Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages

Sam Lindley

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2005

Abstract

This thesis presents a critical analysis of *normalisation by evaluation* as a technique for speeding up compilation of typed functional programming languages. Our investigation focuses on the SML.NET compiler and its typed intermediate language MIL. We implement and measure the performance of normalisation by evaluation for MIL across a range of benchmarks. Taking a different approach, we also implement and measure the performance of a graph-based *shrinking reductions* algorithm for SML.NET.

MIL is based on Moggi's computational metalanguage. As a stepping stone to normalisation by evaluation, we investigate strong normalisation of the computational metalanguage by introducing an extension of Girard-Tait reducibility. Inspired by previous work on local state and parametric polymorphism, we define reducibility for *continuations* and more generally reducibility for *frame stacks*. First we prove strong normalistion for the computational metalanguage. Then we extend that proof to include features of MIL such as sums and exceptions.

Taking an incremental approach, we construct a collection of increasingly sophisticated normalisation by evaluation algorithms, culminating in a range of normalisation algorithms for MIL. Congruence rules and α-rules are captured by a compositional parameterised semantics. *Defunctionalisation* is used to eliminate η-rules. Normalisation by evaluation for the computational metalanguage is introduced using a monadic semantics. Variants in which the monadic effects are made explicit, using either state or control operators, are also considered.

Previous implementations of normalisation by evaluation with sums have relied on continuation-passing-syle or control operators. We present a new algorithm which instead uses a single reference cell and a zipper structure. This suggests a possible alternative way of implementing Filinski's *monadic reflection* operations.

In order to obtain benchmark results without having to take into account all of the features of MIL, we implement two different techniques for eliding language constructs. The first is not semantics-preserving, but is effective for assessing the efficiency of normalisation by evaluation algorithms. The second is semantics-preserving, but less flexible. In common with many intermediate languages, but unlike the computational metalanguage, MIL requires all non-atomic values to be named. We use either control operators or state to ensure each non-atomic value is named.

We assess our normalisation by evaluation algorithms by comparing them with a spectrum of progressively more optimised, rewriting-based normalisation algorithms. The SML.NET front-end is used to generate MIL code from ML programs, including the SML.NET compiler itself. Each algorithm is then applied to the generated MIL code. Normalisation by evaluation always performs faster than the most naïve algorithms — often by orders of magnitude. Some of the algorithms are slightly faster than normalisation by evaluation. Closer inspection reveals that these algorithms are in fact defunctionalised versions of normalisation by evaluation algorithms.

Our normalisation by evaluation algorithms perform unrestricted inlining of functions. Unrestricted inlining can lead to a super-exponential blow-up in the size of target code with respect to the source. Furthermore, the worst-case complexity of compilation with unrestricted inlining is non-elementary in the size of the source code. SML.NET alleviates both problems by using a restricted form of normalisation based on Appel and Jim's *shrinking reductions*. The original algorithm is quadratic in the worst case. Using a graph-based representation for terms we implement a compositional linear algorithm. This speeds up the time taken to perform shrinking reductions by up to a factor of fourteen, which leads to an improvement of up to forty percent in total compile time.

Acknowledgements

Many people have contributed to my well-being in the time that I have spent writing my thesis. I would like to thank them all. Here, I shall just mention those who had a direct impact on my thesis.

First, and foremost, I would like to thank my supervisor, Ian Stark. He provided me with invaluable guidance throughout my PhD. His insightful comments and unwavering enthusiasm provided much inspiration.

My second supervisor, Stephen Gilmore, gave helpful comments on earlier drafts of this thesis. I am particularly grateful to him for encouraging me to attend IFL ’04, despite the fact that my thesis deadline was fast approaching. The change of scene helped clarify my thoughts.

I would like to thank Hayo Thielecke and Phil Wadler for agreeing to be my examiners.

I would like to thank Olivier Danvy for hosting me at BRICS for three months. My visit was funded by a Marie Curie Fellowship. In this time I learnt a great deal about normalisation by evaluation, and had many fruitful discussions with Olivier and his students.

I would like to thank Nick Benton for hosting me as a postgraduate intern at Microsoft Research Cambridge for three months. Nick Benton, Andrew Kennedy and Claudio Russo helped me understand the workings of SML.NET, making my implementation work much easier.

I would like to thank Jon Cook for many lunchtime conversations, walks up Blackford Hill, and reading an earlier draft of this thesis.

