Proving the Correctness of Compiler Optimisations Based on Strictness Analysis*

Geoffrey Burn[1] and Daniel Le Métayer[2]

[1] Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, United Kingdom
[2] Irisa/Inria, Campus de Beaulieu, 35042 Rennes Cedex, France

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.

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 can be done in 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 have been surprisingly few attempts at tackling step (3). In this paper we show how 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:

- changing the evaluation order to evaluate 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').

* Correspondence regarding this paper should be addressed to the second author. The first author was 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). The second author was on leave from INRIA/IRISA and was 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 to generate better code, and are proven to preserve the semantics of programs. Any of these can then replace the cps-translation phase in the compiler described in [FM91], so that we can demonstrate an optimising compiler which has been proved correct.

We start by showing how simple strictness information can be used to change 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 type of a translated function makes explicit whether or not an (evaluated) argument is being passed in a closure in the heap (i.e. whether or not it is 'boxed'); important information for a compiler-writer. This appears to be a natural alternative to that given in [JL91] for expressing the boxed/unboxed distinction.

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.

A survey of related work can be found in Section 4, and Section 5 reviews the benefits of this approach and identifies areas of further research.

We would like to stress that translating programs into continuation-passing style as an early stage in a compiler is of more than theoretical interest. Steele was the first to show that it was beneficial to do this for Scheme programs in his seminal work on the Rabbit compiler [Ste78]. Some of the most efficient implementations of Scheme [KKR+86, Kra88] and ML [App92] use cps-translation. This experience suggests that it might be worthwhile using cps-translation in compilers for lazy functional languages. The alternative cps-translations we have given in this paper could be used in such a compiler to produce better code. For example, they can be used in the context of [FM91] to produce a correct optimising compiler.

Space precludes us from presenting any proofs. These can be found in the full version of the paper, available as a technical report from the institution of each author.

2 Using Simple Strictness Information

Figures 1 and 2 describe the syntax of our functional language and its semantics. Note that we use lifted function domains. This is consistent with most implementations of lazy functional languages which evaluate expressions as far

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

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

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

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

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

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

Semantics of the Types

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

Semantics of the Language Terms

Figure 2. The Semantics of Λ_T

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

Translation of Types

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

Translation of Terms

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

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

as *Weak Head Normal Form (WHNF)* (no evaluation inside lambda abstractions) [Gun92, Chapter 4]. The functions *lift* and *drop* are used to map values from a domain to its lifted counterpart (and vice versa).

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.

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.

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 $\mathsf{B}\ [\![\sigma]\!] = \mathsf{C}\ [\![\sigma]\!] \to \mathbf{Ans}$. This can be stated formally by Theorem 2.

Definition 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.

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

■

Expressions of type $\mathsf{C}\ [\![\sigma]\!] = \mathsf{U}\ [\![\sigma]\!] \to \mathbf{Ans}$ 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].

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

$$\begin{aligned} F &= \lambda x.x + 1 \\ E &= F\ (\mathbf{plus}\ 2\ 7). \end{aligned}$$

Applying the translation rules from Figure 3 yields, after reducing "administrative redexes" introduced by the translation process (the interested reader can find in [DF91] and [SF92] techniques for the systematic elimination of administrative redexes):

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

The application of $\mathcal{N}\ [\![E]\!]$ to some continuation k gives rise to the following reductions:

$$\begin{aligned} &(\lambda c.F_1\ c\ (\lambda c.(\mathbf{plus_c}\ c\ 2\ 7)))\ k \\ &\twoheadrightarrow F_1\ k\ (\lambda c.(\mathbf{plus_c}\ c\ 2\ 7)) \\ &\twoheadrightarrow (\lambda c.(\mathbf{plus_c}\ c\ 2\ 7))\ (\lambda m.\mathbf{plus_c}\ k\ m\ 1) \\ &\twoheadrightarrow \mathbf{plus_c}\ (\lambda m.\mathbf{plus_c}\ k\ m\ 1)\ 2\ 7 \\ &\twoheadrightarrow (\lambda m.\mathbf{plus_c}\ k\ m\ 1)\ 9 \\ &\twoheadrightarrow \mathbf{plus_c}\ k\ 9\ 1 \\ &\twoheadrightarrow k\ 10 \end{aligned}$$

We note that F_1 is passed the unevaluated argument $(\lambda c.\mathbf{plus_c}\ c\ 2\ 7)$ which is immediately evaluated in the body of F_1 (fourth reduction). This cost of passing an unevaluated argument may be significant in terms of computation time and in terms of memory consumption (it may even change the order of magnitude of memory complexity of the program). 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 is the case if the divergence of the argument implies the divergence of the function application, or in other words, the function is strict. Strictness analysis can detect a subset of the cases when this condition is satisfied.

