# A Unified Approach to Strictness Analysis and Optimising Transformations

P. N. Benton
University of Cambridge *
Nick.Benton@cl.cam.ac.uk

**Abstract**

We present an inference system for translating programs in a PCF-like source language into a variant of Moggi's computational lambda calculus. This translation combines a simple strictness analysis with its associated optimising transformations into a single system. The correctness of the translation is established using a logical relation between the denotational semantics of the source and target languages.

## 1 Introduction

### 1.1 Background

Strictness analysis of lazy functional programs has been studied extensively during the the last 15 years or so, usually with the justification that the results of the analysis can be used in an optimising compiler [Myc81, BHA86, Ben92]. There has, however, been surprisingly little serious work on just *how* the results of strictness analysis can be used as the basis for optimising transformations. This is probably because it turns out to be rather more difficult to express and justify these optimisations than one might at first imagine. Roughly speaking, it seems we have to decide

1. What optimisations we wish to perform.
2. How to express these optimisations in some formal framework.
3. Exactly what information has to be gathered to enable each optimisation.
4. How to prove the correctness of the optimisations.

The next few paragraphs attempt to sketch the range of possible answers to each of these questions and to indicate which choices were made in some of the previous work on the subject.

*University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK. Research supported by the EU BRA 8130 LOMAPS.

### 1.1.1 Which optimisations?

The most basic optimisation we could want to perform is to evaluate some function arguments to weak head normal form before the function call [Bur91, NN90, DH93, Amt93]. If our language includes datatypes such as pairs or lists, then we might wish to evaluate some arguments beyond WHNF (evaluating all the spine cells of a list, for example) [Bur91, NN90]. These optimisations then naturally suggest a further class of complementary optimsations in which, for example, functions are compiled to *expect* arguments which have already been evaluated to a certain degree. This can then be extended to higher order – a function can be compiled to expect as its argument a strict function, which in turn will expect its argument to be already evaluated. Evaluated arguments to monomorphic functions can be passed unboxed, which improves both space and time efficiency. In call-by-need implementations, knowing that a certain value is already evaluated can save unnecessary graph update operations as well as evaluations.

In addition to all these levels of optimisation, we also have to decide whether or not to compile multiple versions of functions for use in different contexts (we might, for example, compile one version of `map` which expects a strict function as its first argument and one version which doesn't). And if we decide to opt for multiple versions, we then have to decide whether or not the choice of versions should be static (determined at compile-time), dynamic (determined at run-time) or some mixture of the two. Multiple code versions also raise the more pragmatic questions of how to control the exponential blowup in code size which can result and how to deal with the potential loss of sharing (which can actually lead to slower code by duplicating evaluations unnecessarily).

On parallel hardware, strictness analysis can also be used to decide which expressions should be evaluated in parallel. In this paper we will only explicitly consider optimisations for sequential implementations, though much of the discussion is applicable to parallel ones too. In passing, however, we remark that in a parallel system, compiling functions to expect evaluated arguments does not seem to be particularly useful. This is because a function body cannot be compiled simply to assume that an argument is fully evaluated, since the thread which is evaluating that argument may not have finished (or even started) when the body attempts to use the argument. Hence the body needs to make a run-time examination of the argument, which is just what we were trying to avoid, and be prepared either to block or possibly simply to do the evaluation itself.

### 1.1.2 Formalising the optimisations

There are two main approaches to expressing and reasoning about strictness-based optimisations, which reflect different schools of thought about the foundations of functional languages. What we might call the 'lambda calculus' approach [Bur91] starts from the idea that leftmost reduction is but one of many reduction strategies for the lambda calculus, and that what we wish to do is work out when it is safe to use some alternative strategy. The Church-Rosser theorem is central to this approach, as it is this which ensures (roughly) that any choice of strategy which preserves termination is safe. The 'programming language' approach [NN90, Amt93] eschews all mention of reduction strategies (and, indeed, classical results about the lambda calculus) and instead looks at different translations of the source programming language into some other target language with a fixed operational semantics. The 'programming language' approach has several advantages over the 'lambda calculus' approach. Firstly, the very notion of reduction strategy is unnecessary, unrealistic and messy – not only are strategies complicated things to reason about,

but as they will ultimately be realised by different translations of the source language into machine code, we might as well just work with translations directly. Secondly, the 'lambda calculus' approach does not deal well with the optimisations which involve knowing that certain things will already have been evaluated. Thirdly, the 'programming language' view is the only one which makes sense for other kinds of programming languages and optimisations. Hence we should be able to draw on, and contribute to, work on other compile-time transformations which do not seem to have any simple link with traditional results about the lambda calculus, such as the optimisation of data representations.

Of course, if we decide to use strictness information to change the translation of our source language into some intermediate language, then we have to decide what that target language should be. Nielson and Nielson [NN90] use strictness information to change the translation of a combinatory source language into a lazy variant of the categorical abstract machine. This has the advantage of being very close to implementation practice, but it is probably slightly too low-level for correctness proofs to be comfortable and the structure of optimisations can be rather hard to see amid the details of the compiled code. Danvy and Hatcliff [DH93] use strictness information to control the translation of a call-by-name source language into continuation passing style. Amtoft [Amt93] chooses to translate his call-by-name source language into a call-by-value target language, using derivations in a strictness type system to improve on the well-known naive translation. In this paper, we shall take the target language to be a variant of Moggi's computational lambda calculus.

### 1.1.3 Gathering the information

Deciding what information has to be gathered to enable optimisations is also rather tricky. Whatever analysis technique one uses (e.g. abstract interpretation or type inference), there is a choice to be made between performing a 'sticky' analysis, which analyses the entire program first and produces some kind of annotated program as the input to a subsequent transformation phase, or a 'non-sticky' analysis in which the transformation or code-generation phase calls the analyser on the fly to establish particular properties of program fragments in order to justify particular optimisations. As the Nielsons observe, the correctness of the sticky analysis is hard to establish, as "the semantic content of such annotations is somewhat subtle" [NN90], essentially because the strictness analyser gives the strictness properties of an expression as a function of the strictness properties of its free variables. It therefore seems, at least at first sight, necessary to combine the strictness analysis with some kind of 'collecting interpretation' or 'flow analysis' which computes (an approximation to) the set of (strictness properties of) terms which could become bound to those variables during execution [HY91]. For this reason [NN90] uses a non-sticky analysis. Burn [Bur91] does use a sticky analysis to annotate applications with evaluation transformer information derived by abstract interpretation, but he appears to propose the use of rather weak ('context free') annotations for higher-order functions. Amtoft's system is also sticky – the role of the annotated program is played by a derivation in his strictness type system, though a single term can have many valid strictness derivations and hence many annotations. Danvy and Hatcliff assume that a sticky analysis has already supplied them with an annotated program, and do not discuss how the information is gathered.

A further complication of sticky analyses is that some thought must be given to maintaining the correctness of the annotations as the program is transformed or compiled. A related disadvantage of non-sticky analyses is that if they are implemented naively then they may require the properties of expressions to be repeatedly recomputed – to compile a

compound expression, some property is computed which involves computing properties of subexpressions. One then recursively compiles the subexpressions, which involves computing their properties all over again. Whilst it seems simple to fix this by returning strictness properties and a code stream together in a bottom-up fashion, it should be noted that we probably do not wish to compute *all* the strictness properties of the subexpressions before compiling a compound expression. Furthermore, we wish the subexpressions to be compiled in different ways according to the properties deduced of the larger expression.[1]

### 1.1.4 Proving correctness

Burn approaches the correctness of his optimisations using a mixture of techniques, appealing to denotational semantics and computational adequacy for the correctness of the abstract interpretation and to the Church-Rosser and head-normalisation theorems, together with a certain amount of informal English argument for the correctness of the idea of evaluation transformers. The final stage, compiling different reduction strategies into different code sequences for the Spineless G-Machine, is not justified.

Danvy and Hatcliff show that their CPS transformation of annotated programs is correct by deriving it from the composition of a translation of annotated programs to a call-by-value language with `delay` and `force` constructs and a CPS translation of this extended call-by-value language. The correctness of each of these component translations is established from a denotational semantics.

Amtoft proves the correctness of his translation by establishing directly from the operational semantics that the call-by-value evaluation of the translation of a program terminates with a value iff the call-by-name evaluation of the original program terminates with that value. A particularly pleasant feature of this proof is that the analysis is not first proved correct in isolation – the correctness property of the analysis is simply that the associated transformations are correct (cf. [Wan93]). Nielson and Nielson do not address the question of correctness at all, though their paper does consider more sophisticated optimisations than the other works cited. In this paper we shall establish correctness by purely denotational techniques.

## 1.2 This paper

This paper takes a similar approach to that of Amtoft. We essentially use a strictness type system to improve the translation of a simply typed lambda calculus with constants, $\Lambda_T$, into a variant of the computational lambda calculus [Mog89, Mog91], called $\Lambda_{\mathrm{op}}$. The target language $\Lambda_{\mathrm{op}}$, which was first proposed as a language for expressing strictness optimisations in [Ben92], has a type system which makes an explicit distinction between *computations*, which are expressions which are potentially unevaluated, and *values*, which are expressions in WHNF. This appears to be just the level of extra refinement which we need to express both the eager evaluation of function arguments and the complementary optimisations which are based on knowing that certain expressions will already have been evaluated. $\Lambda_{\mathrm{op}}$ is in many respects similar to languages with explicit boxed and unboxed types presented by Peyton Jones and Launchbury in [PJL91] and by Leroy in [Ler92], and indeed many of the same issues arise in the optimisation of data representations (passing

[1] Of course, this is just the sort of situation in which one might hope that writing the compiler itself in a lazy language might alleviate the problem, but one can hardly expect to get exactly the desired behaviour for free.

arguments boxed or unboxed) as in strictness-based optimisations (passing arguments unevaluated or evaluated).