Last, but not least, I would like to thank my father, Richard Lindley, for reading through this thesis, despite not understanding any of the technical content.

The first three years of my PhD were funded by EPSRC. My visit to Microsoft Research Cambridge helped fund the fourth year of my PhD.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 3 is based on joint work in collaboration with Ian Stark. A paper [LS05] will be presented at TLCA '05. Much of the work for Chapter 7 was conducted whilst visiting Microsoft Research Cambridge from November 2003 to February 2004. A paper [BKLR05], in collaboration with Nick Benton, Andrew Kennedy and Claudio Russo, was presented at IFL '04.

(*Sam Lindley*)

For Shabana

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we investigate *normalisation by evaluation* (NBE) and assess its use as a technique for speeding up the compilation of typed functional programming languages. Focusing on the SML.NET compiler, we implement normalisation by evaluation for SML.NET and compare its performance with that of other normalisation algorithms. Following a different path, closer to that of the existing compiler, we also implement and assess the performance of a graph-based *shrinking reductions* algorithm for SML.NET.

With the development of increasingly powerful hardware, the speed of compilation is constantly improving. However, as hardware becomes more powerful compilers take advantage of this extra power. Increasingly, compilers for typed functional programming languages use *typed intermediate languages*. These offer many advantages over untyped intermediate languages, but typically lead to longer compile times. Some compilers also use *whole program compilation*. This enables more aggressive optimisations to be performed, but means the program must be recompiled entirely every time a change is made, even if this change affects only a single module.

Typed intermediate languages have recently become popular [Mor95, TMC$^+$96, Sha97a, SLM98, Sha97b, JM97, PCHS00, BK00]. The main idea of this line of work is to preserve type information throughout compilation. This offers a number of advantages over untyped compilation. In particular, the type information allows new type-directed program transformations to be performed. As a further benefit, the type information can be used to validate code. For instance, type information has been used to direct the generation of provably secure mobile code [Nec00]. Being able to type-check intermediate code is also useful for debugging the compiler itself. There are, of course, disadvantages of keeping type information available. If one is not careful about how types are implemented, then concrete representations of types can become very large, and this can be expensive, both in terms of time and space.

The main advantage of whole program compilation is that it allows more optimisation to be performed than separate compilation does. In particular, more optimisations can take place across module boundaries, and certain optimisations such as monomorphisation become more feasible than with separate compilation. The disadvantage is the length of time taken to recompile the program when a small change has been made to a single module. A separate compiler would just recompile the relevant module, but a whole program compiler has to recompile the entire program.

The technique of *normalisation by evaluation* has been studied in many settings. In particular, it has been suggested that normalisation by evaluation is an efficient way of performing normalisation [BS91, BES98, BES03], but until now it has not been applied to compilation. The SML.NET compiler, which compiles Standard ML source code to .NET bytecode, spends a significant proportion of time normalising terms in its typed intermediate language MIL. Motivated by this, we investigate the hypothesis that normalisation by evaluation is both fast (compared with rewriting-based normalisation algorithms), and viable as a technique for the implementation of compilers for typed functional programming languages.

1.1 SML.NET

SML.NET [smla, BKR, BKR04] is a compiler for the strict functional programming language Standard ML [MTHM97, GR04]. It is based on MLj [mlj, BKR98, BK99] which is an ML to Java bytecode compiler. It takes a Standard ML program and produces .NET bytecode as output. The important features of SML.NET are:

- a typed intermediate language MIL
- whole program compilation

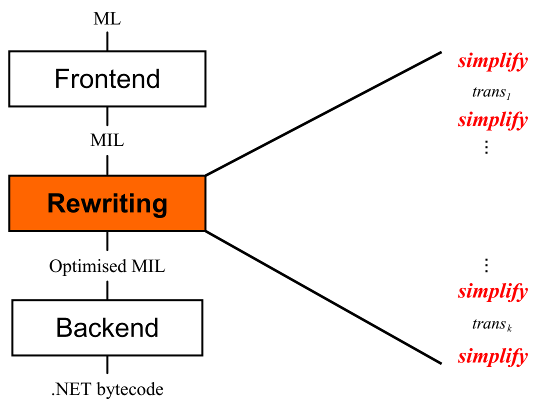

Figure 1.1: The phases of the SML.NET compiler

- interoperability with other .NET programs and libraries

The interoperability extensions are a key part of SML.NET, but they are not relevant to the concerns of this thesis. Typically they form only a small part of a program and SML.NET does not try to optimise them, so they have little impact on compilation time.

