J. Functional Programming **6** (1): 75–109, January 1996

Proving the correctness of compiler optimisations based on a global analysis: a study of strictness analysis†

GEOFFREY BURN

Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK

DANIEL LE MÉTAYER

Irisa/Inria, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract

A substantial amount of work has been devoted to the proof of correctness of various program analyses but much less attention has been paid to the correctness of compiler optimisations based on these analyses. In this paper we tackle the problem in the context of strictness analysis for lazy functional languages. 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.

Capsule Review

Burn and Le Métayer assume that the results of strictness analysis are available to a compiler, and that the strictness analysis is correct. Given this information, the authors show how it can be used by a compiler written using the continuation passing style in a functional language. The transformations that use the strictness information are then proven to be correct.

This paper is likely to be of interest both to those working in the area of compiler optimisation and to readers interested in correctness proofs for non-trivial programs written in a functional language.

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:

† 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 and was partially funded by the SERC Visiting Fellowship GR/H 19330.

1. Proving the correctness of the original (unoptimised) compiler.
2. Proving the correctness of the analysis.
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:

(a) changing the evaluation order to evaluate an argument expression instead of passing it as an unevaluated closure; and
(b) 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').

Translating programs into Continuation-Passing Style (CPS) allows us to express both uses of strictness information because:

(a) a cps-translation captures the evaluation order of expressions; and
(b) 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 Fradet and Le Métayer (1991), 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: first, we give a cps-conversion where functions can be compiled knowing that some of their arguments have been evaluated (section 2.2); and second, 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 Peyton-Jones and Launchbury (1991) 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

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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

(3) $$\frac{E_1 : \sigma \rightarrow \tau,\ E_2 : \sigma}{(E_1\ E_2) : \tau}$$ (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

Fig. 1. Definition of the language Λ_T.

the first to show that it was beneficial to do this for Scheme programs in his seminal work on the Rabbit compiler (Steele Jr., 1978). Some of the most efficient implementations of Scheme (Kranz *et al.*, 1986; Kranz, 1988) and ML (Appel, 1992) 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 Fradet and Le Métayer (1991) to produce a correct optimising compiler.

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 as *Weak Head Normal Form* (WHNF) (no evaluation inside lambda abstractions) (Gunter, 1992, Chapt. 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 Fradet and Le Métayer (1991). 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

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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

Semantics of the Types

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

Semantics of the Language Terms

Fig. 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

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

Translation of Terms

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

Fig. 3. The call-by-name cps-conversion.

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 (Plotkin, 1975). Furthermore, this redex is always at the head of the expression (Fradet and Le Métayer, 1991), and the expression can be reduced without dynamic search for the next redex, just like machine code.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

We should mention at this stage another possibility for passing the continuation as an extra argument to a lambda abstraction. We could have chosen to pass it as the second argument of the new function rather than passing it as the first argument. This is expressed as follows:

$$\mathcal{N}' \; [\![\lambda x.E]\!] \; = \; \lambda c.c \; (\lambda x.\lambda c.\mathcal{N}' \; [\![E]\!] \; c).$$

This rule can be simplified by η-conversion into:

$$\mathcal{N}' \; [\![\lambda x.E]\!] \; = \; \lambda c.c \; (\lambda x.\mathcal{N}' \; [\![E]\!]).$$

The rule for application becomes:

$$\mathcal{N}' \; [\![E_1 \; E_2]\!] \; = \; \lambda c.\mathcal{N}' \; [\![E_1]\!] \; (\lambda f.f \; (\mathcal{N}' \; [\![E_2]\!]) \; c).$$

Passing the continuation as the second argument is quite common in the literature, see (Appel, 1992; Danvy and Hatcliff, 1993; Reynolds, 1974; Plotkin, 1975) for instance, but the continuation in first position also occurs in Fisher (1972, 1993), Flanagan *et al.* (1993) and Sabry and Felleisen (1993). The above simplification rule for lambda abstraction suggests that passing the continuation as the second argument sometimes leads to a more compact representation. This potential advantage seems to disappear when *administrative redexes* (redexes introduced by the translation process) are systematically reduced as shown in Danvy and Hatcliff (1994) and Sabry and Felleisen (1993). In any case, the impact of this choice is not significant for the results presented here. Our choice was motivated by the fact that the work described in this paper is part of a broader project for the design of a complete compiler described as a succession of transformations (Fradet and Le Métayer, 1991; Giorgi and Le Métayer, 1990). Having the continuation in first position allows us to produce machine code without leaving the purely functional framework. The head function becomes the next instruction to execute and the (contiguous) continuation is the rest of the code. The interested reader is refered to Fischer (1993) and Danvy and Hatcliff (1994) for further discussions on this choice.

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

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

Expressions of type $\mathsf{C} \; [\![\sigma]\!] = \mathsf{U} \; [\![\sigma]\!] \rightarrow \mathbf{Ans}$ are continuations: they take the result of evaluating an expression of type $\mathsf{U} \; [\![\sigma]\!]$ into an answer. Meyer and Wand (1985)

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

first showed that the type of the cps-translation of an expression could be derived from the type of the original expression.

Each primitive operator **op** has a cps version $\mathbf{op}_c$. For instance $\mathbf{plus}_c$ performs the addition and passes the result to its continuation. We also use primitive **eq** in the example in section 3; it is treated in the same way as **plus**.

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

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

We first show the application of the rules of Figure 3 to F and E:

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

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

We implicitly reduce the administrative redexes introduced by the translation process. The interested reader can find in Danvy and Filinski (1991) and Sabry and Felleisen (1992) techniques for the systematic elimination of administrative redexes.

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

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

We note that F_1 is passed the unevaluated argument $(\lambda c.\mathbf{plus_c}\ c\ 2\ 7)$ which is

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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

Translation of Terms

Fig. 4. The cps-conversion using simple strictness information.

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. $\mathscr{S}\ [\![E]\!] = \lambda c.\mathbf{plus_C}\ (\lambda v.F_1\ c\ (\lambda c.c\ v))\ 2\ 7$ and the application of this to k can be reduced as follows:

$$
\begin{array}{ll}
 & (\lambda c.\mathbf{plus_C}\ (\lambda v.F_1\ c\ (\lambda c.c\ v))\ 2\ 7)\ k \\
\rightarrow\!\!\!\rightarrow & \mathbf{plus_C}\ (\lambda v.F_1\ k\ (\lambda c.c\ v))\ 2\ 7 \\
\rightarrow\!\!\!\rightarrow & (\lambda v.F_1\ k\ (\lambda c.c\ v))\ 9 \\
\rightarrow\!\!\!\rightarrow & F_1\ k\ (\lambda c.c\ 9) \\
\rightarrow\!\!\!\rightarrow & (\lambda c.c\ 9)\ (\lambda m.\mathbf{plus_C}\ k\ m\ 1) \\
\rightarrow\!\!\!\rightarrow & (\lambda m.\mathbf{plus_C}\ k\ m\ 1)\ 9 \\
\rightarrow\!\!\!\rightarrow & \mathbf{plus_C}\ k\ 9\ 1 \\
\rightarrow\!\!\!\rightarrow & k\ 10
\end{array}
$$

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:

$$
\begin{array}{lcl}
\mathscr{S}'\ [\![F]\!] & = & \lambda c.c\ F_2 \\
F_2 & = & \lambda c.\lambda x.\mathbf{plus_C}\ c\ x\ 1 \\
\mathscr{S}'\ [\![E]\!] & = & \lambda c.\mathbf{plus_C}\ (F_2\ c)\ 2\ 7.
\end{array}
$$

The reduction of this expression avoids the unnecessary creation of a closure and

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

its evaluation in the body of F_2:

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

In fact, for this particular expression E, $\mathscr{S}'\ [\![E]\!]$ is what is produced by the compilation rules for call-by-value (Fradet and Le Métayer, 1991).

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 Peyton Jones and Launchbury (1991). 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:

1. $\mathscr{N}\ [\![E]\!]$ and $\mathscr{S}\ [\![E]\!]$ have the same type, and a similar theorem to Theorem 2.2 can easily be proved.
2. 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 Benton, 1992; Burn *et al.*, 1986; Jensen, 1992a; Kuo and Mishra, 1989; Mycroft, 1981; and Nielson, 1988, 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 $\mathscr{S}$ conversion scheme so that strictness information can be used in translating subexpressions.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

We illustrate the new transformation with the computation of $\mathscr{S}\ [\![F]\!]$ and $\mathscr{S}\ [\![E]\!]$, where F and E are the expressions introduced above. Note that the resulting expression corresponds to the result of the first optimisation as presented at the beginning of this section.

$$
\begin{array}{lcl}
\mathscr{S}\ [\![F]\!] & = & \mathscr{S}\ [\![\lambda x.\mathbf{plus}\ x\ 1]\!] \\
 & = & \lambda c.c(\lambda c.\lambda x.\mathscr{S}\ [\![\mathbf{plus}\ x\ 1]\!]\ c) \\
 & = & \lambda c.c\ F_1' \\
F_1' & = & \lambda c.\lambda x.\mathscr{S}\ [\![\mathbf{plus}\ x\ 1]\!]\ c \\
 & = & \lambda c.\lambda x.\mathscr{S}\ [\![1]\!]\ (\lambda v.\mathscr{S}\ [\![\mathbf{plus}\ x]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
 & = & \lambda c.\lambda x.(\lambda c.c\ 1)\ (\lambda v.\mathscr{S}\ [\![\mathbf{plus}\ x]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
 & = & \lambda c.\lambda x.\mathscr{S}\ [\![\mathbf{plus}\ x]\!]\ (\lambda f.f\ c\ (\lambda c.c\ 1)) \\
 & = & \lambda c.\lambda x.(\lambda c.\mathscr{S}\ [\![x]\!]\ (\lambda v.\mathscr{S}\ [\![\mathbf{plus}]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))))(\lambda f.f\ c\ (\lambda c.c\ 1)) \\
 & = & \lambda c.\lambda x.(\lambda c.x\ (\lambda v.\mathscr{S}\ [\![\mathbf{plus}]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))))(\lambda f.f\ c\ (\lambda c.c\ 1)) \\
 & = & \lambda c.\lambda x.x\ (\lambda v.\mathscr{S}\ [\![\mathbf{plus}]\!]\ (\lambda f.f\ (\lambda f.f\ c\ (\lambda c.c\ 1))\ (\lambda c.c\ v))) \\
 & = & \lambda c.\lambda x.x\ (\lambda v.\mathbf{plus_C}\ c\ v\ 1)
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{S}\ [\![E]\!] & = & \mathscr{S}\ [\![F\ (\mathbf{plus}\ 2\ 7)]\!] \\
 & = & \lambda c.\mathscr{S}\ [\![\mathbf{plus}\ 2\ 7]\!]\ (\lambda v.\mathscr{S}\ [\![F]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
 & = & \lambda c.(\lambda c.\mathbf{plus_C}\ c\ 2\ 7)\ (\lambda v.\mathscr{S}\ [\![F]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
 & = & \lambda c.\mathbf{plus_C}\ (\lambda v.\mathscr{S}\ [\![F]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v)))\ 2\ 7 \\
 & = & \lambda c.\mathbf{plus_C}\ (\lambda v.F_1'\ c\ (\lambda c.c\ v))\ 2\ 7
\end{array}
$$

The correctness of this translation is expressed by the following theorem ($\emptyset$ denotes the empty environment). 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 closed terms* E: $\mathsf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ \emptyset = \mathsf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \emptyset$.

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 $\mathscr{S}'$ encodes the same evaluation order as $\mathscr{S}$; all that has changed is that some values are passed unboxed. How this is done will now be explained.

