\ (1 \bigcup (inc\ I))\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ v)) \\
 & & \text{if } \forall \rho^{\mathsf{S}} : \mathbf{S}\ [\![E_1]\!]\ \rho^{\mathsf{S}}\ \perp = \perp \\
 & = & \lambda c.\mathcal{S}'\ (inc\ I)\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathcal{S}'\ \emptyset\ V\ [\![E_2]\!])) \\
 & & \text{otherwise} \\
\mathcal{S}'\ I\ V\ [\![\lambda x.E]\!] & = & \lambda c.c\ (\lambda c.\lambda x.\mathcal{S}'\ (dec\ I)\ (V \bigcup \{x\})\ [\![E]\!]\ c) \\
 & & \text{if } 1 \in I \\
\mathcal{S}'\ I\ V\ [\![\lambda x.E]\!] & = & \lambda c.c\ (\lambda c.\lambda x.\mathcal{S}'\ (dec\ I)\ (V \setminus \{x\})\ [\![E]\!]\ c) \\
 & & \text{if } 1 \notin I \\
\mathcal{S}'\ I\ V\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] & = & \mathcal{S}'\ I\ V\ [\![E[\mathbf{fix}_\sigma(\lambda x.E)/x]]\!]
\end{array}
$$

Translation of Terms

Figure 5: The cps-conversion Using Strictness and Evaluation Information

Theorem 2.5

$$\frac{\rho \vdash E : \sigma}{\mathcal{S}'_V\ [\![\rho]\!] \vdash \mathcal{S}'\ I\ V\ [\![E]\!] : \mathsf{B}'_I\ [\![\sigma]\!]}$$

It is easy to see that $\mathcal{S}'_\emptyset\ [\![\rho]\!] = \mathcal{N}\ [\![\rho]\!]$ and that $\mathsf{B}'_\emptyset\ [\![\sigma]\!] = \mathsf{B}\ [\![\sigma]\!]$.

3 Using More Complicated Strictness Information: Changing Evaluation Order

Just using strictness information misses many opportunities for optimisation because functions often require their arguments to be evaluated further than weak head normal form (WHNF). Moreover, the amount of evaluation required of an argument in an application may depend on the context of the application. For example, if an application of the function defined by:

$$append = \mathbf{fix}_{list\ \sigma}\ (\lambda f.\lambda x.\lambda y.\ \mathbf{if}\ (\mathbf{eq}\ x\ \mathbf{nil})\ y\ (\mathbf{cons}\ (\mathbf{hd}\ x)\ (f\ (\mathbf{tl}\ x)\ y)))$$

is in a context where the structure of the result of applying *append* is required, then the structures of both of its arguments are required. Other contexts may require different amounts of evaluation of the arguments.

We have found it useful to characterise an evaluation context by the set of terms whose evaluation would fail to terminate in that context. Such an evaluation context should have two properties:

- if the evaluation of some term fails to terminate, then the evaluation of all terms whose semantics is less defined than that term should fail to terminate; and
- if the evaluation of all expressions which approximate some term fails to terminate, then it should fail to terminate for the term itself.

Scott-closed sets capture denotationally the two properties that we require of an evaluation context. This was first noted explicitly in [Bur91a].

Definition 3.1 *(Scott-closed set) A set S is* Scott-closed *of a domain D if*

1. *it is down-closed, that is, if $\forall d \in D$ such that $\exists s \in S$ such that $d \sqsubseteq s$, then $d \in S$; and*
2. *if $X \subseteq S$ and X is directed, then $\bigsqcup X \in S$.*

We only consider non-empty Scott-closed sets in this paper. ■

We can now give an intuitive explanation of the key features of the transformation rules given in Figure 6. Their correctness is proved as Theorem 3.2.

The first two arguments to the transformation $\mathcal{T}$ have a specific meaning in isolation from each other, but they are really needed because of the way that they interact, so we will firstly explain their independent meaning and then their interaction.

- The first argument to the $\mathcal{T}$ rule counts how many argument expressions have been passed over in order to reach the expression currently being translated. When the translation of some subexpression begins, this index is set to 0 (and so it is 0 for the conversion of the initial term).
- The second argument to the $\mathcal{T}$ rule, Q, is the Scott-closed set representing the evaluation context of an expression.