The structure of SML.NET is illustrated in Figure 1.1. The frontend takes a Standard ML program as input and outputs a MIL term representing the whole program. The rewriting phase performs rewrite operations on the MIL term, outputting another MIL term. Finally, the backend converts this MIL term into .NET bytecode.

Most of the compile time is accounted for by the rewriting phase. The rewriting phase is divided into a series of transformations on the MIL term. One of these, *simplify*, is invoked many times. In fact, the majority of the time in the rewriting phase is spent in the *simplify* transformation. The *simplify* transformation applies rewrite rules similar to those of the λ-calculus [Bar84] and the computational metalanguage [Mog91, HD94]. It has been identified as a bottleneck in the SML.NET compilation process.

1.2 MIL and the computational metalanguage

The computational metalanguage arose out of Moggi's seminal work on using *monads* to model notions of computation [Mog91, Mog89]. It can be characterised as an extension of simply-typed λ-calculus with computation types. The type system is extended, such that side-effecting computations which return a value of type A are assigned the computation type TA. The computational metalanguage captures the essence of computation without committing to any particular monad. Many λ-calculus techniques can easily be adapted to the computational metalanguage. Some typical examples of monads are: the lifting monad (which captures partiality), the state monad, the exceptions monad and the continuation monad [Mog91].

MIL is based on Moggi's computational metalanguage. It incorporates a type and effect system, which supports effect-based optimisations in addition to the generic ones [BK00]. The monad used models partiality, exceptions and operations on ML references.

1.3 Normalisation by evaluation

There are a number of different ways in which *normalisation* can be defined. However, the essence of normalisation is that it defines a procedure for obtaining a *canonical* version of a term or program which we call its *normal form*. We call a function from terms to normal forms a *normalisation function*. Given an appropriate definition of normal form, optimising a program, or performing transformations such as *simplify* can be seen as an instance of normalisation.

The motivation behind normalisation by evaluation is the idea that *semantics* is crucial to the characterisation of normal forms. One would like programs with the same meaning to have the same normal form. If this is the case, and we have a function for computing the meaning of a program, an *evaluation function* $[\![\cdot]\!]$, then all we need is a suitable *reification function* $\downarrow$, which *reifies* a semantic object to its corresponding normal form. We then obtain a normalisation function:

$$norm(e) = \downarrow [\![e]\!] \tag{1.1}$$

Equation (1.1) captures the essence of normalisation by evaluation. Of course, we have to be careful about how we define the semantics. The semantics needs to contain enough syntactic information in order to be able to reconstruct a term (using an appropriate reification function).

Conventional rewriting-based normalisation is defined syntactically via a collection of reduction rules. These are applied repeatedly until no more rules are applicable. The resulting term is said to be in normal form. Normalisation by evaluation offers an alternative approach to normalisation, with a greater emphasis on semantics.

1.3.1 Residualising semantics versus runtime semantics

We distinguish between two kinds of denotational semantics. The *residualising semantics* is the semantics used for normalisation by evaluation. In the context of compilation, the residualising semantics identifies terms which compile to the same target code. In contrast, the *runtime semantics* captures the run-time behaviour of a program. Two programs have the same runtime semantics if they are behaviourally equivalent. In any programming language which supports general recursion it is not possible to decide whether two arbitrary programs have the same runtime semantics, or even whether a given program terminates (the halting problem).

Ideally, the residualising semantics should agree with the runtime semantics, in the sense that all terms identified in the residualising semantics are also identified in the runtime semantics. In this case we say that any normalisation by evaluation algorithm arising from the residualising semantics is *semantics-preserving*. For the purpose of defining normalisation by evaluation algorithms it is unnecessary to consider the runtime semantics. Thus, we shall usually omit it.

1.4 Contributions

The key contributions of this thesis are the following:

- Application of normalisation by evaluation to compiler optimisation.

- Evaluation of the performance of normalisation by evaluation algorithms in comparison with rewriting-based normalisation algorithms. The results show that the performance of normalisation by evaluation is competitive with with that of the most optimised rewriting-based normalisation algorithms.

- A variant of the reducibility method for proving strong normalisation using continuations, or more generally frame stacks. These methods were applied to the computational metalanguage with extensions including sums and exceptions.

- Variants of normalisation by evaluation:
  - for the computational metalanguage
  - with η-contraction
  - with sums using state instead of first-class continuations
  - for a compiler intermediate language MIL

- Obtaining normalisation by evaluation algorithms by program transformation.