Evaluating the argument before the function call can be expressed in the following way for our example. $\mathcal{S}\ [\![E]\!] = \lambda c.\mathbf{plus_c}\ (\lambda v.F_1\ c\ (\lambda c.c\ v))\ 2\ 7$ and the application of this to k can be reduced as follows:

$$
\begin{aligned}
&(\lambda c.\mathbf{plus_c}\ (\lambda v.F_1\ c\ (\lambda c.c\ v))\ 2\ 7)\ k\\
&\twoheadrightarrow \mathbf{plus_c}\ (\lambda v.F_1\ k\ (\lambda c.c\ v))\ 2\ 7\\
&\twoheadrightarrow (\lambda v.F_1\ k\ (\lambda c.c\ v))\ 9\\
&\twoheadrightarrow F_1\ k\ (\lambda c.c\ 9)\\
&\twoheadrightarrow (\lambda c.c\ 9)\ (\lambda m.\mathbf{plus_c}\ k\ m\ 1)\\
&\twoheadrightarrow (\lambda m.\mathbf{plus_c}\ k\ m\ 1)\ 9\\
&\twoheadrightarrow \mathbf{plus_c}\ k\ 9\ 1\\
&\twoheadrightarrow k\ 10
\end{aligned}
$$

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 now compile F and E in the following way:

$$
\begin{aligned}
\mathcal{S}'\ [\![F]\!] &= \lambda c.c\ F_2\\
F_2 \qquad &= \lambda c.\lambda x.\mathbf{plus_c}\ c\ x\ 1\\
\mathcal{S}'\ [\![E]\!] &= \lambda c.\mathbf{plus_c}\ (F_2\ c)\ 2\ 7.
\end{aligned}
$$

The reduction of this expression avoids the unnecessary creation of a closure and its evaluation in the body of F_2:

$$
\begin{aligned}
&(\lambda c.\mathbf{plus_c}\ (F_2\ c)\ 2\ 7)\ k\\
&\twoheadrightarrow \mathbf{plus_c}\ (F_2\ k)\ 2\ 7\\
&\twoheadrightarrow F_2\ k\ 9\\
&\twoheadrightarrow \mathbf{plus_c}\ k\ 9\ 1\\
&\twoheadrightarrow k\ 10
\end{aligned}
$$

In fact, $\mathcal{S}'\ [\![E]\!]$ is what is produced by the compilation rules for call-by-value [FM91].

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

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

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

Translation of Terms

Figure 4. The Cps-conversion Using Simple Strictness Information

$(= \mathsf{U}\,[\![int \to int]\!])$, 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 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. Although the test given in the rule is not effective, many analyses have been developed which can find a subset of the cases when it holds (see [Ben92, BHA86, Jen92a, KM89, Myc81, Nie88] for example).

- 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.

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 3. *For all closed terms of ground type* E: $\mathbf{S}\ [\![\mathcal{S}\ [\![E]\!]]\!]\ \emptyset = \mathbf{S}\ [\![\mathcal{N}\ [\![E]\!]]\!]\ \emptyset$. ■

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; it is expecting a closure as an argument. This means that 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.

The compilation rules in Figure 5 allow values to be passed unboxed. We can see from the first rule for application that the transformation $\mathcal{S}'$ encodes the same evaluation order as $\mathcal{S}$; all that has changed is that some values are passed unboxed. How this is done will now be explained.

The compilation rule $\mathcal{S}'$ takes two extra arguments. In compiling the body of some lambda-term, we need to know which free variables will be unboxed, and the set V contains the names of these variables. Whether or not an expression is to be passed unboxed is decided when translating an application, but at that point we do not know which formal parameter the argument expression will be bound to (consider translating $(\ldots(\lambda x_1.\ldots\lambda x_m.D)\ E_1)\ \ldots\ E_n$). The set I records the numbers of the arguments which are passed unboxed. We can understand the use of I and V as follows. In an application $(E_1\ E_2)$, if E_2 is passed unboxed, we record the fact by putting a 1 into the set I. Whichever rule for application is chosen, the nth argument to $(E_1\ E_2)$ is the $(n+1)st$ argument to E_1. This means that all the indices currently in I have to be incremented. If in translating $(\lambda x.E)$ we find that 1 is in I, then the value bound to x is being passed unboxed, and so x is added to V so that the appropriate rule can be chosen when translating variables. If it is not being passed unboxed, then x has to be removed from V because of the scoping rules for a lambda-term. Since the $(n+1)st$ parameter to $\lambda x.E$ is the nth parameter to E, all the values in I have to be decremented.