First we need a new collection of types $\mathsf{U}'_I\ [\![\sigma]\!]$, $\mathsf{C}'_I\ [\![\sigma]\!]$ and $\mathsf{B}'_I\ [\![\sigma]\!]$ which are defined like $\mathsf{U}\ [\![\sigma]\!]$, $\mathsf{C}\ [\![\sigma]\!]$ and $\mathsf{B}\ [\![\sigma]\!]$ except for the two rules for $\mathsf{U}'_I\ [\![\sigma \to \tau]\!]$. This is because we need to distinguish, in a function type, boxed arguments and unboxed arguments. The subscript I is a set containing the positions of unboxed arguments in the function type. Let us consider, for example, the translation of a term of type $int \to int \to bool$ where the first argument is to be passed boxed and the second unboxed. This is achieved by calculating $\mathsf{C}'_{\{2\}}\ [\![int \to int \to bool]\!]$ as follows:

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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

Translation of Types

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

$$
\begin{array}{lll}
\mathscr{S}'\ I\ V\ [\![x]\!] & = \mathbf{conv}^{\sigma}_I\ (\lambda c.c\ x) & \text{if } x : \mathsf{U}\ [\![\sigma]\!] \in V \\
 & = \mathbf{conv}^{\sigma}_I\ x & \text{if } x : \mathsf{B}\ [\![\sigma]\!] \notin V \\
 & = x & \text{if } x : \mathsf{B}'_I\ [\![\sigma]\!] \\
\mathscr{S}'\ I\ V\ [\![\mathbf{0}]\!] & = \mathscr{N}\ [\![\mathbf{0}]\!] & \text{(and similarly for other basic values)} \\
\mathscr{S}'\ I\ V\ [\![\mathbf{plus}]\!] & = \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.\mathbf{plus_C}\ c_2\ x\ y)) & \\
 & \text{if } 1 \in I \wedge 2 \in I & \\
 & = \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.\mathbf{plus_C}\ c_2\ m\ y))) & \\
 & \text{if } 1 \notin I \wedge 2 \in I & \\
 & = \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.y\ (\lambda m.\mathbf{plus_C}\ c_2\ x\ m))) & \\
 & \text{if } 1 \in I \wedge 2 \notin I & \\
 & = \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.\mathbf{plus_C}\ c_2\ m\ n)))) & \\
 & \text{if } 1 \notin I \wedge 2 \notin I &
\end{array}
$$

$$\mathscr{S}'\ I\ V\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] = \lambda c.\mathscr{S}'\ \emptyset\ V\ [\![E_1]\!](\mathbf{if_C}\ (\mathscr{S}'\ I\ V\ [\![E_2]\!]\ c)\ (\mathscr{S}'\ I\ V\ [\![E_3]\!]\ c))$$

$$\mathscr{S}'\ I\ V\ [\![\mathbf{cons}]\!] = \mathscr{N}\ [\![\mathbf{cons}]\!]$$

$$
\begin{array}{lll}
\mathscr{S}'\ I\ V\ [\![\mathbf{head}]\!] & = \lambda c.c\ (\lambda c_1.\lambda x.(\mathbf{conv}_{(dec\ I)}\ (\mathbf{head}\ x))\ c_1) & \text{if } 1 \in I \\
 & = \lambda c.c\ (\lambda c_1.\lambda x.x(\lambda v.(\mathbf{conv}_{(dec\ I)}\ (\mathbf{head}\ v))\ c_1)) & \text{if } 1 \notin I
\end{array}
$$

$$
\begin{array}{ll}
\mathscr{S}'\ I\ V\ [\![E_1\ E_2]\!] & = \lambda c.\mathscr{S}'\ \emptyset\ V\ [\![E_2]\!]\ (\lambda v.\mathscr{S}'\ (\{1\} \bigcup (inc\ I))\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ v)) \\
 & \text{if } \forall \rho : drop\ (\mathbf{S}\ [\![E_1]\!]\ \rho)\ \bot = \bot \\
 & = \lambda c.\mathscr{S}'\ (inc\ I)\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{S}'\ \emptyset\ V\ [\![E_2]\!])) \\
 & \text{otherwise} \\
\mathscr{S}'\ I\ V\ [\![\lambda x.E]\!] & = \lambda c.c\ (\lambda c.\lambda x.\mathscr{S}'\ (dec\ I)\ (V \bigcup \{x\})\ [\![E]\!]\ c) \quad \text{if } 1 \in I \\
 & = \lambda c.c\ (\lambda c.\lambda x.\mathscr{S}'\ (dec\ I)\ (V \setminus \{x\})\ [\![E]\!]\ c) \quad \text{if } 1 \notin I
\end{array}
$$

$$
\begin{array}{l}
\mathscr{S}'\ I\ V\ [\![\mathbf{fix}_{\sigma}\ (\lambda x.E)]\!] \\
\quad = \mathbf{sel}_{I,V\setminus\{x\}}(\mathbf{fix}\ (\lambda(x_1 : \mathsf{B}'_I\ [\![\sigma]\!], \ldots, x_n : \mathsf{B}'_I\ [\![\sigma]\!]).(\mathscr{S}'\ I_1\ V_1\ [\![E'_1]\!], \ldots, \mathscr{S}'\ I_n\ V_n\ [\![E'_n]\!])))
\end{array}
$$

Translation of Terms

Fig. 5. The cps-conversion using strictness and evaluation information.

$$
\begin{array}{l}
\mathsf{C}'_{\{2\}}\ [\![int \to int \to bool]\!] \\
\quad = \mathsf{U}'_{\{2\}}\ [\![int \to int \to bool]\!] \to \mathbf{Ans} \\
\quad = (\mathsf{C}'_{\{1\}}\ [\![int \to bool]\!] \to \mathsf{B}\ [\![int]\!] \to \mathbf{Ans}) \to \mathbf{Ans} \\
\quad = ((\mathsf{U}'_{\{1\}}\ [\![int \to bool]\!] \to \mathbf{Ans}) \to \mathsf{B}\ [\![int]\!] \to \mathbf{Ans}) \to \mathbf{Ans} \\
\quad = (((\mathsf{C}'_{\emptyset}\ [\![bool]\!] \to \mathsf{U}\ [\![int]\!] \to \mathbf{Ans}) \to \mathbf{Ans}) \to \mathsf{B}\ [\![int]\!] \to \mathbf{Ans}) \to \mathbf{Ans}
\end{array}
$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

$\mathsf{C}'_\emptyset\ [\![bool]\!]$ is the type of the continuation, $\mathsf{U}\ [\![int]\!]$ is the type of the second argument, which is unboxed because 2 is in the initial subscript, and $\mathsf{B}\ [\![int]\!]$ is the type of the first argument, which is boxed.

It is easy to see that:

$$\mathsf{U}'_\emptyset\ [\![\sigma]\!] = \mathsf{U}\ [\![\sigma]\!]$$

$$\mathsf{C}'_\emptyset\ [\![\sigma]\!] = \mathsf{C}\ [\![\sigma]\!]$$

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

The compilation rule $\mathscr{S}'$ 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 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 $\mathbf{conv}_I$, which is defined as follows.

Definition 2.4

$$\begin{array}{lcl} \mathbf{conv}_I^\sigma & : & \mathsf{B}\ [\![\sigma]\!] \to \mathsf{B}'_I\ [\![\sigma]\!] \\ \mathbf{conv}_I^\sigma & = & \lambda H : \mathsf{B}\ [\![\sigma]\!].\lambda c : \mathsf{C}'_I\ [\![\sigma]\!].H\ (W_\sigma\ c\ I) \end{array}$$

$$\begin{array}{lcll} W_\sigma & : & \mathsf{C}'_I\ [\![\sigma]\!] \to Intset \to \mathsf{C}\ [\![\sigma]\!] & \\ W_\sigma\ c\ \emptyset & = & c & \\ W_{\tau\to\sigma}\ c\ I & = & \lambda F.c\ (\lambda c.\ \lambda v.F\ (W_\sigma\ c\ (dec\ I))\ (\lambda c.c\ v)) & if\ 1 \in I \\ W_{\tau\to\sigma}\ c\ I & = & \lambda F.c\ (\lambda c.\ \lambda v.F\ (W_\sigma\ c\ (dec\ I))\ v) & if\ 1 \notin I. \end{array}$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

$(W_\sigma\ c\ I)$ is passed as the continuation of the expression to be converted. It is responsible for reconstructing the expression with the new type. Let us take a simple example to illustrate the definition of $\mathbf{conv}_I$. We have:

$$\begin{aligned}\mathscr{N}\ [\![\lambda x.\mathbf{plus}\ x\ 1]\!] &= \lambda c.c(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1)) \\ &\quad : \mathsf{B}\ [\![int \to int]\!] \\ \mathscr{S}'\ \{1\}\ \emptyset\ [\![\lambda x.\mathbf{plus}\ x\ 1]\!] &= \lambda c.c(\lambda c.\lambda x.\mathbf{plus_c}\ c\ x\ 1) \\ &\quad : \mathsf{B}'_{\{1\}}\ [\![int \to int]\!]\end{aligned}$$

The first equality was established at the beginning of this section and the second one is shown below. We apply $\mathbf{conv}^{int\to int}_{\{1\}}$ to $\lambda c.c(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1))$ and show that it returns $\lambda c.c(\lambda c.\lambda x.\mathbf{plus_c}\ c\ x\ 1)$.

$$\begin{aligned}&\mathbf{conv}^{int\to int}_{\{1\}}(\lambda c.c(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1))) \\ &= (\lambda H.\lambda c.H\ (W_{int\to int}\ c\ \{1\}))(\lambda c.c(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1))) \\ &= (\lambda H.\lambda c.H\ (\lambda F.\ c\ (\lambda c.\ \lambda v.\ F\ c\ (\lambda c.\ c\ v))))(\lambda c.c(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1))) \\ &= \lambda c.(\lambda c.c(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1)))(\lambda F.\ c\ (\lambda c.\ \lambda v.\ F\ c\ (\lambda c.\ c\ v))) \\ &= \lambda c.(\lambda F.\ c\ (\lambda c.\ \lambda v.\ F\ c\ (\lambda c.\ c\ v)))(\lambda c.\lambda x.x\ (\lambda m.\mathbf{plus_c}\ c\ m\ 1)) \\ &= \lambda c.c(\lambda c.\lambda v.\mathbf{plus_c}\ c\ v\ 1)\end{aligned}$$

So $\mathbf{conv}^{int\to int}_{\{1\}}$ translates a function expecting a boxed argument into a function expecting an unboxed argument.

The types are made explicit only when they are useful. Variables of type $\mathsf{B}'_I\ [\![\sigma]\!]$ (in the third case of the rule for variables) are fixed point variables.

A simpler version of the rule for the fixed point operator would be:

$$\mathscr{S}'\ I\ V\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] = \mathbf{fix}\ (\lambda x.\mathscr{S}'\ I\ V\ [\![E]\!])$$

This rule is not sufficient, however, because the variable being fixpointed may appear in several different contexts in E. This justifies the more complicated rule given in Figure 5. For any program, there are only a finite number of contexts, because both I and V contain at most as many elements as the largest arity of a function in a program. For $1 \le i \le n$, the variable x_i stands for x in the context I_i and V_i, and E'_k is obtained from E by replacing each occurrence of x by the appropriate x_i. The function $\mathbf{sel}_{I,V}$ selects the term from the n-tuple returned by the fixed point which corresponds to the context I and V.