Conversion of initial term $E : \sigma$ — $\mathcal{T}\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E]\!]$

$$
\begin{array}{lcl}
\mathcal{T}\ i\ Q\ [\![x]\!] & = & x \\
\mathcal{T}\ i\ Q\ [\![\mathbf{k}_\sigma]\!] & = & \mathcal{N}\ [\![\mathbf{k}_\sigma]\!] \\
\mathcal{T}\ i\ Q\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] & = & \lambda c.\mathcal{T}\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]\ (\mathbf{if_c}\ (\mathcal{T}\ i\ Q\ [\![E_2]\!]\ c)\ (\mathcal{T}\ i\ Q\ [\![E_3]\!]\ c)) \\
\mathcal{T}\ i\ Q\ [\![E_1\ E_2]\!] & = & \lambda c.\mathcal{T}\ 0\ P\ [\![E_2]\!]\ (\lambda v.\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
 & & \text{if } \forall \rho^{\mathbf{S}},\ \forall v_0 \in P,\ \forall v_1,\ \ldots,\ v_i : \mathbf{S}\ [\![E_1]\!]\ \rho^{\mathbf{S}}\ v_0\ v_1\ \ldots\ v_i \in Q \\
 & = & \lambda c.\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathcal{T}\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!])) \\
 & & \text{otherwise} \\
\mathcal{T}\ 0\ Q\ [\![\lambda x.E]\!] & = & \lambda c.c\ (\lambda c.\lambda x.\mathcal{T}\ 0\ \{\perp_{\mathbf{S}_\tau}\}\ [\![E]\!]\ c) \\
\mathcal{T}\ (i+1)\ Q\ [\![\lambda x.E]\!] & = & \lambda c.c\ (\lambda c.\lambda x.\mathcal{T}\ i\ Q\ [\![E]\!]\ c) \\
\mathcal{T}\ i\ Q\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] & = & \mathcal{T}\ i\ Q\ [\![E[\mathbf{fix}_\sigma\ (\lambda x.E)/x]]\!]
\end{array}
$$

Figure 6: The Cps-conversion Using Scott-closed Set Information

- We can motivate the way the evaluation context is passed inwards for the application and abstraction rules in the following way. The general form of an application is:

$$E = (\lambda x_1.\ldots\lambda x_j.[(\lambda y_1\ldots\lambda y_k.D)\ D_1\ \ldots\ D_m])\ E_1\ \ldots E_n.$$

  Suppose that we are calculating $\mathcal{T}\ 0\ Q\ [\![E]\!]$, and that $j = n$. Using the rule for application j times, and then the rule for lambda-abstraction j times, then part of the term from the translation of E will be:

$$\mathcal{T}\ 0\ Q\ [\![(\lambda y_1\ldots\lambda y_k.D)\ D_1\ \ldots\ D_m]\!],$$

  which says that the inner application is to be evaluated in the context given by Q. This corresponds to passing the evaluation context to a tail-call.

Now we can give some important intuitions about the rest of the translation:

- There are two rules for translating an application:
  - the first is where the argument can be evaluated before evaluating the function. The intuition is that $\mathcal{T}\ i\ Q\ [\![E]\!]$ may fail to terminate if the semantics of E applied to i arbitrary arguments gives a value in the set Q, so that $\mathcal{T}\ 0\ P\ [\![E_2]\!]$ may fail to terminate only if $\mathbf{S}\ [\![E_2]\!]\ \rho^{\mathbf{S}} \in P$. But the correctness proof will show that the condition of the translation rule guarantees that $\mathbf{S}\ [\![E_1\ E_2]\!]\ \rho^{\mathbf{S}}$ applied to i arbitrary values is in Q in this case, and so it was all right to fail to terminate.
  - in the second rule, assuming the E_1 has the type $\sigma\rightarrow\tau$, the only evaluation context that can be given for the argument E_2 is $\{\perp_{\mathbf{S}_\sigma}\}$ because, if the expression is ever evaluated, then it will be evaluated at least to WHNF, but it cannot be guaranteed that it will be evaluated any further.
- The same remarks as the ones for Figure 5 hold concerning the fixed point operator.

The following theorem gives the correctness of our translation. It states that translating a term with $\mathcal{T}$ gives essentially the same result as translating it with $\mathcal{N}$. By essentially the same, we mean they have the same semantics. The theorem follows as an easy corollary of Theorem 3.5.

Theorem 3.2 *For all expressions $E : \sigma$,* $\mathbf{S}\ [\![\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!] = \mathbf{S}\ [\![\mathcal{N}\ [\![E]\!]]\!]$.