A major difference between the translation presented here and previous work is that although there is morally a strictness type system underlying the translation, it is completely integrated with a transformation phase. Thus the only 'strictness properties' which are ever visible are in the types of the optimised translations of $\Lambda_T$ terms and we remove the distinction between what optimisations we wish to perform and what information has to be gathered. We manage to obtain much of the benefit of using a collecting interpretation just from the way in which types are used in the translation; we can, for example, often discover that a higher-order function is only ever called with a strict function as argument and compile it accordingly.[2]

The translation is nondeterministic, in that it specifies a *set* of valid translations of a single source language program. We do not examine it detail the problem of how to define and find the 'best' translation, though seems likely that some relatively straighforward heuristics should give fast analysis and good results. The version of the translation presented here does not generate multiple code versions and only treats ground types and function spaces. The analysis inherent in the translation is not a particularly powerful one and we discuss some possible improvements in Section 6.

## 2 The source language $\Lambda_T$

The source language $\Lambda_T$ is a conventional simply-typed lambda calculus with constructs for arithmetic, conditionals and recursion, i.e. an inessential variant of Plotkin's language PCF [Plo77]. The syntax and typing rules of $\Lambda_T$ are shown in Figure 1. The call-by-name

---

$$\begin{array}{lllll} \text{types} & A, B & ::= & \mathtt{nat} \mid A \to B \\ \text{contexts} & \Gamma, \Delta & ::= & a_1 \colon A_1, \ldots, a_n \colon A_n \\ \text{arithmetic} & op & ::= & + \mid - \mid * \end{array}$$

$$\text{Id}\ \frac{}{\Gamma, a \colon A \vdash a \colon A} \qquad \text{Abs}\ \frac{\Gamma, a \colon A \vdash e \colon B}{\Gamma \vdash \lambda a \colon A.e \colon A \to B}$$

$$\text{App}\ \frac{\Gamma \vdash e \colon A \to B \qquad \Gamma \vdash f \colon A}{\Gamma \vdash e\ f \colon B} \qquad \text{Rec}\ \frac{\Gamma, a \colon A \vdash e \colon A}{\Gamma \vdash \mathtt{rec}(a \colon A.\ e) \colon A}$$

$$\text{Nat}\ \frac{}{\Gamma \vdash \underline{n} \colon \mathtt{nat}} \qquad \text{Arith}\ \frac{\Gamma \vdash e \colon \mathtt{nat} \qquad \Gamma \vdash f \colon \mathtt{nat}}{\Gamma \vdash e\ \underline{op}\ f \colon \mathtt{nat}}$$

$$\text{Cond}\ \frac{\Gamma \vdash e \colon \mathtt{nat} \qquad \Gamma \vdash f_1 \colon A \qquad \Gamma \vdash f_2 \colon A}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ f_1\ \mathtt{else}\ f_2 \colon A}$$

---

Figure 1: Syntax and type rules of $\Lambda_T$

[2]It should be intuitively clear that types are ideally suited to obtaining the kind of information gathered by a collecting interpretation. For example, the fact that a variable has a particular type is a restriction on the set of terms which may end up bound to that variable during execution.

operational semantics of $\Lambda_T$ is given by a big-step evaluation relation $\Downarrow$, which relates closed terms of type $A$ to weak head normal forms (canonicals) of type $A$. The definition of $\Downarrow$ is shown in Figure 2.

$$\frac{}{\lambda a\colon A.e \Downarrow \lambda a\colon A.e} \qquad \frac{}{\underline{n} \Downarrow \underline{n}}$$

$$\frac{e \Downarrow \lambda a\colon A.e' \qquad e'[f/a] \Downarrow k}{e\ f \Downarrow k}$$

$$\frac{e[\mathtt{rec}(a\colon A.\ e)/a] \Downarrow k}{\mathtt{rec}(a\colon A.\ e) \Downarrow k} \qquad \frac{e \Downarrow \underline{m} \qquad f \Downarrow \underline{n}}{e\ \underline{op}\ f \Downarrow \underline{m\ op\ n}}$$

$$\frac{e \Downarrow \underline{0} \qquad f_1 \Downarrow k}{\mathtt{if}\ e\ \mathtt{then}\ f_1\ \mathtt{else}\ f_2 \Downarrow k} \qquad \frac{e \Downarrow \underline{n+1} \qquad f_2 \Downarrow k}{\mathtt{if}\ e\ \mathtt{then}\ f_1\ \mathtt{else}\ f_2 \Downarrow k}$$

Figure 2: Operational semantics of $\Lambda_T$

$\Lambda_T$ also can also be given the usual denotational semantics using pointed $\omega$-cpos (domains) and continuous maps, as shown in Figure 3. We assume the usual notational conventions concerning environments, $\rho$, and standard results concerning the well-definedness of the denotational semantics, substitution and so on. One piece of notation which may not be familiar is that if $\Gamma = a_1\colon A_1, \ldots, a_n\colon A_n$ and $\rho$ is an environment, then we write $\rho\colon\Gamma$ to mean that the domain of $\rho$ is $\{a_1, \ldots, a_n\}$ and that for all $i$, $\rho(a_i) \in [\![A_i]\!]$.

Types

$$\begin{array}{rcl} [\![\mathtt{nat}]\!] & = & IN_\perp \\ [\![A \to B]\!] & = & [\![A]\!] \to [\![B]\!] \end{array}$$

Terms

$$\begin{array}{rcl}
[\![a]\!]\rho & = & \rho(a) \\
[\![\lambda a\colon A.e]\!]\rho & = & \lambda d \in [\![A]\!].[\![e]\!]\rho[a \mapsto d] \\
[\![e\ f]\!]\rho & = & ([\![e]\!]\rho)\ ([\![f]\!]\rho) \\
[\![\underline{n}]\!]\rho & = & [n] \\
[\![\mathtt{rec}(a\colon A.\ e)]\!]\rho & = & \bigsqcup_{i\in\omega} d_i \quad \text{where } d_0 = \perp_{[\![A]\!]},\ d_{n+1} = [\![e]\!]\rho[a \mapsto d_n] \\
[\![e\ \underline{op}\ f]\!]\rho & = & \begin{cases} [m\ op\ n] & \text{if } [\![e]\!]\rho = [m] \text{ and } [\![f]\!]\rho = [n] \\ \perp_{IN_\perp} & \text{otherwise} \end{cases} \\
[\![\mathtt{if}\ e\ \mathtt{then}\ f_1\ \mathtt{else}\ f_2]\!]\rho & = & \begin{cases} [\![f_1]\!]\rho & \text{if } [\![e]\!]\rho = [0] \\ [\![f_2]\!]\rho & \text{if } [\![e]\!]\rho = [n+1] \\ \perp & \text{otherwise} \end{cases}
\end{array}$$

Figure 3: Denotational semantics of $\Lambda_T$

**Proposition 1 (Computational adequacy for $\Lambda_T$)** *For any program (closed term of type* `nat`*), $e$ in $\Lambda_T$*

$$[\![e]\!] = [n] \;\Leftrightarrow\; e \Downarrow \underline{n}$$

**Proof.** Standard. See, for example, [Plo77, Ben92, Win93] □

# 3 The target language $\Lambda_{\mathrm{op}}$

$\Lambda_{\mathrm{op}}$ is based on Moggi's computational lambda calculus, but has a more syntactic, operational flavour. It is intended as a compiler intermediate language which has just enough extra structure to express the kinds of optimisations which we wish to perform as a result of strictness analysis, but which is sufficiently high-level not to be tied to a particular implementation technique and for denotational reasoning to be straightforward. The type system of $\Lambda_{\mathrm{op}}$ separates computations from values in a rather literal way – values are expressions in weak head normal form, whereas computations are unevaluated expressions. The syntax and typing rules of $\Lambda_{\mathrm{op}}$ are shown in Figure 4. We use Greek letters for $\Lambda_{\mathrm{op}}$ types, to distinguish them from the types of $\Lambda_T$ and it is important to note that we use different metavariables ($\sigma$,$\tau$) for value types and arbitrary types ($\delta$,$\gamma$). Types of the form