We use again the expression E and F defined above to illustrate $\mathscr{S}'$. Note that the result of this translation is indeed the expression we were aiming at at the beginning of the section.

$$\begin{aligned}\mathscr{S}'\ \emptyset\ \emptyset\ [\![E]\!] &= \mathscr{S}'\ \emptyset\ \emptyset\ [\![F\ (\mathbf{plus}\ 2\ 7)]\!] \\ &= \lambda c.\mathscr{S}'\ \emptyset\ \emptyset\ [\![\mathbf{plus}\ 2\ 7]\!]\ (\lambda v.\mathscr{S}'\ \{1\}\ \emptyset\ [\![F]\!]\ (\lambda f.f\ c\ v)) \\ &= \lambda c.(\lambda c.\mathbf{plus_c}\ c\ 2\ 7)\ (\lambda v.\mathscr{S}'\ \{1\}\ \emptyset\ [\![F]\!]\ (\lambda f.f\ c\ v)) \\ &= \lambda c.\mathbf{plus_c}\ (\lambda v.\mathscr{S}'\ \{1\}\ \emptyset\ [\![F]\!]\ (\lambda f.f\ c\ v))\ 2\ 7 \\ &= \lambda c.\mathbf{plus_c}\ (\lambda v.(\lambda c.c\ F_2)\ (\lambda f.f\ c\ v))\ 2\ 7 \\ &= \lambda c.\mathbf{plus_c}\ (\lambda v.F_2\ c\ v)\ 2\ 7 \\ &= \lambda c.\mathbf{plus_c}\ (F_2\ c)\ 2\ 7\end{aligned}$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

$$\begin{aligned}
\mathscr{S}'\ \{1\}\ \emptyset\ [\![F]\!] &= \mathscr{S}'\ \{1\}\ \emptyset\ [\![\lambda x.\mathbf{plus}\ x\ 1]\!] \\
&= \lambda c.c(\lambda c.\lambda x.\mathscr{S}'\ \emptyset\ \{x\}\ [\![\mathbf{plus}\ x\ 1]\!]\ c) \\
&= \lambda c.c\ F_2 \\
F_2 \qquad &= \lambda c.\lambda x.\mathscr{S}'\ \emptyset\ \{x\}\ [\![\mathbf{plus}\ x\ 1]\!]\ c \\
&= \lambda c.\lambda x.\mathscr{S}'\ \emptyset\ \{x\}\ [\![1]\!]\ (\lambda v.\mathscr{S}'\ \{1\}\ \{x\}\ [\![\mathbf{plus}\ x]\!]\ (\lambda f.f\ c\ v)) \\
&= \lambda c.\lambda x.(\lambda c.c\ 1)\ (\lambda v.\mathscr{S}'\ \{1\}\ \{x\}\ [\![\mathbf{plus}\ x]\!]\ (\lambda f.f\ c\ v)) \\
&= \lambda c.\lambda x.\mathscr{S}'\ \{1\}\ \{x\}\ [\![\mathbf{plus}\ x]\!]\ (\lambda f.f\ c\ 1) \\
&= \lambda c.\lambda x.(\lambda c.\mathscr{S}'\ \emptyset\ \{x\}\ [\![x]\!]\ (\lambda v.\mathscr{S}'\ \{1,2\}\ \{x\}\ [\![\mathbf{plus}]\!](\lambda f.f\ c\ v)))(\lambda f.f\ c\ 1) \\
&= \lambda c.\lambda x.(\lambda c.\mathscr{S}'\ \{1,2\}\ \{x\}\ [\![\mathbf{plus}]\!]\ (\lambda f.f\ c\ x))(\lambda f.f\ c\ 1) \\
&= \lambda c.\lambda x.\mathscr{S}'\ \{1,2\}\ \{x\}\ [\![\mathbf{plus}]\!]\ (\lambda f.f\ (\lambda f.f\ c\ 1)\ x) \\
&= \lambda c.\lambda x.\mathbf{plus_C}\ c\ x\ 1
\end{aligned}$$

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

Definition 2.5 *If ρ is a type environment, then its transformation $\mathscr{S}'_V\ [\![\rho]\!]$ is defined by the rule:*

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

Theorem 2.6

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

It is easy to see that $\mathscr{S}'_\emptyset\ [\![\rho]\!] = \mathscr{N}\ [\![\rho]\!]$.

The correctness of $\mathscr{S}'$ is formulated in the following theorem, which is proved in Appendix 1.

Theorem 2.7 *For all terms $E : \sigma$,*

$$\mathbf{S}\ [\![\mathscr{S}'\ I\ V\ [\![E]\!]]\!]\ \rho = \mathbf{S}\ [\![\mathbf{conv}_I^\sigma\ (\mathscr{S}\ [\![E]\!])]\!]\ (Box_V\ \rho),$$

where Box_V boxes basic values which are unboxed.

3 Using more sophisticated evaluation 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:

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

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

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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 (Burn, 1991b; Jensen, 1992b; Leung and Mishra, 1991; Nielson and Nielson, 1992; Wadler and Hughes, 1987; Wadler, 1987). 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:

(a) 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
(b) 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 (Burn, 1991a).

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.

† Results from some initial experiments investigating this question can be found elsewhere (Finne and Burn, 1993).

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathscr{T}\ V\ i\ Q_j\ [\![x]\!] \quad = x \quad \text{if } x \notin V \\
&\qquad\qquad\quad = x_{ij} \quad \text{if } x \in V \\
&\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{k}_\sigma]\!] \quad = \mathscr{N}\ [\![\mathbf{k}_\sigma]\!] \text{ if } \mathbf{k}_\sigma \neq \mathbf{if} \\
&\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] \\
&\quad = \lambda c.\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]\ (\mathbf{if_C}\ (\mathscr{T}\ V\ i\ Q_j\ [\![E_2]\!]\ c)\ (\mathscr{T}\ V\ i\ Q_j\ [\![E_3]\!]\ c)) \\
&\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!] \\
&\quad = \lambda c.\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]\ (\lambda v.\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
&\quad \text{if } \forall \rho,\ \forall v_0 \in P,\ \forall v_1,\ \ldots,\ v_i : drop(\ldots(drop(\mathbf{S}\ [\![E_1]\!]\ \rho)v_0)\ \ldots)v_i \in Q_j \\
&\quad = \lambda c.\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!])) \\
&\quad \text{otherwise} \\
&\mathscr{T}\ V\ 0\ Q_j\ [\![\lambda x.E]\!] \quad = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\tau}\}\ [\![E]\!]\ c) \\
&\mathscr{T}\ V\ (i+1)\ Q_j\ [\![\lambda x.E]\!] \quad = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]\ c) \\
&\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!] \\
&\quad = \quad \mathbf{sel}_{ij}(\mathbf{fix}\ (\lambda(x_{i_1j_1},\ldots,x_{i_nj_n}).(\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!],\ \ldots,\ \mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!]))) \\
&\qquad \text{where } W = V \bigcup \{x\}
\end{aligned}
$$

Fig. 6. The cps-conversion using Scott-closed set information.

The key idea behind the transformation rules defined in Figure 6 is the following fact, sometimes known as the Evaluation Transformer Theorem (Burn, 1991a, Theorem 7.5).

Fact 3.2 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 $\mathscr{T}\ \emptyset\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E]\!]$. The third argument to the translation function $\mathscr{T}$ 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 $\mathscr{T}$ collects the set of fixed point variables as these need to be distinguished from lambda-bound variables when proving the correctness of this translation. The second argument to $\mathscr{T}$ is introduced 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_j 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

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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:

1. The type of $\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]$ is the same as the type of $\mathscr{N}\ [\![E]\!]$.
2. 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.
3. 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 3.2). 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_j; $(E_1\ E_2)$ 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 second 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 Mycroft (1981), Burn *et al.* (1986), Wadler and Hughes (1987), Hunt (1991), Jensen (1992a) and Leung and Mishra (1991), 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 $\{\perp_{\mathbf{S}_\sigma}\}$ 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.
4. 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_j must be $\{\perp_{\mathbf{S}_{\sigma\to\tau}}\}$, 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.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

5. The rule for fixed points is again quite complicated, for similar reasons to those discussed in section 2.2. A simpler rule would be:

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

which would make the first argument of $\mathscr{T}$ unnecessary. This rule is not satisfactory because 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:

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

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_{ij} stands for x in in the context where it is applied to i arguments and has evaluation context Q_j. The function $\mathbf{sel}_{ij}$ selects the term from the n-tuple returned by the fixed point which corresponds to i and the evaluation context Q_j.
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 we are calculating $\mathscr{T}\ V\ 0\ Q_j\ [\![E]\!]$ where E is the expression $(\lambda x_1 \ldots \lambda x_n.D)\ E_1\ \ldots E_n$. 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 $\mathscr{T}\ V\ 0\ Q_j\ [\![D]\!]$, which says that the inner application is to be evaluated in the context given by Q_j. This corresponds to passing the evaluation context to a tail-call.

To illustrate the transformation $\mathscr{T}$, let us consider two common evaluation contexts BOT and INF (Burn, 1991a; Wadler, 1987). BOT contains only $\bot$ and INF is the Scott-closed set containing all infinite lists and lists ending with a $\bot$. In operational terms, BOT is the context corresponding to the evaluation up to WHNF and INF is the context indicating the evaluation of the whole structure of the expression. We assume that the transformation relies on a strictness analyser which is powerful enough to show that (see Wadler (1987) for such an analyser):

$$\forall x \in \mathbf{S}_\sigma,\ \ \forall y \in INF.\ \ \mathbf{cons}\ x\ y\ \in INF$$

$$\forall x \in INF.\ \ reverse\ x \in BOT$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

We show how the expression *reverse* (**cons** e_1 e_2) is transformed by $\mathscr{T}$.

$$
\begin{aligned}
T_1 &= \mathscr{T}\ \emptyset\ 0\ BOT\ [\![reverse\ (\mathbf{cons}\ e_1\ e_2)]\!] \\
&= \lambda c.\mathscr{T}\ \emptyset\ 0\ INF\ [\![(\mathbf{cons}\ e_1\ e_2)]\!]\ T_2 \\
T_2 &= \lambda v.\mathscr{T}\ \emptyset\ 1\ BOT\ [\![reverse]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v)) \\
T_1 &= \lambda c.(\lambda c.\mathscr{T}\ \emptyset\ 0\ INF\ [\![e_2]\!]\ (\lambda v.\mathscr{T}\ \emptyset\ 1\ INF\ [\![\mathbf{cons}\ e_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v))))\ T_2 \\
&= \lambda c.\mathscr{T}\ \emptyset\ 0\ INF\ [\![e_2]\!]\ (\lambda v.\mathscr{T}\ \emptyset\ 1\ INF\ [\![\mathbf{cons}\ e_1]\!]\ (\lambda f.f\ T_2\ (\lambda c.c\ v))) \\
&= \lambda c.\mathscr{T}\ \emptyset\ 0\ INF\ [\![e_2]\!] \\
&\qquad (\lambda v.(\lambda c.\mathscr{T}\ \emptyset\ 2\ INF\ [\![cons]\!]\ (\lambda f.f\ c\ (\mathscr{T}\ \emptyset\ 0\ BOT\ [\![e_1]\!])))(\lambda f.f\ T_2\ (\lambda c.c\ v))) \\
&= \lambda c.\mathscr{T}\ \emptyset\ 0\ INF\ [\![e_2]\!] \\
&\qquad (\lambda v.\mathscr{T}\ \emptyset\ 2\ INF\ [\![cons]\!]\ (\lambda f.f\ (\lambda f.f\ T_2\ (\lambda c.c\ v))\ (\mathscr{T}\ \emptyset\ 0\ BOT\ [\![e_1]\!]))) \\
&= \lambda c.\mathscr{T}\ \emptyset\ 0\ INF\ [\![e_2]\!]\ (\lambda v.\mathbf{cons_C}\ T_2\ (\mathscr{T}\ \emptyset\ 0\ BOT\ [\![e_1]\!])\ \ (\lambda c.c\ v))
\end{aligned}
$$

The result shows that e_2 is executed first, with evaluation context INF (which means that the whole structure of e_2 can be computed). The result of the evaluation is then passed to $\mathbf{cons_C}$ with the unevaluated argument e_1. This is the expected evaluation order since *reverse* cannot produce any result if e_2 is an infinite list. This contrasts with e_1 whose termination should not affect the termination of the whole expression (so its evaluation has to be postponed).

The following theorem gives the correctness of our translation. It states that translating a term with $\mathscr{T}$ gives essentially the same result as translating it with $\mathscr{N}$. The proof of this theorem is presented in Appendix 2.

Theorem 3.3 *For all closed terms of ground type* $E : \sigma$,
$\mathbf{S}\ [\![\mathscr{T}\ \emptyset\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!]\ \emptyset = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \emptyset$.

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 (Schmidt, 1980; Wand, 1982; Nielson and Nielson, 1988; Dybjer, 1985; Morris, 1973; Mosses, 1980; Thatcher *et al.*, 1981; Lester, 1987; Lester, 1988; Cousineau *et al.*, 1987; Fradet and Le Métayer, 1991); and step (2): proving the correctness of the result of the analysis (Cousot and Cousot, 1979; Cousot and Cousot, 1992b; Cousot and Cousot, 1992a; Burn, 1991b; Nielson, 1989; Wadler and Hughes, 1987; Leung and Mishra, 1991; Jensen, 1992a; Benton, 1992).

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 Nielson (1985) and Gerhart (1975). 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. It does not seem that this method is directly applicable to strictness analysis because only a weak equivalence is

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

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 Lester (1988) and Danvy and Hatcliff (1993). 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 Lester (1988), only simple strictness information is considered.

A less thoroughgoing attempt at this problem is also presented in Burn (1991b), 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 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 (Burn, 1991b, Sect. 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' (Burn, 1990). Again we should be able to modify the rule for application

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

to accommodate this. Instead of using $\mathscr{T}\ V\ 0\ \{\perp_{S_\sigma}\}\ [\![E_2]\!]$ in the case that the argument expression cannot be evaluated, this can be changed to $\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]$ 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.