Notice that each function may be compiled in several different ways, depending on the calling context. Figure 5 gives the four different ways that an application of **plus** can be compiled. There is clearly an engineering decision to be made about how many versions of code should be produced for a function.

Some functions are compiled when their application context is not known (for example, functions which are passed as arguments to another function, or

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

Translation of Types

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

$$
\begin{array}{l}
\mathcal{S}'\ I\ V\ [\![x]\!] = conv_I\ (\lambda c.c\ x) \text{ if } x \in V \\
\qquad\qquad = conv_I\ x \qquad \text{if } x \notin V \\
\mathcal{S}'\ I\ V\ [\![\mathbf{0}]\!] = \mathcal{N}\ [\![\mathbf{0}]\!] \ \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)) \\
\qquad\qquad \text{if } 1 \in I \wedge 2 \in I \\
\qquad\qquad = \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))) \\
\qquad\qquad \text{if } 1 \notin I \wedge 2 \in I \\
\qquad\qquad = \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))) \\
\qquad\qquad \text{if } 1 \in I \wedge 2 \notin I \\
\qquad\qquad = \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)))) \\
\qquad\qquad \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_{(dec\ I)}\ (\mathbf{head}\ x))\ c_1) \qquad \text{if } 1 \in I \\
\qquad\qquad = \lambda c.c\ (\lambda c_1.\lambda x.x(\lambda v.(conv_{(dec\ I)}\ (\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)) \\
\qquad\qquad \text{if } \forall \rho^{\mathbf{S}} : drop\ (\mathbf{S}\ [\![E_1]\!]\ \rho)\ \perp = \perp \\
\qquad\qquad = \lambda c.\mathcal{S}'\ (inc\ I)\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathcal{S}'\ \emptyset\ V\ [\![E_2]\!])) \\
\qquad\qquad \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 \\
\qquad\qquad = \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)]\!] \\
= sel\ (\mathbf{fix}\ (\lambda(x_1, \ldots, x_n).(\mathcal{S}'\ I_1\ V_1\ [\![E']\!], \ldots, \mathcal{S}'\ I_n\ V_n\ [\![E']\!])))
\end{array}
$$

Translation of Terms

Figure 5. The Cps-conversion Using Strictness and Evaluation Information

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 is a bit complicated because the variable being fixpointed may appear in several different contexts in E. For any program, there are only a finite number of these contexts, because both I and V contain at most as many elements as the largest arity of a function in a program. For $1 \leq i \leq n$, the variable x_i stands for x in the context I_i and V_i, and E' is obtained from E by replacing each occurrence of x by the appropriate x_i. The function sel selects the term from the n-tuple returned by the fixed point which corresponds to the context I and V.

The well-typing of translated terms is established by the following theorem.

Definition 4. If ρ is a type environment, then its transformation $\mathcal{S}'_V\ [\![\rho]\!]$ is defined by the rule:

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

■

Theorem 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 $\mathsf{B}'_\emptyset\ [\![\sigma]\!] = \mathsf{B}\ [\![\sigma]\!]$.

The correctness of $\mathcal{S}'$ is formulated in the following theorem.

Theorem 6. *For all terms E,*

$$\mathbf{S}\ [\![\mathcal{S}'\ I\ V\ [\![E]\!]]\!]\ \rho = \mathbf{S}\ [\![conv_I\ (\mathcal{S}\ [\![E]\!])]\!]\ \emptyset\ (Box_V\ \rho^{\mathbf{S}}),$$

where Box_V boxes basic values which are unboxed. ■

3 Using More Sophisticated Information

The previous section exposes the optimisations made possible when evaluation order could be changed, but took no account of how much evaluation could be done to an expression; expressions were only evaluated to WHNF. However, it is clear that some functions require more evaluation of their arguments. For example, the function *reverse* defined by:

$$\begin{aligned} reverse\ x = {} & rev\ x\ \mathbf{nil} \\ & \text{where} \\ & rev = \mathbf{fix}\ (\lambda f.\lambda x.\lambda y.\mathbf{if}\ (\mathbf{eq}\ x\ \mathbf{nil})\ y\ (f\ (\mathbf{tl}\ x)\ (\mathbf{cons}\ (\mathbf{hd}\ x)\ y))) \end{aligned}$$

must traverse the whole of its argument list before it can return a result in WHNF. Moreover, the amount of evaluation required of an argument in an application depends on the amount of evaluation required of the application. For example, if we tried to sum all the elements in the list $(reverse\ E)$, then not only does the structure of E have to be traversed, but all the elements of E have to be evaluated as well.

We call the amount of evaluation required of an expression the *evaluation context* of the expression. A number of useful evaluation contexts such as *head-strictness* or *tail-strictness* have been defined in the literature and several analyses have been proposed to derive context information automatically (see [Bur91b, Jen92b, LM91, NN92, WH87, Wad87] for example). However the various ways of exploiting this context information within a compiler have never been described formally and little work has been done on assessing their effectiveness. We show in this section how these optimisations can be described formally in our framework. We stress the fact that our goal is to expose and prove the possible optimisations but we do not take any position on which of these optimisations should really be integrated within a compiler. This last issue depends on a number of lower level implementation decisions and is better addressed by an experimental study.

An evaluation context can be specified by the set of terms whose evaluation would fail to terminate in that context. For example, evaluating an expression to WHNF will fail to terminate if and only if the expression has no WHNF; and the context representing the evaluation of the structure of a list expression will contain infinite lists and lists having a bottom tail at some point because a program evaluating the structure of such lists would fail to terminate. In general 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 [Bur91a].

Definition 7. *(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. ■

The key idea behind following transformation rules defined in Figure 6 is the following fact, sometimes known as the Evaluation Transformer Theorem [Bur91a, Theorem 7.5].

$$
\begin{array}{l}
\mathcal{T}\ i\ Q\ [\![x]\!] = x \\
\mathcal{T}\ i\ Q\ [\![\mathbf{k}_\sigma]\!] = \mathcal{N}\ [\![\mathbf{k}_\sigma]\!] \\
\mathcal{T}\ i\ Q\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] \\
\quad = \lambda c.\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]\ (\mathbf{if_c}\ (\mathcal{T}\ i\ Q\ [\![E_2]\!]\ c)\ (\mathcal{T}\ i\ Q\ [\![E_3]\!]\ c)) \\
\mathcal{T}\ i\ Q\ [\![E_1\ E_2]\!] \\
\quad = \lambda c.\mathcal{T}\ 0\ P\ [\![E_2]\!]\ (\lambda v.\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
\quad \text{if } \forall \rho^{\mathbf{S}},\ \forall v_0 \in P,\ \forall v_1,\ \ldots,\ v_i : drop\,(\ldots(drop\,(\mathbf{S}\ [\![E_1]\!]\ \rho)\,v_0)\ \ldots)\,v_i \in Q \\
\quad = \lambda c.\mathcal{T}\ (i+1)\ Q\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E_2]\!])) \\
\quad \text{otherwise} \\
\mathcal{T}\ 0\ Q\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_\tau}\}\ [\![E]\!]\ c) \\
\mathcal{T}\ (i+1)\ Q\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathcal{T}\ i\ Q\ [\![E]\!]\ c) \\
\mathcal{T}\ i\ Q\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] \\
\quad = sel\ (\mathbf{fix}\ (\lambda(x_1,\ldots,x_n).(\mathcal{T}\ i_1\ Q_1\ [\![E']\!],\ldots,\mathcal{T}\ i_n\ Q_n\ [\![E']\!])))
\end{array}
$$

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

Fact 8. *Let S and T be Scott-closed sets. If it is safe to evaluate the application $(E_1\ E_2)$ in the context T, and we know that for all $s \in S$, $\mathbf{S}\ [\![E_1]\!]\ \rho\ s \in T$, then it is safe to evaluate E_2 in the context S.* ■

The key intuition about safety is that the evaluation of an expression fails to terminate when being evaluated in the context Q if and only if the semantics of the expression is in Q. It is important to note that the Evaluation Transformer Theorem does not establish a unique context S for evaluating the argument expression; it says that any context satisfying the condition is acceptable.