---

$$\begin{array}{lllll} \text{value types} & \sigma,\tau & ::= & \iota \mid \delta \to \gamma \\ \text{types} & \delta,\gamma & ::= & \sigma \mid \sigma_\perp \\ \text{contexts} & \Theta & ::= & a_1{:}\,\delta_1, \ldots, a_n{:}\,\delta_n \end{array}$$

$$\mathrm{Id}\ \frac{}{\Theta, a{:}\,\delta \vdash a{:}\,\delta}$$

$$\mathrm{Abs}\ \frac{\Theta, a{:}\,\delta \vdash e{:}\,\sigma_\perp}{\Theta \vdash \lambda a{:}\,\delta.e{:}\,\delta \to \sigma_\perp} \qquad \mathrm{App}\ \frac{\Theta \vdash e{:}\,\delta \to \sigma_\perp \qquad \Theta \vdash f{:}\,\delta}{\Theta \vdash e\ f{:}\,\sigma_\perp}$$

$$\mathrm{Abs}'\ \frac{\Theta, a{:}\,\delta \vdash e{:}\,\sigma}{\Theta \vdash \underline{\lambda} a{:}\,\delta.e{:}\,\delta \to \sigma} \qquad \mathrm{App}'\ \frac{\Theta \vdash e{:}\,\delta \to \sigma \qquad \Theta \vdash f{:}\,\delta}{\Theta \vdash e\ f{:}\,\sigma_\perp}$$

$$\mathrm{Val}\ \frac{\Theta \vdash e{:}\,\sigma}{\Theta \vdash [e]{:}\,\sigma_\perp} \qquad \mathrm{Let}\ \frac{\Theta \vdash e{:}\,\sigma_\perp \qquad \Theta, a{:}\,\sigma \vdash f{:}\,\tau_\perp}{\Theta \vdash \mathit{let}\ a \leftarrow e\ \mathit{in}\ f{:}\,\tau_\perp}$$

$$\mathrm{Rec}\ \frac{\Theta, a{:}\,\sigma_\perp \vdash e{:}\,\sigma_\perp}{\Theta \vdash \mathit{rec}(a{:}\,\sigma_\perp.\ e){:}\,\sigma_\perp} \qquad \mathrm{Cond}\ \frac{\Theta \vdash e{:}\,\iota \qquad \Theta \vdash f_1{:}\,\sigma_\perp \qquad \Theta \vdash f_2{:}\,\sigma_\perp}{\Theta \vdash \mathit{if}\ e\ \mathit{then}\ f_1\ \mathit{else}\ f_2{:}\,\sigma_\perp}$$

$$\mathrm{Nat}\ \frac{}{\Theta \vdash \underline{n}{:}\,\iota} \qquad \mathrm{Arith}\ \frac{\Theta \vdash e{:}\,\iota \qquad \Theta \vdash f{:}\,\iota}{\Theta \vdash e\ \underline{op}\ f{:}\,\iota_\perp}$$

---

Figure 4: Syntax and type rules of $\Lambda_{\mathrm{op}}$

$\sigma_\perp$ will be referred to as computation types or lifted types. The version of $\Lambda_{\mathrm{op}}$ used here varies slightly from that originally proposed in [Ben92] in that it includes a variant form of lambda abstraction $\underline{\lambda}a.e$ in which the body is a value. This is slightly inelegant and not

really necessary, but has been included firstly because it corresponds more closely to what one would wish to do in an implementation (as it saves some unecessary evaluations), and secondly because it makes translated terms somewhat smaller.

The operational semantics of $\Lambda_{\mathrm{op}}$ is defined by big-step evaluation relation $\Downarrow$ which relates (closed) computations of type $\sigma_\perp$ to values of type $\sigma$. The operational semantics is shown in Figure 5. There are no rules in the operational semantics of $\Lambda_{\mathrm{op}}$ which say that

$$\frac{e[f/a] \Downarrow v}{(\lambda a{:}\,\delta.e)\ f \Downarrow v} \qquad \frac{}{(\underline{\lambda} a{:}\,\delta.e)\ f \Downarrow e[f/a]}$$

$$\frac{}{[v] \Downarrow v} \qquad \frac{e \Downarrow v \qquad f[v/a] \Downarrow v'}{\mathit{let}\ a \leftarrow e\ \mathit{in}\ f \Downarrow v'}$$

$$\frac{}{\underline{m}\ \underline{op}\ \underline{n} \Downarrow \underline{m\ op\ n}} \qquad \frac{e[rec(a{:}\,\sigma_\perp.\, e)/a] \Downarrow v}{rec(a{:}\,\sigma_\perp.\, e) \Downarrow v}$$

$$\frac{f_1 \Downarrow v}{\mathit{if}\ \underline{0}\ \mathit{then}\ f_1\ \mathit{else}\ f_2 \Downarrow v} \qquad \frac{f_2 \Downarrow v}{\mathit{if}\ \underline{n+1}\ \mathit{then}\ f_1\ \mathit{else}\ f_2 \Downarrow v}$$

Figure 5: Operational semantics of $\Lambda_{\mathrm{op}}$

canonicals evaluate to themselves, as there were for $\Lambda_T$, because the type system makes the fact that they are already in WHNF explicit.

$\Lambda_{\mathrm{op}}$ has a denotational semantics which uses $\omega$-cpos which are not necessarily pointed (predomains) and continuous maps. This semantics is shown in Figure 6. We use $\varrho$ for $\Lambda_{\mathrm{op}}$ envinronments, and again assume trivial standard results about the denotational semantics. Note that the only denotational difference between the two kinds of lambda-abstraction is in the types and that as we have not syntactically distinguished the two kinds of application, we have not distinguished them in the denotational semantics.

**Proposition 2 (Computational adequacy for $\Lambda_{\mathrm{op}}$)** *If $e$ is a closed $\Lambda_{\mathrm{op}}$ term of type $\sigma_\perp$ then*

$$[\![e]\!] = [d] \ \Leftrightarrow\ e \Downarrow v\ \&\ [\![v]\!] = d$$

**Proof.** This is a slight variant on the logical relations argument used to prove Proposition 1. □

Note that we have adequacy at all types for $\Lambda_{\mathrm{op}}$, but only at the ground type for $\Lambda_T$. The semantics validates various equational laws, but the three important ones involving computational types are:

$$\mathit{let}\ a \leftarrow [e]\ \mathit{in}\ f \ = \ f[e/a] \tag{1}$$

$$\mathit{let}\ a \leftarrow (\mathit{let}\ b \leftarrow e\ \mathit{in}\ f)\ \mathit{in}\ g \ = \ \mathit{let}\ b \leftarrow e\ \mathit{in}\ (\mathit{let}\ a \leftarrow f\ \mathit{in}\ g) \tag{2}$$

$$\mathit{let}\ a \leftarrow e\ \mathit{in}\ [a] \ = \ e \tag{3}$$

where Equation 2 carries the side-condition that $b$ is not free in $g$.[3]

[3] A proof-theoretic account of the the computational lambda calculus, including these rules, can be found in [BBdP95].

Types

$$\begin{array}{rcl} [\![\iota]\!] & = & IN \\ [\![\gamma \to \delta]\!] & = & [\![\gamma]\!] \to [\![\delta]\!] \\ [\![\sigma_\perp]\!] & = & [\![\sigma]\!]_\perp \end{array}$$

Terms

$$\begin{array}{rcl} [\![a]\!]\varrho & = & \varrho(a) \\ [\![\lambda a{:}\,\delta.e]\!]\varrho & = & \lambda d \in [\![\delta]\!].[\![e]\!]\varrho[a \mapsto d] \\ [\![\underline{\lambda} a{:}\,\delta.e]\!]\varrho & = & \lambda d \in [\![\delta]\!].[\![e]\!]\varrho[a \mapsto d] \\ [\![e\ f]\!]\varrho & = & ([\![e]\!]\varrho)\ ([\![f]\!]\varrho) \\ [\![[e]]\!]\varrho & = & [[\![e]\!]\varrho] \\ [\![let\ a \leftarrow e\ in\ f]\!]\varrho & = & \begin{cases} [\![f]\!]\varrho[a \mapsto d] & \text{if } [\![e]\!]\varrho = [d] \\ \perp & \text{otherwise} \end{cases} \\ [\![rec(a{:}\,\sigma_\perp.\ e)]\!]\varrho & = & \bigsqcup_{i\in\omega} d_i \quad \text{where } d_0 = \perp_{[\![\sigma_\perp]\!]} \text{ and } d_{n+1} = [\![e]\!]\varrho[a \mapsto d_n] \\ [\![if\ e\ then\ f_1\ else\ f_2]\!]\varrho & = & \begin{cases} [\![f_1]\!]\varrho & \text{if } [\![e]\!]\varrho = 0 \\ [\![f_2]\!]\varrho & \text{if } [\![e]\!]\varrho = n+1 \end{cases} \\ [\![\underline{n}]\!]\varrho & = & n \\ [\![e\ \underline{op}\ f]\!]\varrho & = & [([\![e]\!]\varrho)\ op\ ([\![f]\!]\varrho)] \end{array}$$

Figure 6: Denotational semantics of $\Lambda_{\mathrm{op}}$

# 4 Translating $\Lambda_T$ into $\Lambda_{\mathrm{op}}$

There is a well-known call-by-name translation of the lambda calculus into the computational lambda calculus due to Moggi [Mog89] which gives a natural default translation of $\Lambda_T$ into $\Lambda_{\mathrm{op}}$. Under this default translation, a typing judgement of the form $\Gamma \vdash e{:}\,A$ in $\Lambda_T$ is translated to a judgement $\Gamma^n_\perp \vdash e^n{:}\,A^n_\perp$ in $\Lambda_{\mathrm{op}}$, where $A^n$ is defined inductively as:

$$\mathtt{nat}^n = \iota \qquad\qquad (A \to B)^n = (A^n_\perp \to B^n_\perp)$$

In particular, a source term of functional type is translated into a target term which is (a computation of) a function from computations to computations. There is also a call-by-value translation, also due to Moggi, in which terms of functional type are translated into (computations of) functions from *values* to computations. [4]