Appendix A

In this appendix we prove Theorem 2.7 which establishes the correctness of the transformation $\mathscr{S}'$ presented in section 2. We first recall this theorem and give the definitions of Box_V and $\mathbf{conv}_I^\sigma$. Box_V applies to environments and is used to replace any unboxed value by the corresponding boxed value. V is the set of unboxed variables. The function $\mathbf{conv}_I^\sigma$ transforms an expression of type $\mathsf{B}\ [\![\sigma]\!]$ into an expression of type $\mathsf{B}'_I\ [\![\sigma]\!]$. I is the set of argument positions for which unboxed values are going to be supplied, so if σ is of function type, then $\mathbf{conv}_I^\sigma\ (\mathscr{S}\ [\![E]\!])$ is a new expression which accepts these arguments unboxed.

Theorem A.1 *For all terms* $E : \sigma$,

$$\mathsf{S}\ [\![\mathscr{S}'\ I\ V\ [\![E]\!]]\!]\ \rho = \mathsf{S}\ [\![\mathbf{conv}_I^\sigma\ (\mathscr{S}\ [\![E]\!])]\!]\ (Box_V\ \rho).$$

Definition A.2

$$\begin{array}{rcll} (Box_V\ \rho)\ x & = & box_\sigma(\rho\ x) & \text{if } x : \sigma \in V \\ & = & x & \text{if } x \notin V \end{array}$$

$$\begin{array}{rcl} box_\sigma & : & \mathsf{U}\ [\![\sigma]\!] \to \mathsf{B}\ [\![\sigma]\!] \\ box_\sigma\ x & = & \lambda c.c\ x \end{array}$$

Definition A.3

$$\begin{array}{rcl} \mathbf{conv}_I^\sigma & : & \mathsf{B}\ [\![\sigma]\!] \to \mathsf{B}'_I\ [\![\sigma]\!] \\ \mathbf{conv}_I^\sigma & = & \lambda H : \mathsf{B}\ [\![\sigma]\!].\lambda c : \mathsf{C}'_I\ [\![\sigma]\!].H\ (W_\sigma\ c\ I) \end{array}$$

$$\begin{array}{rcll} W_\sigma & : & \mathsf{C}'_I\ [\![\sigma]\!] \to Intset \to \mathsf{C}\ [\![\sigma]\!] & \\ W_\sigma\ c\ \emptyset & = & c & \\ W_{\tau\to\sigma}\ c\ I & = & \lambda F.c\ (\lambda c.\ \lambda v.F\ (W_\sigma\ c\ (dec\ I))\ (\lambda c.c\ v)) & \text{if } 1 \in I \\ W_{\tau\to\sigma}\ c\ I & = & \lambda F.c\ (\lambda c.\ \lambda v.F\ (W_\sigma\ c\ (dec\ I))\ v) & \text{if } 1 \notin I. \end{array}$$

We prove Theorem A.1 by structural induction on E. For the sake of conciseness, types are often omitted when not really useful. Syntactic and semantic values of

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

basic operators are distinguished using respectively boldface and italics fonts (as in $\mathbf{conv}_I^\sigma$ and $conv_I^\sigma$).

1. $\mathscr{S}'\ I\ V\ [\![x]\!] = \mathbf{conv}_I^\sigma\ (\lambda c.c\ x)$ if $x : \mathsf{U}\ [\![\sigma]\!] \in V$
   $= \mathbf{conv}_I^\sigma\ x$ if $x : \mathsf{B}\ [\![\sigma]\!] \notin V$
   $= x$ if $x : \mathsf{B}'_I\ [\![\sigma]\!]$.

   Let us assume $x : \mathsf{U}\ [\![\sigma]\!] \in V$ (the other cases are similar).

   $$\begin{aligned}\mathbf{S}\ [\![\mathscr{S}'\ I\ V\ [\![x]\!]]\!]\ \rho &= \mathbf{S}\ [\![\mathbf{conv}_I^\sigma\ (\lambda c.c\ x)]\!]\ \rho \\ &= conv_I^\sigma\ (\lambda c.c\ (\rho\ x)) \\ \mathbf{S}\ [\![\mathbf{conv}_I^\sigma\ (x)]\!]\ (Box_V\ \rho) &= conv_I^\sigma\ (\lambda c.c\ (\rho\ x)),\end{aligned}$$

2. $\mathscr{S}'\ I\ V\ [\![\mathbf{plus}]\!] = \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.\mathbf{plus_c}\ c_2\ x\ y))$
   if $1 \in I \wedge 2 \in I$
   $= \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.\mathbf{plus_c}\ c_2\ m\ y)))$
   if $1 \notin I \wedge 2 \in I$
   $= \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.y\ (\lambda m.\mathbf{plus_c}\ c_2\ x\ m)))$
   if $1 \in I \wedge 2 \notin I$
   $= \lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.\mathbf{plus_c}\ c_2\ m\ n))))$
   if $1 \notin I \wedge 2 \notin I$.

   We consider only the case $1 \notin I \wedge 2 \in I$, the other cases can be treated in a similar way. Let $int3 = int \to int \to int$.

   $$\begin{aligned}&\mathbf{S}\ [\![\mathbf{conv}_{\{2\}}^{int3}\ (\mathscr{S}\ [\![\lambda x.\lambda y.plus\ x\ y]\!])]\!]\ (Box_V\ \rho) \\ &= conv_{\{2\}}^{int3}(\lambda c.c\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.plus_c\ c_2\ m\ n))))) \\ &= \lambda c.(Wc\ \{2\})\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.plus_c\ c_2\ m\ n)))) \\ &= \lambda c.(\lambda F.c\ (\lambda c.F(Wc\ \{1\})))\ (\lambda c_1.\lambda x.c_1\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.plus_c\ c_2\ m\ n)))) \\ &= \lambda c.c\ (\lambda c.\lambda x.(Wc\ \{1\})\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.plus_c\ c_2\ m\ n)))) \\ &= \lambda c.c\ (\lambda c.\lambda x.c\ (\lambda c.\lambda v.\ (\lambda c_2.\lambda y.x\ (\lambda m.y\ (\lambda n.plus_c\ c_2\ m\ n)))\ c\ (\lambda c.c\ v))) \\ &= \lambda c.c\ (\lambda c.\lambda x.c\ (\lambda c.\lambda v.x\ (\lambda m.(\lambda c.c\ v)(\lambda n.plus_c\ c\ m\ n)))) \\ &= \lambda c.c\ (\lambda c.\lambda x.c\ (\lambda c.\lambda v.x\ (\lambda m.plus_c\ c\ m\ v))) \\ &= \mathbf{S}\ [\![\mathscr{S}'\ \{2\}\ V\ [\![\mathbf{plus}]\!]]\!]\ \rho.\end{aligned}$$

3. $\mathscr{S}'\ I\ V\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] = \lambda c.\mathscr{S}'\ \emptyset\ V\ [\![E_1]\!](\mathbf{if_c}(\mathscr{S}'\ I\ V\ [\![E_2]\!]c)(\mathscr{S}'\ I\ V\ [\![E_3]\!]c))$

   $$\begin{aligned}&\mathbf{S}\ [\![\mathbf{conv}_I\ (\mathscr{S}\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!])]\!]\ (Box_V\ \rho) \\ &= \mathbf{S}\ [\![\mathbf{conv}_I\ (\lambda c.\mathscr{S}\ [\![E_1]\!]\ (\mathbf{if_c}\ (\mathscr{S}\ [\![E_2]\!]\ c)\ (\mathscr{S}\ [\![E_3]\!]\ c)))]\!]\ (Box_V\ \rho) \\ &= conv_I(\lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho) \\ &\quad (if_c((\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho))c)((\mathbf{S}\ [\![\mathscr{S}\ [\![E_3]\!]]\!]\ (Box_V\ \rho))c))) \\ &= \lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho) \\ &\quad (if_c((\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho))(W\ c\ I))((\mathbf{S}\ [\![\mathscr{S}\ [\![E_3]\!]]\!]\ (Box_V\ \rho))(W\ c\ I))) \\ &= \lambda c.\mathbf{S}\ [\![\mathbf{conv}_\emptyset(\mathscr{S}\ [\![E_1]\!])]\!]\ (Box_V\ \rho) \\ &\quad (if_c((\mathbf{S}\ [\![\mathbf{conv}_I(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho))\ c)((\mathbf{S}\ [\![\mathbf{conv}_I(\mathscr{S}\ [\![E_3]\!])]\!]\ (Box_V\ \rho))\ c)) \\ &= \lambda c.\mathbf{S}\ [\![\mathscr{S}'\ I\ V\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!]]\!]\ \rho \\ &\quad \text{from the induction hypothesis.}\end{aligned}$$

4. $\mathscr{S}'\ I\ V\ [\![\mathbf{cons}]\!] = \mathscr{S}\ [\![\mathbf{cons}]\!]$.
   We must have $I = \emptyset$ in this case (because *cons* is not strict) and $conv_\emptyset = id$ implies the desired result.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

5. $\mathscr{S}'\ I\ V\ [\![\mathbf{head}]\!] = \lambda c.c\ (\lambda c_1.\lambda x.(\mathbf{conv}_{(dec\ I)}\ (\mathbf{head}\ x))\ c_1)$ if $1 \in I$
$= \lambda c.c\ (\lambda c_1.\lambda x.x(\lambda v.(\mathbf{conv}_{(dec\ I)}\ (\mathbf{head}\ v))\ c_1))$ if $1 \notin I$.

Let us consider the case $1 \in I$.

$$
\begin{aligned}
&\mathbf{S}\ [\![\mathbf{conv}_I\ (\mathscr{S}\ [\![\mathbf{head}]\!])]\!]\ (Box_V\ \rho)\\
&\quad = conv_I(\lambda c.c\ (\lambda c_1.\lambda x.x\ (\lambda v.head\ v\ c_1)))\\
&\quad = \lambda c.(W\ c\ I)(\lambda c_1.\lambda x.x\ (\lambda v.head\ v\ c_1))\\
&\quad = \lambda c.c\ (\lambda c.\lambda v.(\lambda c.c\ v)(\lambda v.head\ v\ (W\ c\ (decI))))\\
&\qquad (since\ 1\ \in\ I)\\
&\quad = \lambda c.c\ (\lambda c.\lambda v.head\ v\ (W\ c\ (decI)))\\
&\quad = \lambda c.c\ (\lambda c.\lambda v.conv_{(dec\ I)}(head\ v))\ c)\\
&\quad = \mathbf{S}\ [\![\mathscr{S}'\ I\ V\ [\![\mathbf{head}]\!]]\!]\ \rho.
\end{aligned}
$$

6. $\mathscr{S}'\ I\ V\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{S}'\ \emptyset\ V\ [\![E_2]\!]\ (\lambda v.\mathscr{S}'\ (\{1\} \bigcup (inc\ I))\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ v))$
if $\forall \rho : drop\ (\mathbf{S}\ [\![E_1]\!]\ \rho)\ \bot = \bot$
$= \lambda c.\mathscr{S}'\ (inc\ I)\ V\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{S}'\ \emptyset\ V\ [\![E_2]\!]))$
otherwise.