- Evaluation of the performance of a shrinking reductions algorithm for a production compiler using a ‘one-pass’ imperative algorithm.

1.5 Literature overview

In this section we give a brief overview of existing work that is relevant to the thesis.

1.5.1 Lambda calculus

The λ-calculus [Chu41] lays the foundation for functional programming languages. Barendregt gives a comprehensive account of the untyped λ-calculus in [Bar84]. Hankin’s book [Han94] provides a lighter introduction to λ-calculi including some typed variants. Barendregt [Bar92] gives a more comprehensive introduction to typed λ-calculi. He emphasises the difference between Curry-style and Church-style typing, and introduces higher order typed λ-calculi via pure type systems and the λ-cube.

Girard [Gir72] and Reynolds [Rey74] independently discovered the second order λ-calculus.

De Bruijn [dB70] and Howard [How80] independently clarified the relationship between logic and typed λ-calculi — the so-called *Curry-Howard isomorphism*. Girard et al.'s book [GLT89] explains the correspondence and gives an account of typed λ-calculi from a proof-theoretic perspective.

1.5.2 Strong normalisation and reducibility

Strong normalisation for the simply-typed λ-calculus (or rather its proof-theoretic counterpart) using a reducibility argument was first proved by Tait [Tai67]. Prawitz adapted the proof to include sums [Pra71]. Making "splendid use of impredicativity" [Gal90] the reducibility method was extended by Girard [Gir72] to System F and $F\omega$. Subsequently Girard's proof has been adapted to other higher-order calculi such as the Calculus of Constructions [Coq90]. The impredicative reducibility techniques can also be adapted to calculi with positive inductive types. Abel and Altenkirch [AA00] give a predicative strong normalisation proof for the simply-typed λ-calculus extended with strictly positive recursive types. Gallier [Gal90] gives an overview of reducibility proofs, with an emphasis on strong normalisation for higher-order λ-calculi.

1.5.3 The computational metalanguage and monads

Moggi introduced the computational metalanguage [Mog91, Mog89] as the internal language of a cartesian closed category with a *strong monad*. First Hatcliff and Danvy [HD94], and then Sabry and Wadler [SW97] explored the relationship between the computational metalanguage and continuation-passing style (CPS).

Wadler [Wad90] identified a connection between monads and list comprehensions, and showed showed how monads can be used to capture side-effecting computations in pure functional programming languages. Peyton Jones and Wadler [PW93] extended this work in order to perform I/O in non-strict functional programming languages. The Haskell [PJ03] programming language uses monads to manage side-effects in this way.

Wadler and Thiemann [WT03] showed that monads can be used to describe effect-typing systems. Independently, Tolmach [Tol98] and Benton and Kennedy [BK00] implemented typed-intermediate languages for ML compilers based on the computational metalanguage, and monadic effects.

Filinski [Fil94, Fil96, Fil99a] showed that the continuation monad can be used to simulate all other *definable* monads. Furthermore he used this result to implement *monadic reflection* operations, in the SML/NJ [smlb] compiler, for arbitrary definable monads. The monadic reflection operations allow one to convert back and forth between representations of effects as behaviour and as data. Benton [Ben04] applied Filinski's technique in order to define monadic embedded interpreters in ML. Filinski used monadic reflection to obtain an extensional CPS transform [Fil01a].

Benton, Hughes and Moggi give a survey of monads from a variety of perspectives [BHM02].

1.5.4 Normalisation by evaluation

Although, the phrase "normalisation by evaluation" was not coined until much later, Martin-Löf [ML75] used normalisation by evaluation in his work on intuitionistic type-theory, as a way of proving normalisation.

Berger and Schwichtenberg [BS91] defined an "inverse of the evaluation functional" (a reification function) for simply-typed λ-calculus. This gave the standard normalisation by evaluation algorithm for simply-typed λ-calculus.

Berger [Ber93] showed how to extract a normalisation by evaluation algorithm from a proof of strong normalisation.

Berger et al. [BES98, BES03] used normalisation by evaluation to speed up the MINLOG theorem prover. They give a domain theoretic semantics for normalisation by evaluation for the simply-typed λ-calculus extended with constants and a class of rewrite rules involving those constants.