The default translation of $\Lambda_T$ into $\Lambda_{\mathrm{op}}$ makes evaluation order very plain by using the *let* construct to evaluate computations of functional or ground type prior to their use in applications or arithmetic operations. Apart from the fact that there is no treatment of updating, the default translation produces results which correspond very closely to the real code that is produced by naive compilers for lazy functional languages. Our aim is to produce a better translation, which, crudely, means one which introduces fewer liftings (i.e. computations rather than values) in the types of translated programs. For example, the default translation of the following $\Lambda_T$ program:

$$(\lambda a{:}\,\mathtt{nat}.a + a)\ (\underline{3} + \underline{4})$$

[4] These translations can also be found in, for example, [Ben92, Cro92]. Their intimate connection with various translations into a language based on linear logic is the subject of [BW95].

is

$$\begin{array}{l} \mathit{let}\ f \leftarrow [\lambda a{:}\,\iota_\bot.\mathit{let}\ b \leftarrow a\ \mathit{in}\ \mathit{let}\ c \leftarrow a\ \mathit{in}\ b + c] \\ \quad \mathit{in}\ f\ (\mathit{let}\ x \leftarrow [\underline{3}]\ \mathit{in}\ \mathit{let}\ y \leftarrow [\underline{4}]\ \mathit{in}\ x + y) \end{array}$$

which is rather verbose, but we can use Equation 1 from the previous section to make some 'administrative reductions' and obtain:

$$(\lambda a{:}\,\iota_\bot.\mathit{let}\ b \leftarrow a\ \mathit{in}\ \mathit{let}\ c \leftarrow a\ \mathit{in}\ b + c)\ (\underline{3} + \underline{4})$$

However, because the function is strict we should prefer a translation in which the argument is evaluated before the call and in which the function is compiled to expect an evaluated argument:

$$\mathit{let}\ b \leftarrow \underline{3} + \underline{4}\ \mathit{in}\ (\lambda a{:}\,\iota.a + a)\ b$$

which will be derivable using the improved translation. Note that the improvement is not just that we save building a closure, but also that the repeated evaluation of that closure is avoided. Of course, real lazy implementations avoid this kind of re-evaluation by updating, but the updates themselves still have a cost and it is still necessary to perform a context switch to evaluate the closure for the second time, even though the evaluation will return immediately with the updated value.

We now describe the improved translation in more detail. To begin with, notice that for each $\Lambda_T$ term $e$ of type $A$, there are several $\Lambda_{\mathrm{op}}$ types $\delta$ of roughly the same 'shape' as $A$ which we might choose as the type of the translation of $e$. Each of these types can be regarded as a 'decoration' of the type $A$. We formalise this notion by defining a map $U$ (for *underlying*) from $\Lambda_{\mathrm{op}}$ types to $\Lambda_T$ types:

$$\begin{array}{rcl} U(\iota) & = & \mathtt{nat} \\ U(\gamma \to \delta) & = & U(\gamma) \to U(\delta) \\ U(\sigma_\bot) & = & U(\sigma) \end{array}$$

The translation is defined by a set of inference rules for deducing *translation judgements* of the form

$$(a_1{:}\,A_1, \ldots, a_n{:}\,A_n \vdash e{:}\,B) \ \ \triangleright\ \ (a_1{:}\,\delta_1, \ldots, a_n{:}\,\delta_n \vdash e'{:}\,\gamma)$$

where

- $a_1{:}\,A_1, \ldots, a_n{:}\,A_n \vdash e{:}\,B$ is a valid typing judgement in $\Lambda_T$
- $a_1{:}\,\delta_1, \ldots, a_n{:}\,\delta_n \vdash e'{:}\,\gamma$ is a valid typing judgement in $\Lambda_{\mathrm{op}}$
- For all $i$, $U(\delta_i) = A_i$
- $U(\gamma) = B$

Roughly speaking, the basic idea behind the translation is that in any derivable translation judgement, $\delta_i$ is a value type $\sigma$ only if $e$ is strict in $a_i$. Similarly, a source language function will only be translated into a target language function with a value type as argument if it is strict. As we have already noted, however, the strictness of $e$ in $a_i$ may depend on the strictness properties of some other free variable $a_j$, so the translation has to be able to cope with such conditional information too. For example, suppose that the following is a derivable translation judgement:

$$f{:}\,\mathtt{nat} \to \mathtt{nat}, a{:}\,\mathtt{nat} \vdash e{:}\,\mathtt{nat} \to \mathtt{nat} \ \ \triangleright\ \ f{:}\,(\iota \to \iota_\bot)_\bot, a{:}\,\iota \vdash e'{:}\,\iota \to \iota_\bot$$

The intuitive reading of this in terms of strictness properties of $e$ is that $e$ is not necessarily strict in $f$ (since the translated type of $f$ is lifted), but that if $f$ is itself a strict function (the translated type of $f$ is a computation of a function from *values* to computations) then $e$ is strict in $a$ (the translated type of $a$ is unlifted) and, moreover, $e$ is then itself a WHNF of a strict function (it translates as a value which is a function from values to computations). The target term $e'$ is a translation of $e$ which assumes that $f$ will evaluate to a strict function expecting an evaluated argument and that $a$ will already be evaluated. Finally, $e'$ itself expects an evaluated argument.

As the preceding explanation shows, giving a clear, intuitive definition of precisely which translations we regard as correct is slightly tricky. It should be stressed, however, that we can give a precise *formal* definition of correctness, and that we do so in the next section.

Because translated terms will, in general, contain administrative redexes, there is some choice about exactly how to present the inference rules which define the translation. One way is to try to build as much peephole optimisation as possible into the translation process itself. This, however, has the effect of increasing considerably the number of translation rules. Whilst this can be alleviated by the use of auxiliary macros, it still complicates the translation and gives more cases for the correctness proof. Since it does not seem easy to remove *all* the administrative redexes by complicating the translation in this way, we have instead opted for a presentation which keeps the inference rules simple at the expense of introducing more administrative redexes. The removal of administrative redexes is then performed by repeatedly applying Equations 1, 2 and 3 as rewrite rules (orienting them from left to right) until no further simplification is possible.[5] Note that one advantage of our separation of computations from values is that the distinction between what we regard as an administrative redex, to be removed at compile-time, and what we regard as a 'real' redex, to be evaluated at run-time, is a very natural one. This is in contrast to the situation for CPS transformations, for which some authors have suggested an extra level of labelling on terms to distinguish those applications and abstractions which are introduced by the transformation itself from those present in the original source program so that redexes introduced by the transformation can be removed at compile-time. When giving examples of derivable translations, we will usually perform the removal of administrative redexes without explicitly mentioning it.

The rules defining the translation are shown in Figure 7. We use the notational conventions that distinct contexts mention distinct sets of variable names, that variable names which occur in the conclusion of a rule but not in the hypotheses are always fresh, and that $\Theta_\perp$ stands for a $\Lambda_{\mathrm{op}}$ context in which every variable is given a lifted type. The translation has several interesting features, the most obvious of which is that we have made contraction explicit and used multiplicative (disjoint) contexts everywhere except in the two arms of the conditional. This is because the subexpressions of compound expressions (such as arithmetic expressions) will generally be strict in different variables. We deal with this by making the variables distinct, concatenating the contexts and then using the contraction rules (C1,C2) to merge distinct variables together in a controlled way. Weakening is built into the (Id) and (Nat) rules, and is only allowed on lifted types, since any variable introduced by weakening is not one in which the associated expression is strict. Similarly, note that in the (AppNS) and (AppNS$'$) rules, all the variables used to derive

[5] That this process terminates follows from a small modification to the strong normalisation proof of [BBdP95]

$$\text{Id}\ \frac{}{\Gamma, a\colon A \vdash a\colon A \ \ \triangleright\ \ \Gamma'_\perp, a\colon \delta \vdash a\colon \delta}\ (U(\delta) = A,\ U(\Gamma'_\perp) = \Gamma)$$

$$\text{Nat}\ \frac{}{\Gamma \vdash \underline{n}\colon \mathtt{nat} \ \ \triangleright\ \ \Gamma'_\perp \vdash \underline{n}\colon \iota}\ (U(\Gamma'_\perp) = \Gamma)$$

$$\text{Val}\ \frac{\Gamma \vdash e\colon A \ \ \triangleright\ \ \Gamma' \vdash e'\colon \sigma}{\Gamma \vdash e\colon A \ \ \triangleright\ \ \Gamma' \vdash [e']\colon \sigma_\perp}$$

$$\text{Let}\ \frac{\Gamma, a\colon A \vdash e\colon B \ \ \triangleright\ \ \Gamma', a\colon \sigma \vdash e'\colon \tau_\perp}{\Gamma, b\colon A \vdash e[b/a]\colon B \ \ \triangleright\ \ \Gamma', b\colon \sigma_\perp \vdash \mathit{let}\ a \leftarrow b\ \mathit{in}\ e'\colon \tau_\perp}$$

$$\text{C1}\ \frac{\Gamma, a\colon A, b\colon A \vdash e\colon B \ \ \triangleright\ \ \Gamma', a\colon \delta, b\colon \delta \vdash e'\colon \gamma}{\Gamma, c\colon A \vdash e[c/a, c/b]\colon B \ \ \triangleright\ \ \Gamma', c\colon \delta \vdash e'[c/a, c/b]\colon \gamma}$$

$$\text{C2}\ \frac{\Gamma, a\colon A, b\colon A \vdash e\colon B \ \ \triangleright\ \ \Gamma', a\colon \sigma, b\colon \sigma_\perp \vdash e'\colon \gamma}{\Gamma, c\colon A \vdash e[c/a, c/b]\colon B \ \ \triangleright\ \ \Gamma', c\colon \sigma \vdash e'[c/a, [c]/b]\colon \gamma}$$