We consider first the case $\forall \rho : drop\ (\mathbf{S}\ [\![E_1]\!]\ \rho)\ \bot = \bot$.

$$
\begin{aligned}
&\mathbf{S}\ [\![\mathbf{conv}_I\ (\mathscr{S}\ [\![E_1\ E_2]\!])]\!]\ (Box_V\ \rho)\\
&\quad = conv_I(\lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho)\\
&\qquad (\lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.f\ c\ (\lambda c.c\ v))))\\
&\quad = \lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho)\\
&\qquad (\lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.f\ (W\ c\ I)\ (\lambda c.c\ v)))\\
&\quad = \lambda c.\mathbf{S}\ [\![\mathbf{conv}_{\emptyset}(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho)\\
&\qquad (\lambda v.\mathbf{S}\ [\![\mathbf{conv}_{(\{1\} \bigcup (inc\ I))}(\mathscr{S}\ [\![E_1]\!])]\!]\ (Box_V\ \rho)\ (\lambda f.f\ c\ v)).
\end{aligned}
$$

This is because:

$$
\begin{aligned}
&\mathscr{S}\ [\![\mathbf{conv}_{(\{1\} \bigcup (inc\ I))}(\mathscr{S}\ [\![E_1]\!])]\!](Box_V\ \rho)\ (\lambda f.f\ c\ v)\\
&\quad = (\lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!](W\ c\ (\{1\} \bigcup (inc\ I)))]\!]\ (Box_V\ \rho))\ (\lambda f.f\ c\ v)\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (W\ (\lambda f.f\ c\ v)(\{1\} \bigcup (inc\ I)))\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.\ (\lambda f.f\ c\ v)(\lambda c.\lambda v.f\ (W\ c\ I)(\lambda c.c\ v)))\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.f\ (W\ c\ I)\ (\lambda c.c\ v)).
\end{aligned}
$$

The desired result follows from the derivation above by structural induction. Let us now consider the second case in the rule for application.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

$$\begin{aligned}
&\mathbf{S}\ [\![\mathbf{conv}_I\ (\mathscr{S}\ [\![E_1\ E_2]\!])]\!]\ (Box_V\ \rho)\\
&\quad = conv_I(\lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.f\ c\ (\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho))))\\
&\quad = \lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.f\ (W\ c\ I)\ (\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho)))\\
&\quad = \lambda c.\mathbf{S}\ [\![\mathbf{conv}_{(inc\ I)}(\mathscr{S}\ [\![E_1]\!])]\!]\ (Box_V\ \rho)\\
&\qquad (\lambda f.f\ c\ (\mathbf{S}\ [\![\mathbf{conv}_\emptyset(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho)))\\
&\qquad \text{This is because:}\\
&\mathbf{S}\ [\![\mathbf{conv}_{(inc\ I)}(\mathscr{S}\ [\![E_1]\!])]\!]\ (Box_V\ \rho)\ (\lambda f.f\ c\ (\mathbf{S}\ [\![\mathbf{conv}_\emptyset(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho)))\\
&\quad = (\lambda c.\mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\\
&\qquad (W\ c\ (inc\ I)))(\lambda f.f\ c\ (\mathbf{S}\ [\![\mathbf{conv}_\emptyset(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho)))\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\\
&\qquad (W\ (\lambda f.f\ c\ (\mathbf{S}\ [\![\mathbf{conv}_\emptyset(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho)))\ (inc\ I))\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)(\lambda f.f\ (W\ c\ I)\ (\mathbf{S}\ [\![\mathbf{conv}_\emptyset(\mathscr{S}\ [\![E_2]\!])]\!]\ (Box_V\ \rho)))\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E_1]\!]]\!]\ (Box_V\ \rho)\ (\lambda f.f\ (W\ c\ I)\ (\mathbf{S}\ [\![\mathscr{S}\ [\![E_2]\!]]\!]\ (Box_V\ \rho))).
\end{aligned}$$

And the desired result follows from the derivation above by structural induction.

7. $$\begin{aligned}\mathscr{S}'\ I\ V\ [\![\lambda x.E]\!] &= \lambda c.c\ (\lambda c.\lambda x.\mathscr{S}'\ (dec\ I)\ (V \bigcup \{x\})\ [\![E]\!]\ c) \quad \text{if } 1 \in I\\ &= \lambda c.c\ (\lambda c.\lambda x.\mathscr{S}'\ (dec\ I)\ (V \setminus \{x\})\ [\![E]\!]\ c) \quad \text{if } 1 \notin I.\end{aligned}$$

We consider only the case $1 \in I$ (the other case can be proven in the same way, replacing v by $\lambda c.c\ v$).

$$\begin{aligned}
&\mathbf{S}\ [\![\mathbf{conv}_I\ (\mathscr{S}\ [\![\lambda x.E]\!])]\!]\ (Box_V\ \rho)\\
&\quad = conv_I(\lambda c.c\ (\lambda c.\ \lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ ((Box_V\ \rho)[v/x])\ c))\\
&\quad = \lambda c.(W\ c\ I)\ (\lambda c.\ \lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ ((Box_V\ \rho)[v/x])\ c)\\
&\quad = \lambda c.c\ (\lambda c.\ \lambda v.(\lambda c.\ \lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ ((Box_V\ \rho)[v/x])\ c)(W\ c\ (dec\ I))(\lambda c.c\ v))\\
&\quad = \lambda c.c\ (\lambda c.\ \lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ ((Box_V\ \rho)[(\lambda c.c\ v)/x])(W\ c\ (dec\ I)))\\
&\quad = \lambda c.c\ (\lambda c.\ \lambda v.\mathbf{S}\ [\![\mathbf{conv}_{dec\ I}(\mathscr{S}\ [\![E]\!])]\!]\ Box_{(V \bigcup \{x\})}(\rho[v/x])\ c).
\end{aligned}$$

This is because:

$$\begin{aligned}
&\mathbf{S}\ [\![\mathbf{conv}_{dec\ I}(\mathscr{S}\ [\![E]\!])]\!]\ Box_{(V \bigcup \{x\})}(\rho[v/x])\ c\\
&\quad = \mathbf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ ((Box_V\ \rho)[box(v)/x])(W\ c\ (dec\ I)).
\end{aligned}$$

The result then follows by structural induction.

8. $$\begin{aligned}
&\mathscr{S}'\ I\ V\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!]\\
&= \mathbf{sel}_{I,V\setminus\{x\}}\\
&\quad (\mathbf{fix}\ (\lambda(x_1 : \mathsf{B}'_I\ [\![\sigma]\!], \ldots, x_n : \mathsf{B}'_I\ [\![\sigma]\!]).(\mathscr{S}'\ I_1\ V_1\ [\![E']\!], \ldots, \mathscr{S}'\ I_n\ V_n\ [\![E']\!]))
\end{aligned}$$

$$\begin{aligned}
&\mathbf{S}\ [\![\mathbf{conv}_I\ (\mathscr{S}\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!])]\!]\ (Box_V\ \rho)\\
&\quad = conv_I(fix_\sigma(\lambda v.\mathbf{S}\ [\![\mathscr{S}\ [\![E]\!]]\!]\ (Box_{V\setminus\{x\}}\ (\rho[v/x]))))
\end{aligned}$$

$$\begin{aligned}
&\mathbf{S}\ [\![\mathscr{S}'\ I\ V\ [\![\mathbf{fix}_\sigma\ (\lambda x.E)]\!]]\!]\ \rho\\
&\quad = sel_{I,V\setminus\{x\}}\ (fix\ (\lambda(v_1 : \mathsf{B}'_I\ [\![\sigma]\!], \ldots, v_n : \mathsf{B}'_I\ [\![\sigma]\!]).\\
&\qquad (\mathbf{S}\ [\![\mathscr{S}'\ I_1\ V_1\ [\![E']\!]]\!]\ \rho[v_i/x_i], \ldots, \mathbf{S}\ [\![\mathscr{S}'\ I_n\ V_n\ [\![E']\!]]\!]\ \rho[v_i/x_i]))).
\end{aligned}$$

We prove the desired result by fixed point induction using the property

$$P(A_{IV}, B_{IV}) \Leftrightarrow (A_{IV} = conv_I\ B_{IV}).$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

The base case ($P(\perp,\perp) = \perp$) is obvious. We assume $P(A_{IV}, B_{IV})$ and we show $P(F(A_{IV}), G(B_{IV}))$ with F and G the transformers associated with the fixed point rules for $\mathscr{S}'$ and $\mathscr{S}$, respectively. The result is derived by structural induction, making use of the fixed point induction hypothesis.

Appendix B

In this appendix we prove Theorem 3.3, which establishes the correctness of the transformation $\mathscr{T}$ presented in section 3. The essential difference between transformations $\mathscr{N}$ and $\mathscr{T}$ lies in the rule for application. The rationale behind the proof is that $\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q\ [\![E]\!]]\!]\ \rho'$ cannot be more defined than $\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho$ (where ρ and ρ' are related environments) and the two expressions may differ only when $\mathbf{S}\ [\![E]\!]\ \overline{\rho} \in Q$. This captures the intuition that the evaluation of the expression transformed by $\mathscr{T}$ in a particular evaluation context may fail to terminate only if the semantic value of the expression belongs to the Scott-closed set defining the evaluation context. When $Q = \{\perp_{\mathbf{S}_\sigma}\}$, $V = \emptyset$, $i = 0$, $\rho = \rho' = \emptyset$ we get:

$$\mathbf{S}\ [\![\mathscr{T}\ \emptyset\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!]\ \emptyset \sqsubseteq \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \emptyset;$$

from Theorem B.7, and

$$\mathbf{S}\ [\![\mathscr{T}\ \emptyset\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!]\ \emptyset = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \emptyset \text{ or } \mathbf{S}\ [\![E]\!]\ \emptyset = \perp_{\mathbf{S}_\sigma}$$

from Theorem B.16.

Using a simple lemma (Lemma B.3) and these two theorems, we are able to conclude the desired result. It may seem strange that we have to prove Theorem B.7, because Theorem B.16 is stronger. However, the former is needed to prove the latter.

Before embarking on the proof, we make three important notes:

1. We assume that all the quantifications are over appropriately typed terms.
2. We do not put *lift* and *drop* in the statements of the theorems and proofs as it makes them unreadable. It is easy to take the theorems and proofs as presented here and make them correct with respect to using the lifted function space.
3. In our proofs we will reproduce the relevant clauses from the definition of $\mathscr{T}$ in Figure 6 to save the reader from having to refer back to the figure.

We use the notation $\Uparrow(E)$ to denote the fact that the normal order evaluation of E diverges. We have the following computational adequacy theorem (Gunter, 1992, Theorem 6.9).

Fact B.1 For all closed expressions $E : \sigma$, for all environments ρ,

$$\mathbf{S}\ [\![E]\!]\ \rho = \perp_{\mathbf{S}_\sigma} \text{ if and only if } \Uparrow(E).$$

The following lemma says that $\mathscr{N}\ [\![E]\!]$ has the same termination behaviour as E (Fradet, 1988, Lemma 4.2.4).

Lemma B.2 $\Uparrow(E)$ if and only if $\forall c.\ \Uparrow(\mathscr{N}\ [\![E]\!]\ c)$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

Lemma B.3 *For all closed terms* $E : \sigma$,

$$\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \emptyset = \perp_{\mathbf{S}_{\mathbf{B}[\![\sigma]\!]}} \Leftrightarrow \mathbf{S}\ [\![E]\!]\ \emptyset = \perp_{\mathbf{S}_\sigma}.$$

Proof Obvious from Fact B.1 and Lemma B.2.

The following fact is proved in (Muylaert Filho and Burn, 1993, Proposition 6.2) (ρ is supposed to be an appropriately typed environment):

Fact B.4 For all $E : \sigma$,

$$\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho \neq \perp_{\mathbf{S}_{\mathbf{B}[\![\sigma]\!]}} \quad \Longrightarrow \quad \exists v \text{ such that } \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho = \lambda c.c\ v.$$

In the statement and proofs of the main theorems it is useful to define a continuation which supplies arguments to a function. This is provided in the following definition.

Definition B.5

$$\begin{aligned} c^{[]} &= c \\ c^{[v_1,\ldots,v_i]} &= \lambda f.f\ c^{[v_2,\ldots,v_i]}\ v_1 \end{aligned}$$

The following definition allows us to state Theorem B.7 for the case of variables. $Pcontenv_V(\rho', \rho)$ establishes a relationship between environments ρ' and ρ to be used in the semantics of terms transformed by $\mathscr{T}$ and by $\mathscr{N}$ respectively. Variables x_{ij} correspond to occurrences of x in the original expression which are transformed by $\mathscr{T}$ in a context where they are applied to i arguments and have evaluation context Q_j.

Definition B.6 $Pcontenv_V(\rho', \rho)$ *if and only if*

1. $\forall x \in V, \forall i, j, \rho'\ x_{ij} \sqsubseteq \rho\ x$; and
2. $\forall x \notin V, \rho'\ x = \rho\ x$.

Theorem B.7 $\forall E : \sigma, \forall V, \forall i, \forall Q, \forall \rho', \rho$ *satisfying* $Pcontenv_V(\rho', \rho)$

$$\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q\ [\![E]\!]]\!]\ \rho' \sqsubseteq \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho.$$

Proof We prove this by induction over the rules defining $\mathscr{T}\ V\ i\ Q\ [\![E]\!]$. We follow the same convention as in Appendix 1 concerning syntactic and semantic values of primitive operators (using boldface fonts and italics fonts, respectively).