In the following definition, $\mathbf{FV}(e_1, \ldots, e_n)$ is the set of free variables of e_1 to e_n; $\Uparrow(E)$ means that the (call-by-name) reduction of E diverges; $\Downarrow(E)$ means that the (call-by-name) reduction of E converges; and $E_1 \twoheadrightarrow E_2$ means that E_1 reduces to E_2 in zero or more steps.

Definition 3.3

$$
\begin{array}{rlll}
\Re(e, e') \iff & \mathbf{FV}(e, e') = \emptyset & & \\
& \textit{and}\ \forall c \Uparrow (e'\ c) & \implies & \mathbf{S}\ [\![e]\!] = \bot \\
& \textit{and}\ \exists c \Downarrow (e'\ c) & \implies & \exists e'_0, \forall c, e'\ c \twoheadrightarrow c\ e'_0 \\
& & & \textit{and}\ \mathbf{S}\ [\![c\ e'_0]\!] = \mathbf{S}\ [\![\mathcal{N}\ [\![e]\!]\ c]\!]
\end{array}
$$

In order to state and prove Theorem 3.5 we have to be able to apply a function to a number of arguments. The following definition defines a continuation that supplies the arguments listed as its superscript to the function it is the continuation of, and then behaves like itself. Where the definition comes from should be clear from considering the translation of application (and perhaps it is easiest to see this in the call-by-name cps-translation given in Figure 3).

Definition 3.4

$$
\begin{array}{lcl}
c^{[\,]} & = & c \\
c^{[e'_1, \ldots, e'_i]} & = & \lambda f.f\ c^{[e'_2, \ldots, e'_i]}\ e'_1
\end{array}
$$

To prove the correctness of our translation we have to consider arbitrary i, Q and E. The statement of the theorem comes in two parts:

- the first part formalises our intuition that $\mathcal{T}\ i\ Q\ [\![E]\!]$ may fail to terminate only if the semantics of E applied to i arbitrary values gives a value in Q; and
- the second part says that if it terminates, then the semantics of the translated expression is the equal to that which is given by the $\mathcal{N}$ translation function.

The theorem is complicated in order to give enough power for the inductive step. Both parts of the theorem need to be proved together because they rely on each other.

Theorem 3.5

1. $\forall c \Uparrow ((\mathcal{T}\ i\ Q\ [\![E]\!]\ c^{[e'_1, \ldots, e'_i]})\ [d'_j/x_j]_{j=1}^{j=k})$ *and* $\Re(e_i, e'_i)$ *and* $\Re(d_i, d'_i)$ *and* $\mathbf{FV}(E) = \{x_1, \ldots, x_k\}$ *and* $\mathbf{FV}(e_i, e'_i, d_i, d'_i) = \emptyset$

$$\implies \mathbf{S}\ [\![E\ e_1\ \ldots e_i[d_j/x_j]_{j=1}^{j=k}]\!] \in Q.$$

2. $\exists c \Downarrow ((\mathcal{T}\ i\ Q\ [\![E]\!]\ c^{[e'_1, \ldots, e'_i]})[d'_j/x_j]_{j=1}^{j=k})$ *and* $\Re(e_i, e'_i)$ *and* $\Re(d_i, d'_i)$ *and* $\mathbf{FV}(E) = \{x_1, \ldots, x_k\}$ *and* $\mathbf{FV}(e_i, e'_i, d_i, d'_i) = \emptyset$

$$
\begin{array}{l}
\implies \exists e'_0, \forall c (\mathcal{T}\ i\ Q\ [\![E]\!]\ c^{[e'_1, \ldots, e'_i]})[d'_j/x_j]_{j=1}^{j=k} \twoheadrightarrow c\ e'_0 \\
\textit{and}\ \mathbf{S}\ [\![c\ e'_0]\!] = \mathbf{S}\ [\![(\mathcal{N}\ [\![E]\!]\ c^{[e'_1, \ldots, e'_i]})[d'_j/x_j]_{j=1}^{j=k}]\!]
\end{array}
$$

Proof

The theorem is proved by structural induction over the terms in the language Λ_T. The cases for all constants except **if** are trivial because they are in normal form, and $\mathcal{T}\ i\ Q\ [\![E]\!]$ is defined to be $\mathcal{N}\ [\![E]\!]$.

In this abstract we will prove the first rule for application.