$$\text{Abs}\ \frac{\Gamma, a\colon A \vdash e\colon B \ \ \triangleright\ \ \Gamma', a\colon \delta \vdash e'\colon \sigma_\perp}{\Gamma \vdash (\lambda a\colon A.e)\colon A \to B \ \ \triangleright\ \ \Gamma' \vdash (\lambda a\colon \delta.e')\colon \delta \to \sigma_\perp}$$

$$\text{Abs}'\ \frac{\Gamma, a\colon A \vdash e\colon B \ \ \triangleright\ \ \Gamma', a\colon \delta \vdash e'\colon \sigma}{\Gamma \vdash (\lambda a\colon A.e)\colon A \to B \ \ \triangleright\ \ \Gamma' \vdash (\underline{\lambda} a\colon \delta.e')\colon \delta \to \sigma}$$

$$\text{AppS}\ \frac{\Gamma \vdash e\colon A \to B \ \ \triangleright\ \ \Gamma' \vdash e'\colon (\sigma \to \tau_\perp)_\perp \qquad \Delta \vdash f\colon A \ \ \triangleright\ \ \Delta' \vdash f'\colon \sigma_\perp}{\Gamma, \Delta \vdash e\ f\colon B \ \ \triangleright\ \ \Gamma', \Delta' \vdash \mathit{let}\ x \leftarrow e'\ \mathit{in}\ \mathit{let}\ y \leftarrow f'\ \mathit{in}\ x\ y\colon \tau_\perp}$$

$$\text{AppS}'\ \frac{\Gamma \vdash e\colon A \to B \ \ \triangleright\ \ \Gamma' \vdash e'\colon (\sigma \to \tau)_\perp \qquad \Delta \vdash f\colon A \ \ \triangleright\ \ \Delta' \vdash f'\colon \sigma_\perp}{\Gamma, \Delta \vdash e\ f\colon B \ \ \triangleright\ \ \Gamma', \Delta' \vdash \mathit{let}\ x \leftarrow e'\ \mathit{in}\ \mathit{let}\ y \leftarrow f'\ \mathit{in}\ x\ y\colon \tau_\perp}$$

$$\text{AppNS}\ \frac{\Gamma \vdash e\colon A \to B \ \ \triangleright\ \ \Gamma' \vdash e'\colon (\sigma_\perp \to \tau_\perp)_\perp \qquad \Delta \vdash f\colon A \ \ \triangleright\ \ \Delta'_\perp \vdash f'\colon \sigma_\perp}{\Gamma, \Delta \vdash e\ f\colon B \ \ \triangleright\ \ \Gamma', \Delta'_\perp \vdash \mathit{let}\ x \leftarrow e'\ \mathit{in}\ x\ f'\colon \tau_\perp}$$

$$\text{AppNS}'\ \frac{\Gamma \vdash e\colon A \to B \ \ \triangleright\ \ \Gamma' \vdash e'\colon (\sigma_\perp \to \tau)_\perp \qquad \Delta \vdash f\colon A \ \ \triangleright\ \ \Delta'_\perp \vdash f'\colon \sigma_\perp}{\Gamma, \Delta \vdash e\ f\colon B \ \ \triangleright\ \ \Gamma', \Delta'_\perp \vdash \mathit{let}\ x \leftarrow e'\ \mathit{in}\ x\ f'\colon \tau_\perp}$$

$$\text{Rec}\ \frac{\Gamma, a\colon A \vdash e\colon A \ \ \triangleright\ \ \Gamma', a\colon \sigma_\perp \vdash e'\colon \sigma_\perp}{\Gamma \vdash \mathtt{rec}(a\colon A.\ e)\colon A \ \ \triangleright\ \ \Gamma' \vdash \mathit{rec}(a\colon \sigma_\perp.\ e')\colon \sigma_\perp}$$

$$\text{Arith}\ \frac{\Gamma \vdash e\colon \mathtt{nat} \ \ \triangleright\ \ \Gamma' \vdash e'\colon \iota_\perp \qquad \Delta \vdash f\colon \mathtt{nat} \ \ \triangleright\ \ \Delta' \vdash f'\colon \iota_\perp}{\Gamma, \Delta \vdash e\ \underline{op}\ f\colon \mathtt{nat} \ \ \triangleright\ \ \Gamma', \Delta' \vdash \mathit{let}\ x \leftarrow e'\ \mathit{in}\ \mathit{let}\ y \leftarrow f'\ \mathit{in}\ x\ \underline{op}\ y\colon \iota_\perp}$$

$$\text{Cond}\ \frac{\Gamma \vdash e\colon \mathtt{nat} \ \ \triangleright\ \ \Gamma' \vdash e'\colon \iota_\perp \qquad \Delta \vdash f\colon A \ \ \triangleright\ \ \Delta' \vdash f'\colon \sigma_\perp \qquad \Delta \vdash g\colon A \ \ \triangleright\ \ \Delta' \vdash g'\colon \sigma_\perp}{\Gamma, \Delta \vdash \mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g\colon A \ \ \triangleright\ \ \Gamma', \Delta' \vdash \mathit{let}\ x \leftarrow e'\ \mathit{in}\ \mathit{if}\ x\ \mathit{then}\ f'\ \mathit{else}\ g'\colon \sigma_\perp}$$

Figure 7: Optimising translation of $\Lambda_T$ into $\Lambda_{\text{op}}$

the typing of the argument must be lifted, since they might not be used if the function turns out not to need the argument.

The observant reader will notice that the strictness type system which morally underlies the optimising translation is essentially based on intuitionistic relevance logic. The basic idea of relevance logic is that in proving a sequent $\Gamma \vdash A$, all the assumptions in $\Gamma$ must actually be used at least once in proving $A$, which is enforced by restricting the weaking rule. This contrasts with linear logic, which restricts both weakening and contraction so that all the assumptions must be used *exactly* once. Just as linear logic reintroduces weakening and contraction in a controlled way, via the exponential modality !, so one can add a modality to intuitionistic relevance logic to reintroduce, but control, weakening. In our system, the role of this modality is played by the lifting operator of $\Lambda_{\mathrm{op}}$: see the (Id) and (Nat) rules of the translation. The language $\Lambda_{\mathrm{op}}$ is, however, *not* the term calculus which arises by the Curry-Howard isomorphism from such a relevance logic (though see [BBdP95]). $\Lambda_{\mathrm{op}}$ allows unrestricted weakening and contraction, but the strictness translation prevents weakening being used to introduce variables of unlifted type. The idea of relevance also lies behind Wright's work [Wri92] on 'neededness analysis' and the connection with relevance logic has been made more explicit by Baker-Finch [BF92]. Their work is concerned only with analysis and formulates correctness in terms of the syntactic notion of 'neededness', which is defined via a labelled reduction system which tracks the descendents of individual redexes through $\beta$-reduction.

Clearly, we need to check that for every $\Lambda_T$ term, there is *some* $\Lambda_{\mathrm{op}}$ term to which it translates. But this is easy, as we can just use Moggi's call-by-name translation:

**Lemma 3** *If* $\Gamma \vdash e\colon A$ *then* $\Gamma \vdash e\colon A \quad \triangleright \quad \Gamma^n_\perp \vdash e^n\colon A^n_\perp$. □

However, the point is that in general we can do rather better than Moggi's translation. For example:

1. $\vdash (\lambda a\colon \mathtt{nat}.a + a)\ (\underline{3} + \underline{4})\colon \mathtt{nat} \quad \triangleright \quad \vdash \mathit{let}\ b \leftarrow \underline{3} + \underline{4}\ \mathit{in}\ (\lambda a\colon \iota.a + a)\ b\colon \iota_\perp$, which was the motivating example we gave earlier.

2. For the factorial function, we obtain

$$\begin{array}{l}\vdash \mathtt{rec}(f\colon \mathtt{nat} \to \mathtt{nat}.\ \lambda n\colon \mathtt{nat}.\mathtt{if}\ n\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ n * (f\ (n - \underline{1}))) \colon \mathtt{nat} \to \mathtt{nat}\ \triangleright \\ \vdash \mathit{rec}(f\colon (\iota \to \iota_\perp)_\perp.\ [\lambda n\colon \iota.\mathit{if}\ n\ \mathit{then}\ [\underline{1}]\ \mathit{else}\ \mathit{let}\ f' \leftarrow f\ \mathit{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{let}\ n_1 \leftarrow (n - \underline{1})\ \mathit{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{let}\ n_2 \leftarrow (f'\ n_1)\ \mathit{in}\ n * n_2]) \colon (\iota \to \iota_\perp)_\perp\end{array}$$

   which, as we would hope, recognises that the function is strict and so compiles it to expect an evaluated argument. Note that the argument to the recursive call is evaluated eagerly, just as it would be in a strict language.