Meanwhile Altenkirch et al. [AHS95] gave a categorical account of normalisation by evaluation for simply-typed λ-calculus using a "twisted gluing" construct. Cubric et al. [CDS98] investigated normalisation by evaluation for simply-typed λ-calculus using the Yoneda embedding. Subsequently Fiore [Fio02] analysed normalisation by evaluation for simply-typed λ-calculus via "extensional normalisation" using the standard (not twisted) categorical gluing construct.

Mogensen [Mog99] gave a normalisation by evaluation algorithm for untyped λ-calculus, and an implementation in Scheme [KCE98]. Aehlig and Joachimski [AJ04] studied normalisation by evaluation for untyped λ-calculus using term-rewriting techniques. Filinski and Rohde [FR04] gave a semantic account of normalisation by evaluation for untyped λ-calculus using minimal invariants.

Filinski [Fil01b] introduced normalisation by evaluation for the computational λ-calculus, using layered monads [Fil99a] for formalising name generation and for collecting bindings. He extended his algorithm to handle products and sums, and outlined how to prove correctness using a Kripke logical relation.

Altenkirch et al. [ADHS01] described normalisation by evaluation for the simply-typed λ-calculus extended with unit, products and sums. They used normalisation by evaluation to obtain a completeness result for almost bicartesian closed categories.

Coquand and Dybjer [CD97] investigated normalisation by evaluation in intuitionistic model theory. They gave a normalisation by evaluation algorithm for combinatory logic.

Altenkirch et al. [AHS96] gave a normalisation by evaluation algorithm for a polymorphic version of combinatory logic. They extracted the algorithm from a categorical model. Subsequently they extended this work to System F [AHS97].

Vestergaard [Ves] used a syntactic approach to investigate normalisation by evaluation for System F.

Beylin and Dybjer [BD95] constructed a normalisation by evaluation algorithm for the free monoid using a free monoidal category.

1.5.5 Type-directed partial evaluation

Type-directed partial evaluation (TDPE) arises as a special case of normalisation by evaluation in which the residualising semantics coincides with the runtime semantics of a programming language. The native evaluator for the programming language can be used to implement the semantics. The reification function acts as a decompiler. Furthermore, because the residualising semantics coincides with the runtime semantics, arbitrary code can be run before calling reify, which is where the partial evaluation arises. One advantage of TDPE over other variants of normalisation by evaluation is the fact that it can use a native evaluator. Thus, if the evaluator is already optimised then it gives rise to efficient normalisation algorithms. We do not use TDPE, as we shall need to use non-standard semantics. Also, our primary concern is normalisation rather than partial evaluation. However, much of the normalisation by evaluation and TDPE literature overlaps.

Danvy [Dan96] discovered type-directed partial evaluation (TDPE) independently of normalisation by evaluation. Filinski gave a semantic analysis and correctness proof for call-by-name [Fil99b] and for call-by-value [Fil01b, DF02] TDPE. Filinski and Yang [Yan99] implemented TDPE in ML using the native ML evaluator and a clever encoding of types. Rose [Ros] implemented TDPE in Haskell using the native evaluator and type classes. Danvy's lecture notes provide a good introduction to TDPE [Dan98].

Balat et al. [BCF04] gave a categorical treatment of normalisation by evaluation for the simply-typed λ-calculus extended with unit, products and sums. Their notion of normalisation closely follows that of Altenkirch et al. [ADHS01], but the analysis follows that of Fiore [Fio02]. They also gave an implementation of TDPE for the simply-typed λ-calculus extended with unit, products and sums, using the set / cupto [GRR98] control operators.

The book by Jones et al. [JGS93] provides an introduction to more conventional syntax-directed partial evaluation.

1.5.6 Control operators

Delimited continuations in the form of shift and reset control operators have been used in implementations of type-direct partial evaluation since its inception [Dan96]. They prove to be useful in the implementation of normalisation by evaluation algorithms, often giving a more concise direct-style alternative to continuation-passing style.

Continuation-passing style [Plo75] provides a declarative way of analysing and manipulating control flow. Each function is augmented with an extra continuation parameter representing the rest of the computation. In contrast, first-class control operators such as call/cc, allow control flow to be manipulated in *direct-style*, that is, without every function having a continuation parameter.

The untyped call/cc operator was added to Scheme [CFW86] as a means of capturing the current continuation. It is closely related to Landin's J [Thi98], and Reynolds escape [Rey98].

Duba et al. [DHM91] showed how to assign a typing to call/cc in ML. The call/cc operator is now supported by ML compilers such as SML/NJ and MLton [mlt].