1. Suppose that $\forall c, \Uparrow ((\mathcal{T}\ i\ Q\ [\![E_1\ E_2]\!]\ c^{[e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k})$. We first of all note that this is equal to

$$(\mathcal{T}\ 0\ P\ [\![E_2]\!]\ (\lambda v.\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ c^{[\lambda c.c\,v, e'_1,\ldots,e'_i]}))[d'_j/x_j]_{j=1}^{j=k}.$$

There are two cases why this term might diverge.

$\forall c \Uparrow ((\mathcal{T}\ 0\ P\ [\![E_2]\!]\ c)[d'_j/x_j]_{j=1}^{j=k})$: By the induction hypothesis (part 1) this means that $\mathbf{S}\ [\![E_2[d_j/x_j]_{j=1}^{j=k}]\!] \in P$, and by the condition of the translation rule this means that $\mathbf{S}\ [\![(E_1\ E_2\ e_1\ \ldots\ e_i)[d_j/x_j]_{j=1}^{j=k}]\!] \in Q$.

$\exists c \Downarrow ((\mathcal{T}\ 0\ P\ [\![E_2]\!]\ c)[d'_j/x_j]_{j=1}^{j=k})$: By part 2 of the induction hypothesis this means that $\exists e'_0$ such that $\forall c\ (\mathcal{T}\ 0\ P\ [\![E_2]\!]\ c)[d'_j/x_j]_{j=1}^{j=k} \twoheadrightarrow c\ e'_0$ and so the term whose divergence we are considering reduces to

$$(\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ c^{[\lambda c.c\,e'_0, e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k}.$$

If we can prove that $\Re(E_2[d_j/x_j]_{j=1}^{j=k}, \lambda c.c\ e'_0)$, then we can use part 1 of the induction hypothesis to conclude that

$$\mathbf{S}\ [\![(E_1\ E_2\ e_1\ \ldots\ e_i)[d_j/x_j]_{j=1}^{j=k}]\!] \in Q,$$

which is what we are required to show. To show this, we note that part 2 of the induction hypothesis implies that

$$\begin{array}{lll}
\mathbf{S}\ [\![c\ e'_0]\!] & & \\
\quad = & \mathbf{S}\ [\![(\mathcal{N}\ [\![E_2]\!]\ c)[d'_j/x_j]_{j=1}^{j=k}]\!] & \\
\quad = & \mathbf{S}\ [\![\mathcal{N}\ [\![E_2]\!]\ c]\!][\mathbf{S}\ [\![d'_j]\!]\ \rho^{\mathbf{S}}/x_j]_{j=1}^{j=k} & \\
\quad = & \mathbf{S}\ [\![\mathcal{N}\ [\![E_2]\!]\ c]\!][\mathbf{S}\ [\![\mathcal{N}\ [\![d_j]\!]]\!]\ \rho^{\mathbf{S}}/x_j]_{j=1}^{j=k} & (*) \\
 & \text{since } \Re(d_i, d'_i) \Rightarrow \mathbf{S}\ [\![\mathcal{N}\ [\![d_i]\!]]\!] = \mathbf{S}\ [\![d'_i]\!] & \\
\quad = & \mathbf{S}\ [\![\mathcal{N}\ [\![E_2]\!][\mathcal{N}\ [\![d_j]\!]/x_j]_{j=1}^{j=k}\ c]\!] & \\
\quad = & \mathbf{S}\ [\![\mathcal{N}\ [\![E_2[d_j/x_j]_{j=1}^{j=k}]\!]\ c]\!] & \\
 & \text{since } \mathcal{N}\ [\![x]\!] = x &
\end{array}$$

and this implies that $\Re(E_2[d_j/x_j]_{j=1}^{j=k}, \lambda c.c\ e'_0)$ as required. (The step marked $(*)$ can be proved by checking firstly the case that $\mathbf{S}\ [\![d'_i]\!] = \lambda c.c\ \bot$ and secondly the case that $\mathbf{S}\ [\![d'_i]\!] \neq \lambda c.c\ \bot$, corresponding to the two implications in the definition of $\Re$. The first case follows from the correctness of $\mathcal{N}$ and the second case is straightforward.)