3. Here's a higher-order example:

$$\begin{array}{l}\vdash (\lambda f\colon \mathtt{nat} \to \mathtt{nat}.\lambda n\colon \mathtt{nat}.\mathtt{if}\ n\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ f\ (n + \underline{1}))\ (\lambda m\colon \mathtt{nat}.m + \underline{1}) \colon \mathtt{nat} \to \mathtt{nat} \\ \triangleright \\ \vdash (\underline{\lambda} f\colon (\iota \to \iota_\perp)_\perp.\lambda n\colon \iota.\mathit{if}\ n\ \mathit{then}\ [\underline{1}]\ \mathit{else} \\ \qquad\qquad\qquad\qquad \mathit{let}\ f' \leftarrow f\ \mathit{in} \\ \qquad\qquad\qquad\qquad \mathit{let}\ n' \leftarrow (n + \underline{1})\ \mathit{in}\ f'\ n')\ [\lambda m\colon \iota.m + \underline{1}] \colon (\iota \to \iota_\perp)_\perp\end{array}$$

   Here, although the higher-order function is not strict in $f$, it is compiled to expect a strict function as argument, so the application in the *else* branch of the conditional

has the argument passed by value. Note also that the higher-order function returns a WHNF immediately (the use of $\underline{\lambda} f \ldots$ rather than $\lambda f \ldots$), but that the translation cannot exploit the fact that the argument to the higher-order function is itself already a WHNF and so could have been passed by value.

The following is a *non*-example, which reveals a weakness of the analysis built into this system:

$$\begin{array}{l} x\colon \mathtt{nat}, w\colon \mathtt{nat} \vdash (\lambda y\colon \mathtt{nat}.\lambda z\colon \mathtt{nat}.\mathtt{if}\ x\ \mathtt{then}\ y + \underline{1}\ \mathtt{else}\ z + \underline{2})\ w\ w\colon \mathtt{nat} \not\vdash \\ x\colon \iota, w\colon \iota \vdash \mathit{let}\ f \leftarrow (\underline{\lambda} y\colon \iota.\lambda z\colon \iota.\mathit{if}\ x\ \mathit{then}\ y + \underline{1}\ \mathit{else}\ z + \underline{2})\ w \\ \qquad\qquad \mathit{in}\ f\ w\colon \iota_\perp \end{array}$$

The problem here is that the expression is strict in $w$, since whichever branch of the conditional is chosen, $w$ will be evaluated; this cannot be detected in our system because the function containing the conditional is strict in neither of $y$ or $z$ alone. The fact that this expression really *is* strict in $w$ is detectable even in the strictness logic of [Ben92] *without* conjunction, a system which is itself weaker than the standard abstract interpretation of [BHA86]. There are, however, also examples which are detected by this system but are missed by the conjunction-free strictness logic, so these two systems are incomparable in terms of accuracy. Both are strictly weaker than [BHA86], or the equivalent conjunctive strictness logic.

# 5 Correctness of the translation

We now turn to the question of showing that our optimising translation is correct. The criterion for correctness which we naturally adopt is that for any source program $p$ and for any translation $p'$ of $p$, $p$ evaluates to a result $v$ iff $p'$ evaluates to $v$. We shall establish this result via a logical relation, indexed by $\Lambda_{\mathrm{op}}$ types, between the domains used in the semantics of $\Lambda_T$ and the predomains used in the semantics of $\Lambda_{\mathrm{op}}$. Thus for each $\delta$, we have

$$\mathcal{R}_\delta \subseteq [\![U(\delta)]\!] \times [\![\delta]\!]$$

(and we will often use infix notation for $\mathcal{R}$). The relation expresses the sense in which a source term and its translation are 'equivalent'. Unsurprisingly, the definition of the relation has to make reference to some notion of strictness, but this has to be done with some care in order for the proof to work. We make use of a family $\nabla_\delta \subseteq [\![U(\delta)]\!]$ of subsets of the source language domains which are defined simultaneously with the relations $\mathcal{R}_\delta$. One should think of $\nabla_\delta$ as, roughly, the collection of elements of $[\![U(\delta)]\!]$ which are indistinguishable from $\perp$ in all source contexts which translate to target contexts with a hole of type $\delta$. We shall also use the abbreviation

$$\mathcal{I}_\delta = \{x \in [\![U(\delta)]\!] \mid \exists y \in [\![\delta]\!].(x, y) \in \mathcal{R}_\delta\}$$

The definitions of $\mathcal{R}$ and $\nabla$ are as follows:

$$\begin{array}{rcl} \mathcal{R}_\iota & = & \{([n], n) \mid n \in IN\} \\ \nabla_\iota & = & \{\perp\} \\ \mathcal{R}_{\sigma_\perp} & = & \{(a, [d]) \mid (a, d) \in \mathcal{R}_\sigma\} \cup \{(x, \perp) \mid x \in \nabla_\sigma\} \\ \nabla_{\sigma_\perp} & = & \nabla_\sigma \end{array}$$

$$\begin{aligned}
\mathcal{R}_{\sigma\to\delta} &= \{(f,g) \mid \forall x \in \nabla_\sigma . f(x) \in \nabla_\delta,\ \forall (x,y) \in \mathcal{R}_\sigma . (f\ x, g\ y) \in \mathcal{R}_\delta\} \\
\mathcal{R}_{\sigma_\perp\to\delta} &= \{(f,g) \mid \forall (x,y) \in \mathcal{R}_{\sigma_\perp} . (f\ x, g\ y) \in \mathcal{R}_\delta\} \\
\nabla_{\gamma\to\delta} &= \{f \mid \forall x \in \mathcal{I}_\gamma \cup \nabla_\gamma . f(x) \in \nabla_\delta\}
\end{aligned}$$

Note that the definition of $\mathcal{R}$ at function types is of the usual 'takes related arguments to related results' form, but that at types of the form $\sigma \to \delta$ there is an additional requirement that $f$ be strict, in a suitably generalised sense. It is a simple induction on types to show that all the $\mathcal{R}_\delta$ are inclusive and that all the $\nabla_\delta$ are ideals, which we shall need later:

**Lemma 4** *For all $\delta$*

1. *If $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ is an $\omega$-chain in $[\![U(\delta)]\!]$ and $e_0 \sqsubseteq e_1 \sqsubseteq \cdots$ is an $\omega$-chain in $[\![\delta]\!]$ such that for all $i \in \omega . (d_i, e_i) \in \mathcal{R}_\delta$ then $(\bigsqcup_i d_i, \bigsqcup_i e_i) \in \mathcal{R}_\delta$.*

2. *The set $\nabla_\delta$ is non-empty, downwards closed and closed under limits of $\omega$-chains.*

□

Now correctness follows from the following theorem, which is in the spirit of the 'fundamental theorem of logical relations':

**Theorem 5** *If the translation judgement*

$$\Gamma \vdash e : B \quad \triangleright \quad \Gamma' \vdash e' : \delta$$

*is derivable, where*

$$\begin{aligned}
\Gamma &= a_1 : A_1, \ldots, a_n : A_n \\
\Gamma' &= a_1 : \sigma_{1_\perp}, \ldots, a_m : \sigma_{m_\perp}, a_{m+1} : \sigma_{m+1}, \ldots, a_n : \sigma_n
\end{aligned}$$

*then*

1. *For all $\rho : \Gamma$, $\varrho : \Gamma'$ such that $\rho\, \mathcal{R}\, \varrho$ (pointwise), $([\![e]\!]\rho)\ \mathcal{R}_\delta\ ([\![e']\!]\varrho)$.*

2. *If $\rho : \Gamma$ satisfies the following three conditions:*

   *(a)* $\forall 1 \leq i \leq m.\ \rho(a_i) \in \mathcal{I}_{\sigma_{i\perp}}$

   *(b)* $\forall m < j \leq n.\ \rho(a_j) \in \mathcal{I}_{\sigma_j} \cup \nabla_{\sigma_j}$

   *(c)* $\exists m < j \leq n.\ \rho(a_j) \in \nabla_{\sigma_j}$

   *then $[\![e]\!]\rho \in \nabla_\delta$.*

**Proof.** This follows by an induction on the derivation of the translation judgement. We give a few interesting cases:

**Val** For the first part we have to show that for any suitable $\rho$ and $\varrho$, $[\![e]\!]\rho \mathcal{R}_{\sigma_\perp} [\![[e']\!]]\varrho$. But $[\![[e']\!]]\varrho = [[\![e]\!]\varrho]$ so this is immediate from the induction hypothesis and the definition of $\mathcal{R}_{\sigma_\perp}$. For the second part, if $\rho$ satisfies the conditions given then it trivially satisfies the the conditions for part 2 applied to the hypothesis of the rule. Hence by induction $[\![e]\!]\rho \in \nabla_\sigma$ and since $\nabla_{\sigma_\perp} = \nabla_\sigma$ we are done.

**Let** For part 1, assume that $\rho\colon\Gamma$, $\varrho\colon\Gamma'$ with $\rho\mathcal{R}\varrho$ and that $x \in [\![A]\!]$, $y \in [\![\sigma_\perp]\!]$ with $x\mathcal{R}_{\sigma_\perp}y$. We want to show that

$$([\![e[b/a]]\!]\rho[b \mapsto x])\ \mathcal{R}_{\tau_\perp}\ ([\![\mathit{let}\ a \leftarrow b\ \mathit{in}\ e']\!]\varrho[b \mapsto y])$$

By the definition of $\mathcal{R}_{\sigma_\perp}$, there are two cases to consider: either $x \in \nabla_\sigma$ and $y = \perp$ or $y = [y']$ with $x\mathcal{R}_\sigma y'$. In the first case, $\rho[a \mapsto x]$ satisfies the conditions for part 2 of the induction hypothesis, so that

$$[\![e[b/a]]\!]\rho[b \mapsto x] \ = \ [\![e]\!]\rho[a \mapsto x] \ \in \ \nabla_{\tau_\perp}$$

and $[\![\mathit{let}\ a \leftarrow b\ \mathit{in}\ e']\!]\varrho[b \mapsto y] = \perp$ and we are done by the definition of $\mathcal{R}_{\tau_\perp}$. In the second case, $\rho[a \mapsto x]$ and $\varrho[a \mapsto y']$ satisfiy the conditions for part 1 of the induction hypothesis, so we can deduce

$$([\![e[b/a]]\!]\rho[b \mapsto x]) = ([\![e]\!]\rho[a \mapsto x])\ \mathcal{R}_{\tau_\perp}\ ([\![e']\!]\varrho[a \mapsto y']) = ([\![\mathit{let}\ a \leftarrow b\ \mathit{in}\ e']\!]\varrho[b \mapsto [y']])$$

as required. For part 2, if $\rho[b \mapsto x]$ satisfies the relevant conditions, then $\rho[a \mapsto x]$ satisfies the conditions for part 2 of the induction hypothesis, so that

$$([\![e[b/a]]\!]\rho[b \mapsto x]) \ = \ ([\![e]\!]\rho[a \mapsto x]) \ \in \ \nabla_{\tau_\perp}$$

as required.