Felleisen and others [Fel88, FWFD88] introduced the control / prompt operators for handling delimited continuations. Rather than capturing the continuation from the start of the program to a given point in a program (as with call/cc), control and prompt allows one to capture the continuation between a *prompt* and a given point.

Danvy and Filinski [DF90, DF92] introduced the shift / reset operators for manipulating delimited continuations. The control / prompt operators were obtained operationally. In contrast, Danvy and Filinski derived shift / reset by iterating the CPS transform twice, thus obtaining a denotational continuation semantics for delimited continuations. Danvy and Filinski also generalised shift / reset further by iterating the CPS transform repeatedly to obtain the *CPS hierarchy*. This gives rise to a hierarchy of delimited continuation operators allowing one to capture continuations from any of a number of prompts up to a given point.

Wadler [Wad94] used monads to model, and as a basis for typing, delimited continuations. In his work on monadic reflection Filinski [Fil94, Fil96] showed that shift and reset can be used to model any *definable* monad. Furthermore, he gave an implementation of shift and reset in SML/NJ using call/cc and a single reference cell. Gasbichler and Sperber [GS02] gave a direct implementation of shift / reset in a modified version of Scheme.

Kameyama and Hasegawa gave a sound and complete axiomatisation of delimited continuations [KH03]. The axioms support equational reasoning in direct-style. Subsequently Kameyama [Kam04a, Kam04b] extended the axiomatisation to the higher level delimited continuations operators arising from the CPS hierarchy.

Gunter et al. [GRR98] gave a generalisation of shift / reset using the set / cupto construction. This allows a hierarchy of delimited continuations to be invoked by naming the start of each delimited continuation. The set operator is much like the reset operator, except a name is associated with a prompt. The cupto operator is much like the shift operator, but rather than capturing the continuation up to the closest enclosing prompt, it captures the continuation up to an arbitrary named prompt.

Recently Shan [cS04] gave an overview of the various control operators for delimited continuations, and showed that "dynamic" operators such as Felleisen's control can be given a continuation-passing semantics using recursive continuations.

1.5.7 Typed intermediate languages

Recently, the use of typed-intermediate languages in compilers for typed functional programming languages has become increasingly widespread.

Shao and Appel [SA95] obtained improvements both in heap usage and in the speed of compiled code, by performing various type-based optimisations on intermediate representations for the SML/NJ compiler.

The goal of FLINT [Sha97b, Sha97a] is to create a common typed intermediate language for compiling typed functional programming languages. The current "working version" (110.48) of the SML/NJ compiler [smlb] uses FLINT as an intermediate language.

Peyton Jones and Meijer [JM97] proposed a typed intermediate language based on Barendregt's λ cube [Bar92].

TAL [MWCG99] (Typed Assembly Language) is a typed low-level language, for performing well-typed optimisation in the backend of a compiler.

The TIL (Typed Intermediate Language) and TILT (TIL Two) compilers [Mor95, TMC$^+$96, PCHS00] were specifically designed with compilation using typed intermediate representations in mind.

The MLton [mlt, CJW00] compiler takes advantage of typed intermediate languages.

Type and effect systems [NNH99] are used for inferring side-effects. If we know that a section of code can have only a certain class of side-effect then this enables all kinds of optimisations to be performed. For example:

- If a computation has no side-effects, then it is safe to duplicate it.

- If a computation has no side-effects, and its result is never used, then it is safe to eliminate it completely.
- Consecutive computations whose only side-effects are reading the store can be reordered.
- If a computation can throw only a certain class of exception, then there is no need for a handler to check for other kinds of exception.

Jouvelot and Gifford [JG89] used a continuation semantics to handle effects. Thielecke [Thi03] extended this line of work to the typed setting, making use of a polymorphic answer type for controlling effects.

A refinement of typed intermediate languages is to combine types and effects. The ML-Kit compiler uses a type and effect system for memory management using regions [TT97, TBE$^+$01]. MIL uses the type system to keep track of effect information. MIL was first used as the typed intermediate language for the MLj compiler [mlj, BKR98, BK99]. Subsequently MLj became SML.NET [smla, BKR, BKR04].