2. Suppose instead that $\exists c \Downarrow ((\mathcal{T}\ i\ Q\ [\![E_1\ E_2]\!]\ c^{[e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k})$. Then, by part 2 of the induction hypothesis, $\exists e'_0$ such that $\forall c,\ \mathcal{T}\ 0\ P\ [\![E_2]\!]\ c \twoheadrightarrow c\ e'_0$. For the particular continuation that we get from the translation of the application $(E_1\ E_2)$ this means that

$$\begin{array}{ll}
(\mathcal{T}\ i\ Q\ [\![E_1\ E_2]\!]\ c^{[e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k} & \\
\quad \twoheadrightarrow\ (\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ c^{[\lambda c.c\,e'_0, e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k} &
\end{array}$$

Since the reduction of this term terminates, we can invoke part 2 of the induction hypothesis to conclude that $\exists e'$ such that

$$\begin{array}{l}
(\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ c^{[\lambda c.c\,e'_0, e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k} \\
\quad \twoheadrightarrow\ c\ e'
\end{array}$$

and that $\mathbf{S}\ [\![c\ e']\!] = \mathbf{S}\ [\![\mathcal{N}\ [\![E_1]\!]\ c^{[\lambda c.c\, e'_0, e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k}]\!]$. Recalling that reduction preserves semantics, this means that

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{S}\ [\![(\mathcal{T}\ i\ Q\ [\![E_1\ E_2]\!]\ c^{[e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k}]\!]} \\
= & \mathbf{S}\ [\![c\ e']\!] \\
= & \mathbf{S}\ [\![\mathcal{N}\ [\![E_1]\!]\ c^{[\lambda c.c\, e'_0, e'_1,\ldots,e'_i]}[d'_j/x_j]_{j=1}^{j=k}]\!] \\
= & \mathbf{S}\ [\![\mathcal{N}\ [\![E_1]\!]\ (\lambda f.f\ c^{[e'_1,\ldots,e'_i]}\ (\lambda c.c\ e'_0))[d'_j/x_j]_{j=1}^{j=k}]\!] \\
= & \mathbf{S}\ [\![(\mathcal{N}\ [\![E_1\ E_2]\!]\ c^{[e'_1,\ldots,e'_i]})[d'_j/x_j]_{j=1}^{j=k}]\!] \\
 & \text{since } \mathbf{S}\ [\![\lambda c.c\ e'_0]\!] = \mathbf{S}\ [\![\mathcal{N}\ [\![E_2]\!][d'_j/x_j]_{j=1}^{j=k}]\!] \\
 & \text{by part 2 of the induction hypothesis}
\end{array}$$

which is what we were required to show.

■

4 Towards a Real Compiler: the Need for Approximations

So far we have not paid any attention to effectiveness because we wanted to separate the issues of exploiting strictness information in a systematic and provably correct way from the inevitable engineering choices that must be taken in the design of a compiler. Fortunately however our abstract rules can be refined to be used in a real compiler.

There are two reasons why the transformation rules described in the previous sections need to be refined:

1. the conditions for the strictness optimisation that occur in the rule for application are not computable in general; and

2. there is no provision for limiting the size of the code produced for recursive values, which might be infinite.

The solution for solving the first point is to rely on a program analysis to determine this information. Many have been proposed in the literature: [Myc81, BHA86, WH87, Wad87, Hun91, Jen92, LM91] for example. It is important to note that the theory we have presented is not committed to one particular kind of analysis.

The second point can be tackled in a number of ways, some of which give better information than others. Let us mention some of these in the light of the transformation rules in Figure 6. The original rule for $\mathbf{fix}_\sigma$ was:

$$\mathcal{T}\ i\ Q\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] = \mathcal{T}\ i\ Q\ [\![E[\mathbf{fix}_\sigma\ (\lambda x.E)/x]]\!]$$

The most precise and finite refinement of this rule is the following:

$$\mathcal{T}\ i\ Q\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] = \mathbf{fix}_\sigma\ (\lambda x.\mathcal{T}\ i\ Q\ [\![E]\!])$$

and this refinement is correct provided that

$$\mathcal{T}\ i\ Q\ [\![E[E'/x]]\!] = (\mathcal{T}\ i\ Q\ [\![E]\!])[\mathcal{T}\ i\ Q\ [\![E']\!]/x].$$

Intuitively this means that the occurrences of x within E are compiled with the same i and Q as the whole expression.