**C2** For part 1, assume $\rho\colon\Gamma,\varrho\colon\Gamma'$ with $\rho\mathcal{R}\varrho$ and that $x \in [\![A]\!]$, $y \in [\![\sigma]\!]$ with $x\mathcal{R}_\sigma y$. Then $x\ \mathcal{R}_{\sigma_\perp}\ [y]$ so that

$$\begin{array}{rcll} [\![e[c/a, c/b]]\!]\rho[c \mapsto x] & = & [\![e]\!]\rho[a \mapsto x, b \mapsto x] & \\ & \mathcal{R}_\gamma & [\![e']\!]\varrho[a \mapsto y, b \mapsto [y]] & \text{by induction 1} \\ & = & [\![e'[c/a, [c]/b]]\!]\varrho[c \mapsto y] & \end{array}$$

as required. For part 2, it is easy to see that if $\rho[c \mapsto x]$ satisfies the relevant conditions then $\rho[a \mapsto x, b \mapsto x]$ satisfies the conditions for part 2 of the induction hypothesis, so

$$([\![e[c/a, c/b]]\!]\rho[c \mapsto x]) \ = \ ([\![e]\!]\rho[a \mapsto x, b \mapsto x]) \ \in \ \nabla_\gamma$$

**Abs** We consider the case where $\delta = \tau$, i.e. we are introducing a strict function. The case $\delta = \tau_\perp$ is similar. For part 1, assume that $\rho\mathcal{R}\varrho$. We have to show

$$(\lambda x \in [\![A]\!].[\![e]\!]\rho[a \mapsto x]) \quad \mathcal{R}_{\tau\to\sigma_\perp} \quad (\lambda y \in [\![\tau]\!].[\![e']\!]\varrho[a \mapsto y])$$

By the definition of $\mathcal{R}_{\tau\to\sigma_\perp}$, this means that we firstly have to show that if $x \in \nabla_\tau$, then $[\![e]\!]\rho[a \mapsto x] \in \nabla_{\sigma_\perp}$. But this follows from part 2 of the induction hypothesis, since it is easy to see that $\rho[a \mapsto x]$ satisfies the appropriate conditions. Secondly, we have to show that if $x\mathcal{R}_\tau y$ then $[\![e]\!]\rho[a \mapsto x]\ \mathcal{R}_{\sigma_\perp}\ [\![e']\!]\varrho[a \mapsto y]$, which follows from part 1 of the induction hypothesis.

For part 2, assume that $\rho$ satisfies the three conditions, then we have to show that $[\![\lambda a\colon A.e]\!]\rho \in \nabla_{\tau\to\sigma_\perp}$. This means showing that if $x \in \mathcal{I}_\tau \cup \nabla_\tau$ then $[\![e]\!]\rho[a \mapsto x] \in \nabla_{\sigma_\perp}$. This follows by part 2 of the induction hypothesis, since for any such $x$, $\rho[a \mapsto x]$ satisfies the appropriate conditions.

**AppS** For part 1 we assume that $\rho_1 : \Gamma, \varrho_1 : \Gamma'$, $\rho_2 : \Delta, \varrho : \Delta'$ with $\rho_1 \mathcal{R} \varrho_1$ and $\rho_2 \mathcal{R} \varrho_2$. By induction 1, we know that $[\![e]\!]\rho_1 \mathcal{R}_{(\sigma \to \tau_\perp)_\perp} [\![e']\!]\varrho_1$ and $[\![f]\!]\rho_2 \mathcal{R}_{\sigma_\perp} [\![f']\!]\varrho_2$.

Hence either (i) $[\![e]\!]\rho_1 \in \nabla_{\sigma \to \tau_\perp}$ and $[\![e']\!]\varrho_1 = \perp$
or (ii) $[\![e']\!]\varrho_1 = [x']$ with $[\![e]\!]\rho_1 \mathcal{R}_{\sigma \to \tau_\perp} x'$
And either (a) $[\![f]\!]\rho_2 \in \nabla_\sigma$ and $[\![f']\!]\varrho_2 = \perp$
or (b) $[\![f']\!]\varrho_2 = [y']$ with $[\![f]\!]\rho_2 \mathcal{R}_\sigma y'$

In case (i), whichever of (a) or (b) holds, $[\![f]\!]\rho_2 \in \mathcal{I}_\sigma \cup \nabla_\sigma$ so that by, the definition of $\nabla_{\sigma \to \tau_\perp}$, $([\![e]\!]\rho_1)([\![f]\!]\rho_2) \in \nabla_{\tau_\perp}$. Hence

$$([\![e]\!]\rho_1)([\![f]\!]\rho_2) \; \mathcal{R}_{\tau_\perp} \; ([\![\mathit{let}\ x \leftarrow e'\ \mathit{in}\ \mathit{let}\ y \leftarrow f'\ \mathit{in}\ x\ y]\!]\varrho_1\varrho_2) \;=\; \perp$$

In case (ii), if (a) holds then the strictness part of the definition of $\mathcal{R}_{\sigma \to \tau_\perp}$ gives that $([\![e]\!]\rho_1)([\![f]\!]\rho_2) \in \nabla_{\tau_\perp}$ again, and because $[\![f']\!]\varrho_2 = \perp$ we can then conclude that the relation holds as above. If (b) holds then by the logical relation part of the definition of $\mathcal{R}_{\sigma \to \tau_\perp}$ we get

$$([\![e]\!]\rho_1)([\![f]\!]\rho_2) \; \mathcal{R}_{\tau_\perp} \; (x'\ y') \;=\; ([\![\mathit{let}\ x \leftarrow e'\ \mathit{in}\ \mathit{let}\ y \leftarrow f'\ \mathit{in}\ x\ y]\!]\varrho_1\varrho_2)$$

as required.

For part 2, assume that $\rho_1 : \Gamma, \rho_2 : \Delta$ and that the concatenated environment $\rho_1\rho_2$ satisfies the three conditions. Then at least one of $\rho_1$ and $\rho_2$ also satisfies the three conditions on its own (there is at least one variable which is assigned an unlifted type in $\Gamma', \Delta'$ which is bound to an element of the appropriate $\nabla$ by $\rho_1\rho_2$). If $\rho_1$ satisfies the conditions for part 2, then by induction 2, $[\![e]\!]\rho_1 \in \nabla_{\sigma \to \tau_\perp}$. Now, if $\rho_2$ also satisfies the conditions for part 2, we can apply induction 2 to deduce that $[\![f]\!]\rho_2 \in \nabla_\sigma$ and hence $([\![e]\!]\rho_1)([\![f]\!]\rho_2) \in \nabla_{\tau_\perp}$ as required. If, on the other hand $\rho_2$ does not satisfy the three conditions, we must have that for all $a_i : \delta_i \in \Delta$, $\rho_2(a_i) \in \mathcal{I}_{\delta_i}$. And this means that we can apply induction 1 to deduce that $[\![f]\!]\rho_2 \in \mathcal{I}_{\sigma_\perp} = \mathcal{I}_\sigma \cup \nabla_\sigma$. Hence $([\![e]\!]\rho_1)([\![f]\!]\rho_2) \in \nabla_{\tau_\perp}$ again.

If $\rho_1$ does not satisfy the three conditions for part 2 but $\rho_2$ does, then we can apply induction 1 to deduce that $[\![e]\!]\rho_1 \in \mathcal{I}_{(\sigma \to \tau_\perp)_\perp} = \mathcal{I}_{\sigma \to \tau_\perp} \cup \nabla_{\sigma \to \tau_\perp}$ and we can use induction 2 to deduce that $[\![f]\!]\rho_2 \in \nabla_\sigma$. Hence, using either the strictness part of the definition of $\mathcal{R}_{\sigma \to \tau_\perp}$ or the definition of $\nabla_{\sigma \to \tau_\perp}$ according to which part of the union $[\![e]\!]\rho_1$ lies in, we find that $([\![e]\!]\rho_1)([\![f]\!]\rho_2) \in \nabla_{\tau_\perp}$ as required.