1.5.8 Shrinking reductions

Shrinking reduction is a restricted form of reduction in which terms can only decrease in size. Shrinking reduction is used in the SML/NJ compiler [smlb, App92] as a restriction of usual $\beta\eta$-reduction. Appel and Jim [AJ97] improved the shrinking reduction algorithm used by the SML/NJ compiler. However, the worst case time complexity of the improved algorithm is quadratic. They also described a linear-time imperative algorithm, which they did not implement. Other ML compilers [mlt, mlj, smla] also make use of shrinking reduction phases, which use variants of the quadratic algorithm.

1.6 Structure of this thesis

In Chapter 2 we introduce the concepts and notation required for the rest of the thesis. We present our treatment of λ-calculi, the computational metalanguage and MIL. We then formalise normalisation and normalisation by evaluation, and outline some techniques for proving correctness of normalisation by evaluation algorithms.

In Chapter 3 we present three proofs of strong-normalisation for the computational metalanguage. The first proof uses a translation into simply-typed λ-calculus extended with an additional reduction rule, followed by a combinatorial argument. The second proof extends Girard-Tait reducibility with *continuations* for handling computations. The same technique is then applied to extensions and variations of the computational metalanguage. The final proof extends reducibility with frame stacks — a generalisation of continuations. Reducibility over terms is defined uniformly for all term constructors as a function of reducibility over frame stacks. This technique is the most general, and we demonstrate how it can be used to prove strong normalisation for an extension of the computational metalanguage with sums.

In Chapter 4 we return to our discussion of normalisation by evaluation. Normalisation by evaluation for simply-typed λ-calculus is extended to the computational metalanguage, first using a monadic semantics, and then using explicit side-effects. Starting from the normalisation by evaluation algorithm for simply-typed λ-calculus, we show how to eliminate η-expansion, and also how to incorporate η-reduction using program transformations. A normalisation by evaluation algorithm for λ_{ml*} — a restriction of the computational metalanguage — is extended to handle sums, using first-class control operators. We then present an alternative algorithm using state and a *zipper* structure. Finally we discuss practical issues and outline how to implement normalisation by evaluation in ML, taking advantage of the ML module system.

In Chapter 5 we extend and adapt the normalisation by evaluation algorithms for the computational metalanguage to a version of MIL which we generate using the SML.NET frontend. This provides a platform for benchmarking normalisation by evaluation on realistic examples. Two different modular approaches, which allow features to be added incrementally, are introduced. Then we describe how to implement some of the additional features of MIL.

In Chapter 6 we present performance results for the normalisation by evaluation algorithms of Chapter 5. The normalisation by evaluation algorithms are compared with a spectrum of (increasingly optimised) rewriting-based normalisation algorithms. The MIL terms are generated from actual ML programs by the SML.NET frontend. In deriving the spectrum of normalisation algorithms, we discover an alternative view of normalisation by evaluation— by program transformation.

In Chapter 7 we describe a different approach to normalisation using shrinking reductions A one-pass imperative algorithm using a graph-based representation for terms is presented. We discuss our implementation of the algorithm for SML.NET, as a replacement for the existing algorithm. We present benchmarks, which show a significant improvement over the original version.

In Chapter 8 we conclude and discuss future work.

Chapter 2

Background

In this chapter we set the scene for the rest of the thesis. We introduce some notation and terminology for the metalanguage and object languages. Then we present the untyped λ-calculus. Simple types are added, followed by products, sums and computations. We describe the intermediate language MIL and a useful variant λMIL. Then we define normalisation and introduce normalisation by evaluation as a method for performing normalisation.

2.1 The metalanguage and notation

Functional programming languages such as ML are based on the λ-calculus. In this thesis the object languages we are interested in are primarily variants of the λ-calculus. It is also convenient to use λ-style notation as part of our metalanguage for defining and reasoning about algorithms which manipulate object language terms. To distinguish between the object language and the metalanguage, we use conventional λ-notation for the metalanguage and $\mathsf{sans\text{-}serif}$ type for object language syntax constructors.

In the metalanguage $\lambda x.s$ denotes an anonymous function abstraction, and juxtaposition $f a$ denotes the application of the function f to the argument a. For example, $(\lambda x.x+1)2$ denotes the number 3.

Ultimately we shall use ML to implement the algorithms specified in our metalanguage. An object language can then be defined using ML datatypes. This is discussed further in §4.9.