At the other end of the spectrum, the least precise, but always correct refinement, consists in forgetting all the strictness information and compiling the recursive call with $Q = \{\perp_{\mathbf{S}_\sigma}\}$:

$$\mathcal{T}\ i\ Q\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] = \mathbf{fix}_\sigma\ (\lambda x.\mathcal{T}\ i\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E]\!]).$$

This refinement is obviously correct because all the Scott-closed sets considered in this paper are non-empty (and so include bottom).

Both these extreme solutions lead to the production of a single piece of code for a recursive expression. If the condition for the first refinement is not satisfied, which means that the recursive call should be compiled with different strictness attributes, then the compiler could produce several versions of the value. The number of versions is an engineering decision motivated by space-efficiency trade-offs. The important point however is that the number of these versions can be kept finite by relying on one of the two solutions mentioned above at some stage of the refinement.

5 Related Work

As mentioned in the introduction a number of papers have been devoted to step (1): proving the correctness of the original compiler [Sch80, Wan82, NN88, Dyb85, Mor73, Mos80, TWW81, Les87, Les88, CCM87, FM91]; and step (2): proving the correctness of the result of the analysis [CC79, CC92, Bur91b, Nie89, WH87, LM91, Jen92, Ben92].

Some of the work devoted to the proof of step (1) include a number of local optimisations (such as peephole optimisations), but very few consider optimisations relying on a global analysis. The latter are more difficult to validate because they involve context-dependent transformations. The only papers addressing this issue, to our knowledge, are [Nie85] and [Ger75]. The second paper is concerned with partial correctness and relies on progam annotations and theorem-proving methods. The first paper considers a simple imperative language and a collecting semantics associating with each program point the set of states which are possible when control reaches that point. This method is not directly applicable to strictness analysis because only a weak equivalence is obtained in the case of a backwards analysis (whereas termination is the crucial issue in the correctness proof of strictness-based optimisations). Also their methods deal with local transformations where strictness-based optimisations involve global modifications of the program.

The work that is closest in spirit to this paper is [Les88], which states a correctness property of an optimisation based on strictness analysis in the context of combinator graph reduction on a version of the G-machine. The result however is limited to simple strictness (corresponding to Section 2.1 of this paper), and it is expressed in terms of low-level machine steps.

Burn showed that the operational model underlying the transformation given in Figure 6 is correct in [Bur91b]. He also showed how this information could be used in compiling code for an abstract machine. However, the correctness of this step was not considered.

Last but not least, the work presented here is one more demonstration of the significance of continuations. The benefits of continuations to compiler design have been illustrated in [Ste78, KKR+86, Kra88, App92]. The correctness of the continuation-passing translation has been studied in [Rey74, Plo75].

6 Conclusion

A great number of techniques and optimisation methods have been proposed in the last decade for the implementation of functional languages. These techniques are more and more sophisticated, leading to more and more efficient implementations of functional languages. However, it is difficult to give a formal account of the various proposed optimisations and to state precisely how these many techniques relate to each other. This paper can be seen as a first step towards a unified framework for the description of various implementation choices. In the future we propose to make several extensions to this work, including: taking more context into account in compiling a function application; making use of another sort of evaluation information; and describing formally the exploitation of a sticky version of the strictness analysis. We briefly state what these are in the following two paragraphs.

The rule for compiling applications loses the fact that E_1 is applied to E_2 when compiling the body of E_1. This can be seen most clearly where the test for changing the evaluation order is given, where the function is applied to i arbitrary arguments, rather than any arguments it was already applied to (c.f. the concept of 'context-sensitive' evaluation transformers in [Bur91b, Section 5.3]). We envisage that this should be fairly easy to carry over into the proof of Theorem 3.5, where $c^{[e'_1,\ldots,e'_i]}$ is replaced by one which records the arguments in the application.

Projection-based analyses can also give information of the form: "this argument cannot be evaluated yet, but if it is ever evaluated, then do so much evaluation of it" [Bur90]. Again we should be able to modify the rule for application to accommodate this. Instead of using $\mathcal{T}\ 0\ \{\perp_{\mathsf{S}_\sigma}\}\ [\![E_2]\!]$ in the case that the argument expression cannot be evaluated, this can be changed to $\mathcal{T}\ 0\ P\ [\![E_2]\!]$ where P is the Scott-closed set that represents how much evaluation can be done to the expression if it is evaluated.

The optimised compilers described in this paper follow [NN90] in avoiding the introduction of an intermediate pass to annotate the program with strictness information. A two-phases compiler sometimes produces more efficient code because strictness information can be exploited more effectively; we are currently studying its formalisation within our framework.