1. $$\begin{aligned} \mathscr{T}\ V\ i\ Q_j\ [\![x]\!] &= x \quad \text{if } x \notin V \\ &= x_{ij} \quad \text{if } x \in V \\ \mathscr{N}\ [\![x]\!] &= x \end{aligned}$$
   In both cases $Pcontenv_V(\rho', \rho)$ implies the desired result.
2. $\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{k}_\sigma]\!] = \mathscr{N}\ [\![\mathbf{k}_\sigma]\!]$ if $\mathbf{k}_\sigma \neq \mathbf{if}$
   The theorem holds trivially in this case.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

3. $\mathscr{T}\ V\ i\ Q_j\ [\![\textbf{if}\ E_1\ E_2\ E_3]\!] = \lambda c.\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]\ (if_c\ (\mathscr{T}\ V\ i\ Q_j\ [\![E_2]\!]\ c)\ (\mathscr{T}\ V\ i\ Q_j\ [\![E_3]\!]\ c))$

$\mathscr{N}\ [\![\textbf{if}\ E_1\ E_2\ E_3]\!] = \lambda c.\mathscr{N}\ [\![E_1]\!](if_c\ (\mathscr{N}\ [\![E_2]\!]\ c)\ (\mathscr{N}\ [\![E_3]\!]\ c))$

$$
\begin{aligned}
&\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![\textbf{if}\ E_1\ E_2\ E_3]\!]]\!]\ \rho' \\
&\quad = \lambda c.\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]]\!]\ \rho'\ (if_c\ \ (\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E_2]\!]]\!]\ \rho'\ c)\ (\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E_3]\!]]\!]\ \rho'\ c)) \\
&\quad \sqsubseteq \lambda c.\mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ (if_c\ (\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho\ c)\ (\mathbf{S}\ [\![\mathscr{N}\ [\![E_3]\!]]\!]\ \rho\ c)) \\
&\qquad \text{by the induction hypothesis} \\
&\quad = \mathbf{S}\ [\![\mathscr{N}\ [\![\textbf{if}\ E_1\ E_2\ E_3]\!]]\!]\ \rho
\end{aligned}
$$

4. $\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!]$

$= \lambda c.\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]\ (\lambda v.\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v)))$

if $\forall \rho,\ \forall v_0 \in P,\ \forall v_1, \ldots, v_i : drop(\ldots(drop(\mathbf{S}\ [\![E_1]\!]\ \rho)\, v_0)\ \ldots)\, v_i \in Q_j$

$\mathscr{N}\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{N}\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{N}\ [\![E_2]\!]))$

There are two cases to consider.

(a) $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' = \perp_{\mathbf{S}_{\mathbf{B}_{[\![\sigma]\!]}}}$:

This implies $\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!]]\!]\ \rho' = \perp_{\mathbf{S}_{\mathbf{B}_{[\![\tau]\!]}}}$, and so the result holds.

(b) $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' \neq \perp_{\mathbf{S}_{\mathbf{B}_{[\![\sigma]\!]}}}$:

By the induction hypothesis we know that $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' \sqsubseteq \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho$, and so $\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho \neq \perp_{\mathbf{S}_{\mathbf{B}_{[\![\sigma]\!]}}}$ too. By Fact B.4 this means that $\exists v_2$ such that $\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho = \lambda c.c\ v_2$. We can now deduce the required result as follows.

$$
\begin{aligned}
&\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!]]\!]\ \rho' \\
&\quad = \lambda c.\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' \\
&\qquad (\lambda v.\mathbf{S}\ [\![\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]]\!]\ \rho'\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
&\quad \sqsubseteq \lambda c.\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho\ (\lambda v.\mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ (\lambda f.f\ c\ (\lambda c.c\ v))) \\
&\qquad \text{by the induction hypothesis} \\
&\quad = \lambda c.\mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ (\lambda f.f\ c\ (\lambda c.c\ v_2)) \\
&\qquad \text{since } \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho = \lambda c.c\ v_2 \\
&\quad = \mathbf{S}\ [\![\mathscr{N}\ [\![E_1\ E_2]\!]]\!]\ \rho \\
&\qquad \text{since } \lambda c.c\ v_2 = \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho
\end{aligned}
$$

5. $\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!]))$

$\mathscr{N}\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{N}\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{N}\ [\![E_2]\!]))$

$$
\begin{aligned}
&\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!]]\!]\ \rho' \\
&\quad = \lambda c.\mathbf{S}\ [\![\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]]\!]\ \rho'\ (\lambda f.f\ c\ (\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!]]\!]\ \rho')) \\
&\quad \sqsubseteq \lambda c.\mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ (\lambda f.f\ c\ (\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho)) \\
&\qquad \text{by the induction hypothesis} \\
&\quad = \mathbf{S}\ [\![\mathscr{N}\ [\![E_1\ E_2]\!]]\!]\ \rho
\end{aligned}
$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

6. $\mathscr{T}\ V\ 0\ Q_j\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_t}\}\ [\![E]\!]\ c)$
$\mathscr{N}\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{N}\ [\![E]\!]\ c)$

$$\begin{aligned}\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ Q_j\ [\![\lambda x.E]\!]]\!]\ \rho' &= \lambda c.c\ (\lambda c.\lambda v.\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_t}\}\ [\![E]\!]]\!]\ \rho'[v/x]\ c)\\ &\sqsubseteq \lambda c.c\ (\lambda c.\lambda v.\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[v/x]\ c)\\ &\quad \text{by the induction hypothesis since}\\ &\quad Pcontenv_V(\rho',\rho) \Longrightarrow Pcontenv_V(\rho'[v/x],\rho[v/x])\\ &= \mathbf{S}\ [\![\mathscr{N}\ [\![\lambda x.E]\!]]\!]\ \rho\end{aligned}$$

7. $\mathscr{T}\ V\ (i+1)\ Q_j\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]\ c)$
$\mathscr{N}\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{N}\ [\![E]\!]\ c)$

$$\begin{aligned}\mathbf{S}\ [\![\mathscr{T}\ V\ (i+1)\ Q_j\ [\![\lambda x.E]\!]]\!]\ \rho' &= \lambda c.c\ (\lambda c.\lambda v.\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]]\!]\ \rho'[v/x]\ c)\\ &\sqsubseteq \lambda c.c\ (\lambda c.\lambda v.\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[v/x]\ c)\\ &\quad \text{by the induction hypothesis since}\\ &\quad Pcontenv_V(\rho',\rho) \Longrightarrow\\ &\quad Pcontenv_V(\rho'[v/x],\rho[v/x])\\ &= \mathbf{S}\ [\![\mathscr{N}\ [\![\lambda x.E]\!]]\!]\ \rho\end{aligned}$$

8. $\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{fix}\ (\lambda x.E)]\!]$
$= \mathbf{sel}_{ij}(\mathbf{fix}\ (\lambda(x_{i_1 j_1},\ldots,x_{i_n j_n}).(\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!],\ \ldots,\ \mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!])))$
where $W = V \bigcup \{x\}$
$\mathscr{N}\ [\![\mathbf{fix}\ (\lambda x.E)]\!] = \mathbf{fix}(\lambda x.\mathscr{N}\ [\![E]\!])$

We will prove this case by fixed point induction. First of all we note that

$$\begin{aligned}&\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{fix}\ (\lambda x.E)]\!]]\!]\ \rho'\\ &\quad = sel_{ij}\ (fix\ (\lambda(u_1,\ldots,u_n).\ (\mathbf{S}\ [\![\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!]]\!]\ \rho'[u_1/x_{i_1 j_1},\ldots,u_n/x_{i_n j_n}],\\ &\qquad\qquad\qquad\qquad \vdots\\ &\qquad\qquad\qquad\qquad \mathbf{S}\ [\![\mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!]]\!]\ \rho'[u_1/x_{i_1 j_1},\ldots,u_n/x_{i_n j_n}])))\\ &\mathbf{S}\ [\![\mathscr{N}\ [\![\mathbf{fix}\ (\lambda x.E)]\!]]\!]\ \rho = fix\ (\lambda u.\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[u/x])\end{aligned}$$

Our method will be to show that each of the terms in the tuple that arises from the $\mathscr{T}$ translation are less defined or equal to that given by the $\mathscr{N}$ translation, which then establishes our result. This leads us to define the predicate:

$$P[(f_1,\ldots,f_n),g] \equiv \forall 1 \leq k \leq n.\ f_k \sqsubseteq g.$$

Clearly $P[(\perp,\ldots,\perp),\perp]$ is true. Assuming $P[(f_1,\ldots,f_n),g]$, we need to establish the property $P[F\ (f_1,\ldots,f_n), G\ g]$, where F and G are the bodies of the fixed

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

points arising from the $\mathscr{T}$ and $\mathscr{N}$ translation, respectively.

$$
\begin{aligned}
F\ (f_1,\ldots,f_n) \;=\; & (\mathbf{S}\ [\![\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!]]\!]\ \rho'[f_1/x_{i_1j_1},\ldots,f_n/x_{i_nj_n}],\\
& \vdots\\
& \mathbf{S}\ [\![\mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!]]\!]\ \rho'[f_1/x_{i_1j_1},\ldots,f_n/x_{i_nj_n}])\\
\sqsubseteq\ & (\mathbf{S}\ [\![\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!]]\!]\ \rho'[g/x_{i_1j_1},\ldots,g/x_{i_nj_n}],\\
& \vdots\\
& \mathbf{S}\ [\![\mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!]]\!]\ \rho'[g/x_{i_1j_1},\ldots,g/x_{i_nj_n}])\\
& \text{by the fixed point induction hypothesis}\\
\sqsubseteq\ & (\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[g/x],\ldots,\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[g/x])\\
& \text{by the structural induction hypothesis since}\\
& Pcontenv_V(\rho',\rho) \Longrightarrow\\
& Pcontenv_V(\rho'[g/x_{i_1j_1},\ldots,g/x_{i_nj_n}],\rho[g/x])
\end{aligned}
$$

which establishes the result for the induction step. Therefore, we can conclude that it is true for the fixed point, which concludes the proof of the theorem.

The following four definitions are used to give an induction hypothesis which is powerful enough to prove Theorem 3.3, the final result we need before being able to prove Theorem 3.3. Since we have to relate the semantics of terms transformed by $\mathscr{T}$, terms transformed by $\mathscr{N}$ and untransformed terms, we first need to establish a relation between their respective environments. This is done by Definition B.9 which uses a correspondence between values introduced in Definition B.8. We use the following convention: overlined values and environments stand for values and environments for original (untransformed) terms and primed values and environments stand for values and environments for terms transformed by $\mathscr{T}$. Unadorned variables are used for terms transformed by $\mathscr{N}$.

Definition B.8

$$Parg(v,\bar{v}) \Leftrightarrow \exists\ closed\ E.[v = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \emptyset] \text{ and } [\bar{v} = \mathbf{S}\ [\![E]\!]\ \emptyset].$$

Definition B.9 $Penv_V(\rho',\rho,\bar{\rho})$ if and only if $\forall\ x$

1. $Parg(\rho\ x,\bar{\rho}\ x)$ *and*
2. $x \in V \Rightarrow\ \forall c,\forall v_1,\ldots,v_i,\forall \bar{v}_1,\ldots,\bar{v}_i$ such that $\forall 1 \le k \le i.\ Parg(v_k,\bar{v}_k)$:
   $[\rho'\ x_{ij}\ c^{[v_1,\ldots,v_i]} = \rho\ x\ c^{[v_1,\ldots,v_i]}\ \ or\ \ \bar{\rho}\ x\ \bar{v}_1\ldots\bar{v}_i \in Q_j]$ *and*
3. $x \in V \Rightarrow\ \rho'\ x_{ij} \sqsubseteq \rho\ x$ *and*
4. $x \notin V \Rightarrow\ \rho'\ x = \rho\ x.$

It is important to note that $Penv_V(\rho',\rho,\bar{\rho})$ implies $Pcontenv_V(\rho',\rho)$, because we need to use Theorem B.7 in proving Theorem B.16.