**Rec** For part 1, given appropriate $\rho, \varrho$, define $d_0 = \perp_{[\![A]\!]}$, $d_{n+1} = [\![e]\!]\rho[a \mapsto d_n]$ and $d'_0 = \perp_{[\![\sigma_\perp]\!]}$, $d'_{n+1} = [\![e']\!]\varrho[a \mapsto d_n]$. We claim that for all $n$, $d_n \mathcal{R}_{\sigma_\perp} d'_n$, which follows by a little induction. For the base case, observe that by the second part of Lemma 4, $\perp_{[\![A]\!]} \in \nabla_{\sigma_\perp}$ and thus, by the definition of $\mathcal{R}_{\sigma_\perp}$, $d_0 \; \mathcal{R}_{\sigma_\perp} \; d'_0$. Now for the induction step, we assume $d_n \mathcal{R}_{\sigma_\perp} d'_n$ so that $\rho[a \mapsto d_n] \, \mathcal{R} \, \varrho[a \mapsto d'_n]$ and we can apply induction 1 to deduce that

$$d_{n+1} \;=\; ([\![e]\!]\rho[a \mapsto d_n]) \quad \mathcal{R}_{\sigma_\perp} \quad ([\![e']\!]\varrho[a \mapsto d'_n]) \;=\; d'_{n+1}$$

And so by the first part of Lemma 4

$$[\![\mathtt{rec}(a{:}\,A.\ e)]\!]\rho \;=\; \bigsqcup_{i \in \omega} d_i \quad \mathcal{R}_{\sigma_\perp} \quad \bigsqcup_{i \in \omega} d'_i \;=\; [\![\mathit{rec}(a{:}\,\sigma_\perp.\ e')]\!]\varrho$$

as required.

For part 2, if $\rho$ satisfies the three conditions, then with $d_n$ defined as above it is another easy induction on $n$, using induction 2, to show that for all $n$, $d_n \in \nabla_{\sigma_\perp}$. Then as $\nabla_{\sigma_\perp}$ is closed under limits of chains (Lemma 4),

$$[\![\mathtt{rec}(a{:}\,A.\,e)]\!]\rho \;=\; \bigsqcup_{i\in\omega} d_i \;\in\; \nabla_{\sigma_\perp}$$

as required.

□

**Corollary 6** *For any program $p$, if the translation judgement $\vdash p{:}\mathtt{nat} \;\triangleright\; \vdash p'{:}\,\iota_\perp$ is derivable then for any $n \in IN$, $p \Downarrow \underline{n}$ iff $p' \Downarrow \underline{n}$.*

**Proof.** By Theorem 5, $[\![p]\!]\,\mathcal{R}_{\iota_\perp}\,[\![p']\!]$. By the definition of $\mathcal{R}_{\iota_\perp}$, this means that $[\![p]\!] = [n]$ iff $[\![p']\!] = [n]$ and the result then follows from Propositions 1 and 2. □

Strictly speaking, a further correctness result holds as a corollary of Theorem 5. This states that if $\vdash p{:}\mathtt{nat} \;\triangleright\; \vdash p'{:}\,\iota$ then $p \Downarrow \underline{n}$ iff $p' = \underline{n}$, but as this only happens when the source program is just a numeric literal, it has rather limited scope.

It is interesting to note that the proof of Theorem 5 reveals that the simple-minded strictness analysis which underlies the translation is correct, but surprisingly delicate. The semantics of our source language identifies $\perp$ and $\lambda x.\perp$, which does not cause adequacy to fail because we restrict our observations to whole programs, so that termination at higher types is unobservable. We make use of this identification in the definition of $\nabla_{\gamma\to\delta}$ and, in fact, our translation would be unsound if we added termination testing at higher types to the source language. The problem is in the rules for abstractions, which essentially say that if an expression $e$ is strict in some subset $S$ of its free variables, then when we $\lambda$-abstract on one of the free variables $a$, the resulting abstraction $\lambda a.e$ is still strict in $S \setminus \{a\}$. If we can observe termination at higher types, this is simply not true, as the abstraction is a weak head normal form and its evaluation therefore terminates whatever is substituted for the remaining free variables. If we were to fix this problem by insisting that the context $\Gamma'$ in the abstraction rules contained only lifted types, then the resulting analysis would be hopelessly weak. Other strictness analyses based on 'relevance logic style' type systems are similarly fragile.

# 6 Conclusions and further work

We have shown how a simple strictness analysis and its associated optimisations may be expressed together in a single formal system which gives an improved translation of the source language into a variant of Moggi's computational metalanguage. Although the analysis inherent in this translation is rather weak, the associated optimisations go beyond those often considered in the literature in that, in addition to selectively passing arguments by value, they also allow functions to be compiled to expect arguments which are, for example, already evaluated or known to be strict functions. The correctness of the translation was established by a fairly straightforward logical relations argument which connects the domain-theoretic semantics of the source and target languages.

One obvious piece of further work is to implement the system described here. This will involve deciding what we mean by the 'best' translation of a given source term and

then designing an algorithm to find that translation. The right way to do this seems to be to design a non-standard type inference algorithm which assigns $\Lambda_{\mathrm{op}}$ types to $\Lambda_T$ terms, building up the translation $\Lambda_{\mathrm{op}}$ term as a side-effect of unification. The fact that the rules are very far from being syntax directed suggests that a good starting point would be to define a normal form for translation derivations, in which the contraction, (Let) and (Val) rules are only used in certain restricted places. More fundamentally, however, there is considerable scope for further work on improving the translation itself.

The simple system presented here does not in itself provide a practical basis for strictness analysis and optimisation in 'realistic' functional languages. This is because there is no treatment of pairs or other structured datatypes and because the treatment of functions is only valid for languages like PCF, in which termination at higher type is unobservable. It is, however, a natural first step in a line of research which aims to bring analyses and optimisations closer together. The basic idea is that many compiler optimisations can be presented using a target language which has a fixed operational semantics and a type system making the properties of interest explicit. The purpose of analysing the source program is to validate an improved translation into the target language. Roughly speaking, the types of the target language should correspond to the properties used in the static analysis of the source language. In general, however, there will be many properties which are useful in analysis but which we would not wish to make types of the target language, so there will be three kinds of judgement to relate: the typing judgement in the source language, the analysis judgement in the source language and the typing judgement in the target language. (Alternatively, one could imagine a framework in which the default translation is applied first, and an analysis is then applied to the resulting target program in order to justify target-to-target transformations.) The system presented here only involves two kinds of judgement because the only properties which are used in the analysis are those which correspond to target language types. This is why the system is simple, but not particularly powerful. The next step is to develop a better translation of $\Lambda_T$ into $\Lambda_{\mathrm{op}}$ which exploits the results of a more powerful analysis, such as those in [Ben92]. Such analyses can also deal satisfactorily with languages for which termination at higher type *is* observable, so the fragility of the present system which was described at the end of the previous section would be removed. It would also be interesting to look at defining and justifying a 'polyvariant' translation, in which a single source term may be compiled into multiple target terms for use in different contexts. Here again, there are considerable complexities and it would seem advantageous to separate the question of when, in practice, we wish to generate multiple code versions from that of formally defining the space of theoretically valid polyvariant translations.

It is not yet clear how many different analyses and optimisations can be presented using these ideas, but there are already a number of closely related pieces of work. One of these is Leroy's work on boxing optimisations for ML [Ler92], which we have already mentioned, and another is Schellinx's work on decoration strategies for translating conventional logic into linear logic [Sch94]. The latter discusses improving the Girard translation so as to introduce fewer ! types, which is very similar to what we have done in this paper, especially in the light of the close relationship between translations into linear logic and translations into the computational lambda calculus which is explored in [BW95]. There is a mutually beneficial relationship between theory and practice here: not only does the theory suggest practically useful optimisation techniques, but there appears to be scope for applying ideas from static analysis to, for example, the more theoretical study of linear decoration strategies. Related ideas are behind Abramsky's proposal of 'logic-based

program analysis', based on optimising translations of standard functional programs into linear (or similar) term calculi, as a promising research area [Abr90].

## References


[Abr90] S. Abramsky. Computational interpretations of linear logic. Technical Report 90/20, Department of Computing, Imperial College, London, October 1990.

[Amt93] T. Amtoft. Minimal thunkification. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, September 1993.

[BBdP95] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. Technical Report 365, Computer Laboratory, University of Cambridge, May 1995.

[Ben92] P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, Computer Laboratory, University of Cambridge, December 1992.

[BF92] C. A. Baker-Finch. Relevant logic and strictness analysis. In *Workshop on Static Analysis, LaBRI, Bordeaux.* Bigre, 1992.

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

[Bur91] G. L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass., 1991.

[BW95] P. N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. Preprint, 1995.

[Cro92] R. L. Crole. *Programming Metalogics with a Fixpoint Type.* PhD thesis, Computer Laboratory, University of Cambridge, February 1992. Available as Technical Report 247.

[DH93] O. Danvy and J. Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3), 1993.

[HY91] P. Hudak and J. Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, April 1991.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th Symposium on the Principles of Programming Languages*, pages 177–188. ACM, 1992.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asiloomar, CA*, pages 14–23, 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Department of Computer Science, University of Edinburgh, December 1981.

[NN90] H. R. Nielson and F. Nielson. Context information for lazy code generation. In *LISP and Functional Programming*, June 1990.

[PJL91] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th International Symposium on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Sch94] H. Schellinx. *The Noble Art of Linear Decorating.* PhD thesis, University of Amsterdam, 1994.

[Wan93] M. Wand. Specifying the correctness of binding-time analysis. In *ACM Symposium on Principles of Programming Languages*, pages 137–143. ACM, January 1993.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[Wri92] D. A. Wright. *Reduction Types and Intensionality in the Lambda-calculus.* PhD thesis, University of Tasmania, 1992.