It is also hoped that the methodology of this paper will suggest ways of using information about how expressions have been evaluated for structured data types such as lists, something which has so far eluded even the implementation community.

References

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Ben92] P.N. Benton. Strictness logic and polymorphic invariance. In *Proceedings of the Symposium on Logical Foundations of Computer Science*, 20–24 July 1992.

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis of higher-order functions. *Science of Computer Programming*, 7:249–278, November 1986.

[Bur90] G.L. Burn. Using projection analysis in compiling lazy functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 227–241, Nice, France, 27–29 June 1990.

[Bur91a] G.L. Burn. The evaluation transformer model of reduction and its correctness. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of TAPSOFT'91, Volume 2*, pages 458–482, Brighton, UK, 8–12 April 1991. Springer-Verlag LNCS 494.

[Bur91b] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman in association with MIT Press, 1991. 238pp.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*, pages 269–282. ACM, January 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4), 1992. Special Issue on Abstract Interpretation.

[CCM87] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[Dyb85] P. Dybjer. Using domain algebras to prove the correctness of a compiler. In *Proceedings of STACS85*, pages 98–108. Springer-Verlag LNCS182, 1985.

[Fis72] M. J. Fischer. Lambda calculus schemata. In *ACM Conference on Proving Assertions about Programs*, pages 104–109, New Mexico, January 1972. ACM Sigplan Notices 7(1).

[FM91] P. Fradet and D Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[Ger75] S.L. Gerhart. Correctness-preserving program transformations. In *Proceedings of POPL75*, pages 54–66. ACM, 1975.

[Hun91] L.S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1991.

[Jen92] T.P. Jensen. Disjunctive strictness analysis. In *Proceedings of the 7th Symposium on Logic In Computer Science.* Computer Society Press of the IEEE, 1992.

[JL91] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 636–666, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523.

[KKR+86] D.A. Kranz, R. Kelsey, J.A. Rees, P. Hudak, J. Philbin, and N.I. Adams. Orbit: An optimising compiler for scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[Kra88] D.A. Kranz. *Orbit: An Optimising Compiler for Scheme.* PhD thesis, Department of Computer Science, Yale University, February 1988. Report Number YALEU/DCS/RR-632.

[Les87] D. Lester. The G-machine as a representation of stack semantics. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 46–59. Springer-Verlag LNCS 274, September 1987.

[Les88] D.R. Lester. *Combinator Graph Reduction: A Congruence and its Applications.* DPhil thesis, Oxford University, 1988. *Also* published as Technical Monograph PRG-73.

[LM91] A. Leung and P. Mishra. Reasoning about simple and exhaustive demand in higher-order languages. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 329–351, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523.

[Mor73] F.L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings of POPL73*, pages 144–152. ACM, 1973.

[Mos80] P.D. Mosses. A constructive approach to compiler correctness. In *Proceedings of ICALP80*, pages 449–462. Springer-Verlag LNCS85, 1980.

[MW85] A. Meyer and M. Wand. Continuation semantics in the typed lambda-calculus. In *LNCS 193: Proceedings of Logics of Programs*, pages 219–224, Berlin, 1985. Springer-Verlag.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Department of Computer Science, December 1981. *Also* published as CST-15-81.

[Nie85] F. Nielson. Program transformations in a denotational setting. *ACM TOPLAS*, 7:359–379, 1985.

[Nie89] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[NN88] H Riis Nielson and F. Nielson. Two-level semantics and code generation. *TCS*, 56:59–133, 1988.

[NN90] H. Nielson and F. Nielson. Context information for lazy code generation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 251–263, Nice, France, 27–29 June 1990.

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

[Rey74] J.C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the Second Colloquium on Automata, Languages and Programming*, pages 141–156, Saarbrucken, 1974. Springer-Verlag.

[Sch80] D.A. Schmidt. State transition machines for lambda-calculus expressions. In *Proceedings of the Semantics-Directed Compiler Generation Workshop*, pages 415–440. Springer-Verlag LNCS94, 1980.

[Ste78] G.L. Steele Jr. Rabbit: A compiler for scheme. Technical Report AI Tech. Rep. 474, MIT, Cambridge, Mass., 1978.

[TWW81] J.W. Thatcher, E.G Wagner, and J.B. Wright. More advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.

[Wad87] P.L. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C.L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.

[Wan82] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 385–407. Springer-Verlag LNCS 274, September 1987.