Definition B.10 $Cont_p(c') \Leftrightarrow \exists c,\ \exists\ closed\ E_1,\ldots,E_p$ *such that* $c' = c^{[v_1,\ldots,v_p]}$ *with* $v_i = \mathbf{S}\ [\![\mathscr{N}\ [\![E_i]\!]]\!]\ \emptyset$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

Definition B.11 $\forall E : \sigma_1 \to \ldots \to \sigma_p \to \sigma_{p+1}$, $Pexp(E)$ *if and only if*

$\forall V, \forall i, \forall Q_j, \forall \rho', \forall \rho, \forall \overline{\rho}$ *such that* $Penv_V(\rho', \rho, \overline{\rho})$
$\forall v_1, \ldots, v_i, \forall \overline{v}_1, \ldots, \overline{v}_i$ such that $\forall 1 \leq k \leq i.\ Parg(v_k, \overline{v}_k)$
$\forall c$ *such that* $Cont_p(c^{[v_1,\ldots,v_i]})$:
$[\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]]\!]\ \rho'\ c^{[v_1,\ldots,v_i]} = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho\ c^{[v_1,\ldots,v_i]}]$ or $[\mathbf{S}\ [\![E]\!]\ \overline{\rho}\ \overline{v}_1 \ldots \overline{v}_i \in Q_j.]$

Fact B.12 For all expressions E, if the free variables of E are $\{x_1, \ldots, x_n\}$, then for all $E_1, \ldots, E_n$,

$$\mathscr{N}\ [\![E[E_k/x_k]_{k=1}^{k=n}]\!] = \mathscr{N}\ [\![E]\!][\mathscr{N}\ [\![E_k]\!]/x_k]_{k=1}^{k=n}.$$

The following three lemmas establish results which are needed to prove Theorem B.16.

Lemma B.13

$$\forall E : \sigma.\ Penv_V(\rho', \rho, \overline{\rho}) \Longrightarrow Parg(\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho, \mathbf{S}\ [\![E]\!]\ \overline{\rho}).$$

Proof This is proved using Fact B.12 (which follows from the rule for variables in the definition of $\mathscr{N}\ [\![E]\!]$).

Lemma B.14

$$\forall E : \sigma.\ Penv_V(\rho', \rho, \overline{\rho}) \Longrightarrow [\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho = \bot_{\mathbf{S}_{B[\![\sigma]\!]}} \Leftrightarrow \mathbf{S}\ [\![E]\!]\ \overline{\rho} = \bot_{\mathbf{S}_\sigma}].$$

Proof From Lemmas B.3 and B.13.

Lemma B.15 $\forall E : \sigma, \forall \rho, \rho', \overline{\rho}$ *such that* $Penv_V(\rho', \rho, \overline{\rho})$
$Pexp(E) \Rightarrow \mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!]\ \rho' = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho.$

Proof Let us assume that the lemma does not hold. Then $Pexp(E)$ gives us that $\mathbf{S}\ [\![E]\!]\ \overline{\rho} \in \{\bot_{\mathbf{S}_\sigma}\}$, and Lemma B.14 implies that $\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho = \bot_{\mathbf{S}_{B[\![\sigma]\!]}}$, and so Theorem B.7 allows us to conclude that $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\bot_{\mathbf{S}_\sigma}\}\ [\![E]\!]]\!]\ \rho' = \bot_{\mathbf{S}_{B[\![\sigma]\!]}}$, which is a contradiction.

Theorem B.16 $\forall E : \sigma.\ Pexp(E)$.

Proof We prove this by induction over the rules defining $\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]$.

1. $\mathscr{T}\ V\ i\ Q_j\ [\![x]\!] = x$ if $x \notin V$
   $= x_{ij}$ if $x \in V$
   $\mathscr{N}\ [\![x]\!] = x$
   (a) If $x \notin V$ then we have to prove

$$[\rho'\ x\ c^{[v_1,\ldots,v_i]} = \rho\ x\ c^{[v_1,\ldots,v_i]}] \text{ or } [\overline{\rho}\ x\ \overline{v}_1 \ldots \overline{v}_i \in Q_j].$$

   The result holds because $Penv_V(\rho', \rho, \overline{\rho})$ implies $\rho'\ x = \rho\ x$.
   (b) If $x \in V$ then we have to prove

$$[\rho'\ x_{ij}\ c^{[v_1,\ldots,v_i]} = \rho\ x\ c^{[v_1,\ldots,v_i]}] \text{ or } [\overline{\rho}\ x\ \overline{v}_1 \ldots \overline{v}_i \in Q_j].$$

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

Again this follows from the definition of $Penv_V(\rho', \rho, \overline{\rho})$.

2. $\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{k}_\sigma]\!] = \mathscr{N}\ [\![\mathbf{k}_\sigma]\!]$ if $\mathbf{k}_\sigma \neq \mathbf{if}$
The theorem holds trivially in this case.

3. $\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] = \lambda c.\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]\ (\mathbf{if_c}\ (\mathscr{T}\ V\ i\ Q_j\ [\![E_2]\!]\ c)\ (\mathscr{T}\ V\ i\ Q_j\ [\![E_3]\!]\ c))$
$\mathscr{N}\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!] = \lambda c.\mathscr{N}\ [\![E_1]\!](\mathbf{if_c}\ (\mathscr{N}\ [\![E_2]\!]\ c)\ (\mathscr{N}\ [\![E_3]\!]\ c))$
From Lemma B.15, we have: $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]]\!]\ \rho' = \mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho$.
If $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_{bool}}\}\ [\![E_1]\!]]\!]\ \rho'$ and $\mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho$ are both $\perp_{\mathbf{S}_{\mathbf{B}_{[\![bool]\!]}}}$, then the result clearly holds.
If they are not, then Fact B.4 gives us that there is a v such that they both equal $\lambda c.c\ v$. Again the result clearly holds in the case that v is $\perp_{\mathbf{S}_{bool}}$.
If $v \neq \perp_{\mathbf{S}_{bool}}$, then it is equal to *true* or *false*. Without loss of generality, we will consider the case where $v = true$. Then

$$\begin{aligned} \mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!]]\!]\ \rho' &= \mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E_2]\!]]\!]\ \rho' \\ \mathbf{S}\ [\![\mathscr{N}\ [\![\mathbf{if}\ E_1\ E_2\ E_3]\!]]\!]\ \rho &= \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho \end{aligned}$$

The result then holds immediately by the induction hypothesis.

4. $\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!]$
$= \lambda c.\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]\ (\lambda v.\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]\ (\lambda f.f\ c\ (\lambda c.c\ v)))$
if $\forall \rho,\ \forall v_0 \in P,\ \forall v_1,\ \ldots,\ v_i : drop(\ldots(drop(\mathbf{S}\ [\![E_1]\!]\ \rho)\, v_0)\ \ldots)\, v_i \in Q_j$
$\mathscr{N}\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{N}\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{N}\ [\![E_2]\!]))$
We have to prove that

$$\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho'\ (\lambda v.\mathbf{S}\ [\![\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]]\!]\ \rho' c^{[(\lambda c.c\ v), v_1, \ldots, v_i]}) = \mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ c^{[\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho, v_1, \ldots, v_i]}$$

or

$$\mathbf{S}\ [\![E_1]\!]\ \overline{\rho}\ (\mathbf{S}\ [\![E_2]\!]\ \overline{\rho})\ \overline{v}_1 \ldots \overline{v}_i \in Q_j.$$

There are three cases to consider.

(a) $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' \neq \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho$: By the induction hypothesis we have that $\mathbf{S}\ [\![E_2]\!]\ \overline{\rho} \in P$, and so the condition of using this translation rule gives us that

$$\mathbf{S}\ [\![E_1]\!]\ \overline{\rho}\ (\mathbf{S}\ [\![E_2]\!]\ \overline{\rho})\ \overline{v}_1 \ldots \overline{v}_i \in Q_j,$$

which is what was required.

(b) $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' = \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho = \perp_{\mathbf{S}_{\mathbf{B}_{[\![\sigma]\!]}}}$: From Lemma B.14 we have that $\mathbf{S}\ [\![E_2]\!]\ \overline{\rho} = \perp_{\mathbf{S}_\sigma}$, and, since $\perp_{\mathbf{S}_\sigma} \in P$ (because P is a non-empty Scott-closed set), the condition on using this translation rule implies that $\forall \overline{v}_1, \ldots, \overline{v}_j.\ \mathbf{S}\ [\![E_1]\!]\ \overline{\rho}\ \perp_{\mathbf{S}_\sigma}\ \overline{v}_1 \ldots \overline{v}_j \in Q_j$, and so we conclude that

$$\mathbf{S}\ [\![E_1\ E_2]\!]\ \overline{\rho}\ \ \overline{v}_1 \ldots \overline{v}_i \in Q_j,$$

which is what was required.

(c) $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ P\ [\![E_2]\!]]\!]\ \rho' = \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho = \lambda c.c\ v$: This case arises from Fact B.4. The result we had to prove for the case for this translation rule

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

then reduces to:

$$\mathbf{S}\ [\![\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]]\!]\ \rho' c^{[(\lambda c.c\ v),v_1,\ldots,v_i]} = \mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ c^{[\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho,v_1,\ldots,v_i]}$$

or

$$\mathbf{S}\ [\![E_1]\!]\ \overline{\rho}\ (\mathbf{S}\ [\![E_2]\!]\ \overline{\rho})\ \overline{v}_1 \ldots \overline{v}_i \in Q_j.$$

This follows from the induction hypothesis because $\lambda c.c\ v = \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho$ and, by Lemma B.13, $Parg(\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho, \mathbf{S}\ [\![E_2]\!]\ \overline{\rho})$.

5. $\mathscr{T}\ V\ i\ Q_j\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!]))$
$\mathscr{N}\ [\![E_1\ E_2]\!] = \lambda c.\mathscr{N}\ [\![E_1]\!]\ (\lambda f.f\ c\ (\mathscr{N}\ [\![E_2]\!]))$

In this case we have to prove that

$$\mathbf{S}\ [\![\mathscr{T}\ V\ (i+1)\ Q_j\ [\![E_1]\!]]\!]\ \rho'\ c^{[\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!]]\!]\ \rho',v_1,\ldots,v_i]} = \mathbf{S}\ [\![\mathscr{N}\ [\![E_1]\!]]\!]\ \rho\ c^{[\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho,v_1,\ldots,v_i]}]$$

or

$$[\mathbf{S}\ [\![E_1]\!]\ \overline{\rho}\ (\mathbf{S}\ [\![E_2]\!]\ \overline{\rho})\ \overline{v}_1 \ldots \overline{v}_i \in Q_j].$$

From Lemma B.15, we have: $\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\sigma}\}\ [\![E_2]\!]]\!]\ \rho' = \mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho$. Furthermore, Lemma B.13 gives us that $Parg(\mathbf{S}\ [\![\mathscr{N}\ [\![E_2]\!]]\!]\ \rho, \mathbf{S}\ [\![E_2]\!]\ \overline{\rho})$, and so the result holds for this case from the induction hypothesis on E_1.

6. $\mathscr{T}\ V\ 0\ Q_j\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\tau}\}\ [\![E]\!]\ c)$
$\mathscr{N}\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{N}\ [\![E]\!]\ c)$

To prove $Pexp(\lambda x.E)$ we show that:
$\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ Q_j\ [\![\lambda x.E]\!]]\!]\ \rho'\ c = \mathbf{S}\ [\![\mathscr{N}\ [\![\lambda x.E]\!]]\!]\ \rho\ c$,
where c satisfies $Cont_p(c)$ with $p \geq 1$ (since $\lambda x.E$ has a function type). So $\exists c_0, E_0$ such that $c = c_0^{[\mathbf{S}\ [\![\mathscr{N}\ [\![E_0]\!]]\!]\ \emptyset]}$ and the property we have to prove becomes:

$$\mathbf{S}\ [\![\mathscr{T}\ V\ 0\ \{\perp_{\mathbf{S}_\tau}\}\ [\![E]\!]]\!]\ \rho'[(\mathbf{S}\ [\![\mathscr{N}\ [\![E_0]\!]]\!]\ \emptyset)/x] = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[(\mathbf{S}\ [\![\mathscr{N}\ [\![E_0]\!]]\!]\ \emptyset)/x],$$

which follows from Lemma B.15.