If $E : \sigma$ is the program to be compiled, its translation is $\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E]\!]$. The second argument to the translation function $\mathcal{T}$ is the evaluation context for the expression, so this rule says that the evaluation of the program is to fail to terminate if and only if its denotational semantics is bottom, which is what we would expect.

The first argument to the translation rule $\mathcal{T}$ is needed for the same reason that caused us to introduce the sets I and V in the translation in Figure 5: i counts the number of argument expressions passed over in order to reach E, and so the evaluation context Q concerns E applied to i arguments, not E itself. When the translation of some term is started, i has the value 0. Again the rules for application and lambda-abstraction complement each other: the first increments the value of i, and the second decrements it.

The translation rules make no restrictions on the evaluation contexts which can be used. However, an implementation will have to choose a finite, and probably small, number of evaluation contexts for compiling each function in a program, because each extra evaluation context for a function means another version of the code has to be produced for that function. Furthermore, these contexts should correspond to useful evaluation modes for the various data types used in

the program.

We can now give intuitions about some of the translation rules:

- The type of $\mathcal{T}\ i\ Q\ [\![E]\!]$ is the same as the type of $\mathcal{N}\ [\![E]\!]$.
- The translation rule for the conditional forces the evaluation of E_1 to WHNF, and then passes the evaluation context to whichever of E_2 and E_3 is chosen for evaluation. Similar rules can be defined for any selection function (e.g. **case**) which first evaluates a discriminating expression and then chooses to evaluate a particular expression based on the value of the evaluated expression.
- There are two rules for translating an application. The first one is used when the argument expression can be evaluated, and the condition for applying it is derived from the Evaluation Transformer Theorem (Fact 8). Note that
  - the evaluation context P is any context, selected from the set of contexts chosen for the implementation, which satisfies the Evaluation Transformer Theorem;
  - $(\mathbf{S}\ [\![E_1]\!]\ \rho)$ is applied to $(i+1)$ arguments in the test to get a value in Q since $(E_1\ E_2)$ had to be applied to i arguments, and so E_1 has to be applied to $(i+1)$;
  - the translation of E_2 is given 0 for its first argument; and
  - the test in the rule is clearly not effective. However, many program analyses have been presented in the literature which can determine safe approximations to the information (see [Myc81, BHA86, WH87, Wad87, Hun91, Jen92a, LM91] for example).

  The second translation rule for application is used when there is to be no change of evaluation order. Note that the expression E_2 is compiled with evaluation context $\{\bot_{\mathbf{S}_\sigma}\}$ because we do not know if any evaluation will be done to the expression, but if it is, then the expression has to be evaluated to at least WHNF, and we cannot guarantee that any more evaluation will be allowed.
- There are two rules for translating a lambda-abstraction. The first is used when the lambda-abstraction is either the top-level term or some argument expression. In this case Q must be $\{\bot_{\mathbf{S}_{\sigma\to\tau}}\}$, because functions can only be evaluated to WHNF, and the body of the lambda-abstraction is treated in the same way as the argument expression is dealt with in the second rule for application. The second is used when a lambda-abstraction has been found after passing over a number of argument expressions. Note that the evaluation context is passed into the translation of the body of the lambda-abstraction.
- The rule for fixed points is again quite complicated, for similar reasons to those discussed in Section 2.2. An occurrence of the fixpoint variable may be applied to varying numbers of arguments in E, and an application of the fixpoint variable may appear in a number of different evaluation contexts. As an example of the second problem, consider the translation:

$$\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_{int}}\}\ [\![\mathbf{fix}\ (\lambda f.\lambda x.\mathbf{if}\ E_1\ (\mathit{ignore}\ (f\ E_2))\ (f\ E_3))]\!],$$

where *ignore* is a function which ignores its argument, so that applications of f are in two different evaluation contexts: one which does no evaluation, and one which evaluates an expression to WHNF. As with the rule in Section 2.2, for $1 \leq j \leq n$, the variable x_j stands for x in in the context where it is applied to i_j arguments and has evaluation context Q_j, and E' is obtained from E by replacing each occurrence of x by the appropriate x_j. The function sel selects the term from the n-tuple returned by the fixed point which corresponds to i and the evaluation context Q.

There is one more important point to note about the fixed point rule: there could be an infinite number of contexts for applications of the fixpoint variable in a particular program. The rule we have given assumes that a finite set of contexts has been chosen for a particular program, as discussed earlier in this section.