7. $\mathscr{T}\ V\ (i+1)\ Q_j\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]\ c)$
$\mathscr{N}\ [\![\lambda x.E]\!] = \lambda c.c\ (\lambda c.\lambda x.\mathscr{N}\ [\![E]\!]\ c)$

We have to prove

$$c^{[v_1,\ldots,v_{i+1}]}\ (\lambda c.\lambda v.\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]]\!]\ \rho'[v/x]\ c) = c^{[v_1,\ldots,v_{i+1}]}\ (\lambda c.\lambda v.\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[v/x]\ c)$$

or

$$\mathbf{S}\ [\![\lambda x.E]\!]\ \overline{\rho}\ \overline{v}_1 \ldots \overline{v}_{i+1} \in Q_j.$$

Using the definition of $c^{[v_1,\ldots,v_{i+1}]}$, this is equivalent to showing

$$\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![E]\!]]\!]\ \rho'[v_1/x]\ c^{[v_2,\ldots,v_{i+1}]} = \mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[v_1/x]\ c^{[v_2,\ldots,v_{i+1}]}$$

or

$$\mathbf{S}\ [\![E]\!]\ \overline{\rho}[\overline{v}_1/x]\ \overline{v}_2 \ldots \overline{v}_{i+1} \in Q_j.$$

Now $Penv_V(\rho'[v_1/x], \rho[v_1/x], \overline{\rho}[\overline{v}_1/x])$ since $Penv_V(\rho', \rho, \overline{\rho})$ and $Parg(v_1, \overline{v}_1)$. Therefore, the result holds by the induction hypothesis on E.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

8. $\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{fix}\ (\lambda x.E)]\!]$
$= \mathbf{sel}_{ij}(\mathbf{fix}\ (\lambda(x_{i_1 j_1},\ldots,x_{i_n j_n}).(\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!],\ \ldots,\ \mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!])))$
where $W = V \bigcup \{x\}$
$\mathscr{N}\ [\![\mathbf{fix}\ (\lambda x.E)]\!] = \mathbf{fix}(\lambda \mathbf{x}.\mathscr{N}\ [\![E]\!])$
We will prove this case by fixed point induction. First, we note that

$$\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{fix}\ (\lambda x.E)]\!]]\!]\ \rho' c^{[v_1,\ldots,v_i]}$$
$$= sel_{ij}\ (fix\ (\lambda(u_1,\ldots,u_n).\ (\mathbf{S}\ [\![\mathscr{T}\ W\ i_1\ Q_{j_1}\ [\![E]\!]]\!]\ \rho'[u_1/x_{i_1 j_1},\ldots,u_n/x_{i_n j_n}],$$
$$\vdots$$
$$\mathbf{S}\ [\![\mathscr{T}\ W\ i_n\ Q_{j_n}\ [\![E]\!]]\!]\ \rho'[u_1/x_{i_1 j_1},\ldots,u_n/x_{i_n j_n}])))\ c^{[v_1,\ldots,v_i]}$$
$$\mathbf{S}\ [\![\mathscr{N}\ [\![\mathbf{fix}\ (\lambda x.E)]\!]]\!]\ \rho\ c^{[v_1,\ldots,v_i]} = fix\ (\lambda u.\mathbf{S}\ [\![\mathscr{N}\ [\![E]\!]]\!]\ \rho[u/x])c^{[v_1,\ldots,v_i]}$$
$$\mathbf{S}\ [\![\mathbf{fix}\ (\lambda x.E)]\!]\ \overline{\rho}\ \overline{v_1}\ldots\ \overline{v_i} = fix(\lambda u.\mathbf{S}\ [\![E]\!]\ \overline{\rho}[u/x])\ \overline{v_1}\ldots\ \overline{v_i}$$

To prove $Pexp(\mathbf{fix}\ (\lambda x.E))$ we define property P_3 as follows:

$$P_3(A_{i,j},B,C) \Leftrightarrow (A_{i,j}\ c^{[v_1,\ldots,v_i]} = B\ c^{[v_1,\ldots,v_i]})\ \text{or}\ C\ \overline{v_1}\ldots\overline{v_i} \in Q_j$$

$$and\quad Parg(B,C)\quad and\quad A_{i,j} \sqsubseteq B$$

The second line of the property is required by the induction itself. The base case $P_3(\bot,\bot,\bot)$ is obvious. For the inductive case, we assume $P_3(A_{i,j},B,C)$ and we show:

$$P_3(F(A_{i,j}),G(B),H(C))$$

where F, G and H are the bodies of the fixed point transformers arising from

$$\mathbf{S}\ [\![\mathscr{T}\ V\ i\ Q_j\ [\![\mathbf{fix}\ (\lambda x.E)]\!]]\!]\ \rho',$$

$$\mathbf{S}\ [\![\mathscr{N}\ [\![\mathbf{fix}\ (\lambda x.E)]\!]]\!]\ \rho$$

and

$$\mathbf{S}\ [\![\mathbf{fix}\ (\lambda x.E)]\!]\ \overline{\rho}$$

respectively. The first part of the property is established using the fixed point induction hypothesis and the structural induction hypothesis (on E). The fixed point induction hypothesis can be applied because the second part of property P_3 ensures that the corresponding environments satisfy $Penv_V$. The second part of the inductive case can easily be checked:

$$Parg(B,C) \Rightarrow Parg(G(B),H(C))$$

and

$$A_{i,j} \sqsubseteq B \Rightarrow F(A_{i,j}) \sqsubseteq G(B)$$

The proof of Theorem 3.3 in the main body of the paper now follows from Lemma B.15 and Theorem B.16. We restate it here for completeness.

https://doi.org/10.1017/S0956796800001581 Published online by Cambridge University Press

Theorem 3.3 *For all closed terms of ground type* $E : \sigma$,

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

References

Appel, A. W. (1992) *Compiling with Continuations.* Cambridge University Press.

Benton, P. N. (1992) Strictness Logic and Polymorphic Invariance. In: Nerode, A. and Taitslin, M., editors, *Proc. Int. Symposium on Logical Foundations of Computer Science*, pp. 33–44. Tver, Russia. Springer-Verlag.

Burn, G. L. (1990) Using projection analysis in compiling lazy functional programs. In: *Proc. ACM Conference on Lisp and Functional Programming*, pp. 227–241. ACM Press.

Burn, G. L. (1991a) The Evaluation Transformer Model of reduction and its correctness. In: Abramsky, S. and Maibaum, T. S. E., editors, *Proc, TAPSOFT'91, Vol 2*, pp. 458–482, Brighton, UK. Springer-Verlag.

Burn, G. L. (1991b) *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman (in association with MIT Press).

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

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

Cousot, P. and Cousot, R. (1979) Systematic Design of program analysis frameworks. In: *Proc. 6th Ann. Symposium on Principles of Programming Languages*, pp. 269–282. ACM Press.

Cousot, P. and Cousot, R. (1992a) Abstract interpretation and application to logic programs. *J. Logic Programming*, **13**(2–3): 103–179.

Cousot, P. and Cousot, R. (1992b) Abstract interpretation frameworks. *J. Logic and Computation*, **2**(4).

Danvy, O. and Filinski, A. (1991) Representing Control: a Study of the Cps Transformation. *Technical Report TR CIS-91-2*, Kansas State University.

Danvy, O. and Hatcliff, J. (1993) CPS Transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, **1**(3): 195–212.

Danvy, O. and Hatcliff, J. (1994) A generic account of continuation-passing styles. In: *Proc. 21st ACM Sigplan-Sigact Symposium on Principles Of Programming Languages.* ACM Press.

Dybjer, P. (1985) Using domain algebras to prove the correctness of a compiler. In: *Proc. STACS85*, pp. 98–108. Springer-Verlag.

Finne, S. O. and Burn, G. L. (1993) Assessing the Evaluation Transformer Model of reduction on the spineless G-machine. In: *Proc. Conf. Functional Programming and Computer Architecture*, pp. 331–340. ACM Press.

Fischer, M. J. (1972) Lambda calculus schemata. In: *ACM Conf. on Proving Assertions about Programs*, pp. 104–109, New Mexico. (*ACM Sigplan Notices* **7**(1).)

Fischer, M. J. (1993) Lambda calculus schemata. *Lisp and Symbolic Computation*, **6**(3/4): 259–287.

Flanagan, C., Sabry, A., Duba, B. F. and Felleisen, M. (1993) The essence of compiling with continuations. In: *ACM Sigplan '93 Conf. on Programming Language Design and Implementation*, pp. 237–247, Albuquerque, NM. (*ACM Sigplan Notices*, **28**(6).)

Fradet, P. (1988) *Compilation des langages fonctionnels par transformation de programmes.* PhD thesis, Université de Rennes I.

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

Gerhart, S. L. (1975) Correctness-preserving program transformations. In: *Proc. POPL75*, pp. 54–66. ACM Press.

Giorgi, J. F. and Le Métayer, D. (1990) Continuation-based parallel implementation of functional programming languages. In: *Proc. ACM Conf. on Lisp and Functional Programming*, pp. 227–241.

Gunter, C. A. (1992) *Semantics of Programming Languages: Structures and Techniques.* MIT Press.

Hunt, L. S. (1991) *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College, University of London.

Jensen, T. P. (1992a) *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, University of London.

Jensen, T. P. (1992b) Disjunctive strictness analysis. In: *Proc. 7th Symposium on Logic In Computer Science*, pp. 174–185, Santa Cruz, CA. IEEE Press.

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

Kranz, D. A., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J. and Adams, N. I. (1986) Orbit: an optimising compiler for Scheme. In: *Proc. SIGPLAN '86 Symposium on Compiler Construction*, pp. 219–233. ACM Press.

Kuo, T.-M. and Mishra, P. (1989) Strictness analysis: a new perspective based on type inference. In: *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pp. 260–272, London, UK. ACM Press.

Lester, D. R. (1987) The G-machine as a representation of stack semantics. In: Kahn, G., editor, *Proc. Functional Programming Languages and Computer Architecture Conf.*, pp. 46–59. Springer-Verlag.

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

Leung, A. and Mishra, P. (1991) Reasoning about simple and exhaustive demand in higher-order languages. In: Hughes, J., editor, *Proc. Conference on Functional Programming and Computer Architecture*, pp. 329–351, Cambridge, MA. Springer-Verlag.

Meyer, A. and Wand, M. (1985) Continuation semantics in the typed lambda-calculus. In: *Proc. Logics of Programs*, pp. 219–224. Springer-Verlag.

Morris, F. L. (1973) Advice on structuring compilers and proving them correct. In: *Proc. POPL73*, pp. 144–152 ACM Press.

Mosses, P. D. (1980) A constructive approach to compiler correctness. In: *Proc. ICALP80*, pp. 449–462. Springer-Verlag.

Muylaert Filho, J. and Burn, G. L. (1993) Continuation passing transformation and abstract intrepretation. In: Burn, G. L., Gay, S. J. Ryan, M. D., editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods.* Workshops in Computer Science. Sussex, UK. Springer-Verlag.

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

Nielson, F. (1985) Program transformations in a denotational setting. *ACM TOPLAS*, **7**: 359–379.

Nielson, F. (1988) Strictness analysis and denotational abstract interpretation. *Inform. and Comput.*, **76**: 29–92.

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

Nielson, F. and Nielson, H. R. (1992) The tensor product in Wadler's Analysis of Lists. In: Krieg-Brückner, B., editor, *Proceedings of ESOP'92*, pp. 351–370, Rennes, France. Springer-Verlag.

Nielson, H. R. and Nielson, F. (1988) Two-level Semantics and Code Generation. *Theoretical Computer Science*, **56**: 59–133.

Peyton Jones, S. L. and Launchbury, J. (1991) Unboxed values as first class citizens in a non-strict functional language. In: Hughes, J., editor, *Proc. Conference on Functional Programming and Computer Architecture*, pp. 636–666, Cambridge, MA. Springer-Verlag.

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

Reynolds, J. C. (1974) On the relation between direct and continuation semantics. In: *Proc. 2nd Colloquium on Automata, Languages and Programming*, pp. 141–156. Springer-Verlag.

Sabry, A. and Felleisen, M. (1992) Reasoning about Programs in Continuation-Passing Style. *Technical Report TR 92-180*, Rice University.

Sabry, A. and Felleisen, M. (1993) Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, **6**(3/4).

Schmidt, D. A. (1980) State transition machines for lambda-calculus expressions. In: *Proc. Semantics-Directed Compiler Generation Workshop*, pp. 415–440. Springer-Verlag.

Steele Jr., G. L. (1978) Rabbit: A Compiler for Scheme. *Technical report*, AI Tech. Rep. 474. MIT, Cambridge, MA.

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

Wadler, P. and Hughes, R. J. M. (1987) Projections for strictness analysis. In: Kahn, G., editor, *Proc. Functional Programming Languages and Computer Architecture Conf.*, pp. 385–407. Springer-Verlag.

Wadler, P. L. (1987) Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In: Abramsky, S. and Hankin, C. L., editors, *Abstract Interpretation of Declarative Languages*, pp. 266-275. Ellis Horwood.

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