Further intuition about how the rules for application and abstraction interact can be obtained by pondering on the following example. Suppose that we are calculating $\mathcal{T}\ 0\ Q\ [\![E]\!]$ where $E = (\lambda x_1 \ldots \lambda x_n.D)\ E_1\ \ldots E_n$. Using the rule for application n times, and then the rule for lambda-abstraction n times, then part of the term from the translation of E will be $\mathcal{T}\ 0\ Q\ [\![D]\!]$, 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.

The following theorem gives the correctness of our translation. It states that translating a program (a closed term of ground type) with $\mathcal{T}$ gives essentially the same result as translating it with $\mathcal{N}$. Proving the theorem requires a complicated induction hypothesis. This will appear in the full version of this paper, but it takes a couple of pages to set up and explain, and then several pages to prove it, so we have omitted it from this version.

Theorem 9. *For all closed terms of ground type $E : \sigma$,*

$$\mathbf{S}\ [\![\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!]\ \emptyset = \mathbf{S}\ [\![\mathcal{N}\ [\![E]\!]]\!]\ \emptyset.$$

■

4 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, CC92b, CC92a, Bur91b, Nie89, WH87, LM91, Jen92a, 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 program 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 works that are closest in spirit to this paper are [Les88] and [DH92]. The first 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. The second also studies the exploitation of strictness information in the context of a Continuation Passing Style compiler. The compilation is described as a composition of two transformations: the first stage introduces explicit suspension constructs derived from the strictness annotations; and the second phase is the traditional call-by-value CPS transformation. The main departure from our work is the way strictness information is expressed in the programs. They assume a type checker to guarantee the well-foundedness of the annotations. Also the fact that the second phase is a call-by-value CPS transformation entails that evaluated values are systematically passed unboxed. As in [Les88], only simple strictness information is considered.

A less thoroughgoing attempt at this problem is also presented in [Bur91b], which shows that the operational model underlying the transformation given in Figure 6 is correct. It also shows how to use this information in compiling code for an abstract machine, but the correctness of the code was not considered.

5 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 studying the extension of unboxing to non-basic types. We briefly explain each of these in the following three 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]).

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 $\mathcal{T}\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E_2]\!]$ 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.

Finally, we have only considered unboxed values for basic types. There seems to be no consensus in the implementation community about the most suitable notion of 'unboxedness' for structured data types such as lists. We believe that the methodology of this paper can help expose and explore the various possible options.

References

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

[Ben92] P.N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proceedings of the International Symposium on Logical Foundations of Computer Science*, pages 33–44, Tver, Russia, 20–24 July 1992. Springer-Verlag LNCS620.

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis for 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.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC92b] 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.

[DF91] O. Danvy and A. Filinski. Representing control: a study of the cps transformation. Technical Report TR CIS-91-2, Kansas State University, 1991.

[DH92] O. Danvy and J. Hatcliff. CPS transformation after strictness analysis. Technical report, Kansas State University, 1992.

[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.

[Gun92] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[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.

[Jen92a] T. P. Jensen. *Abstract Interpretation in Logical Form*. PhD thesis, Imperial College, University of London, November 1992.

[Jen92b] T.P. Jensen. Disjunctive strictness analysis. In *Proceedings of the 7th Symposium on Logic In Computer Science*, pages 174–185, Santa Cruz, California, 22–25 June 1992. Computer Society Press of the IEEE.

[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.

[KM89] Tsung-Min Kuo and P. Mishra. Strictness analysis: a new perspective based on type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, London, 11–13 September 1989. ACM.

[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 *Proceedings of Logics of Programs*, pages 219–224, Berlin, 1985. Springer-Verlag LNCS 193.

[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.

[Nie88] F. Nielson. Strictness analysis and denotational abstract interpretation. *Inform. and Comput.*, 76:29–92, 1988.

[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.

[NN92] F. Nielson and H. Riis Nielson. The tensor product in Wadler's analysis of lists. In B. Krieg-Brückner, editor, *Proceedings of ESOP'92*, pages 351–370, Rennes, France, February 1992. Springer-Verlag LNCS582. Preliminary version of Chapter 8 of Two-level Functional Languages, CUP, 1992.

[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.

[SF92] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. Technical Report TR 92-180, Rice University, 1992.

[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.

This article was processed using the LaTeX macro package with LLNCS style