We assume the usual set theoretic, logical and arithmetic operations are available in the metalanguage. In the metalanguage we identify types with sets. We write $s : A$ to indicate that the metalanguage term s has type A or equivalently that s is a member of the set A. The set-theoretic function space between sets A and B is written $A \to B$. The product of sets $A_1,\ldots,A_k$ is written $A_1 \times \cdots \times A_k$. We also define a singleton set (nullary product) $\mathbf{1}$ with a single element $()$. The disjoint union (or disjoint sum) of sets $A_1,\ldots,A_k$ is written $A_1 + \ldots + A_k$ or $\uplus S$ where $S = \{A_i\}$. Tuples are written using round brackets. In the metalanguage we elide sum injections and use syntax to distinguish the different branches of a sum. For instance, consider the set of metalanguage terms of type $S = E + (A \to B)$. We can let e range over E, and f range over $A \to B$. Then a metalanguage term e of type S is in the left branch and a metalanguage term f of type S is in the right branch. Of course, this would not work if two branches were of the same type, but in this thesis the branches of the sum always have distinct types, so this is not a problem.

We shall overload type constructors for use both in the metalanguage and the object languages. This does not lead to ambiguity as it is clear from context whether a type refers to the metalanguage or the object language. We assume an infinite set of object variables $\mathbf{V}$ ranged over by $x,y,z\ldots$. An object *language* is defined as a set of *terms*, given by a grammar along with a set of typing constraints. In the case of untyped languages, this set is empty. Upper case letters denote types, and lower case letters denote terms. For a term m, we distinguish between the bound variables $bv(m)$ and the free variables $fv(m)$ of m. For each language, the usual capture-avoiding substitution $m[x := n]$ is defined as the term m in which n is substituted for all free occurrences of x, with bound variables renamed appropriately in order to prevent variable capture.

List notation We find it convenient to make use of lists in the metalanguage. We use the following notation:

- $\langle\rangle$ denotes the empty list.

- $\langle x_1,\ldots,x_n\rangle$ denotes the list of length n with elements $x_1,\ldots,x_n$.

- $xs +\!\!+ ys$ denotes the concatenation of the list ys onto the end of the list xs.
- $x :: xs$ denotes the list $\langle x\rangle +\!\!+ xs$.
- $\mathbf{A}$ *list* denotes the set of lists whose elements are taken from the set $\mathbf{A}$.

We also define some basic functions on lists. Throughout the thesis we shall define functions by pattern matching.

$$\begin{aligned} rev\ \langle\rangle &= \langle\rangle \\ rev\ (x :: xs) &= rev(xs) +\!\!+ \langle x\rangle \end{aligned}$$

$$\begin{aligned} & map : (\mathbf{A} \to \mathbf{B}) \to \mathbf{A}\ list \to \mathbf{B}\ list \\ map\ f\ \langle\rangle &= \langle\rangle \\ map\ f\ (x :: xs) &= (f\ x) :: (map\ f\ xs) \end{aligned}$$

$$\begin{aligned} & unzip : (\mathbf{A} \times \mathbf{B})\ list \to (\mathbf{A}\ list \times \mathbf{B}\ list) \\ unzip\ \langle\rangle &= (\langle\rangle, \langle\rangle) \\ unzip\ ((x,y) :: ps) &= (x :: xs', y :: ys') \\ & \quad \text{where } (xs', ys') = unzip\ ps \end{aligned}$$

- $rev\ xs$ reverses the list xs. Of course, in actual implementations this is implemented more efficiently using an accumulator.
- $map\ f\ xs$ applies f to each element of xs to give a new list.
- $unzip\ ps$ takes a list of pairs and returns the corresponding pair of lists.

2.1.1 Relations and calculi

A *binary relation* on a set S is a subset of $S \times S$. A *subrelation* $\mathcal{R}'$ of $\mathcal{R}$ is a subset of $\mathcal{R}$. We often use infix notation $e \mathcal{R} e'$ for binary relation membership $(e, e') \in \mathcal{R}$.

Equivalence relation An equivalence relation is a binary relation $\mathcal{R}$ that is reflexive: $(e,e) \in \mathcal{R}$ for all m, symmetric: $(e,e') \in \mathcal{R}$ iff $(e',e) \in \mathcal{R}$, and transitive: whenever $(e,e') \in \mathcal{R}$ and $(e',e'') \in \mathcal{R}$ we have that $(e,e'') \in \mathcal{R}$.

Calculus We define a *calculus* as an object language along with an equivalence relation $\mathcal{R}$ on terms. We call $\mathcal{R}$ the *convertibility relation*. We are particularly interested in two flavours of calculi, each with a different set of restrictions placed on $\mathcal{R}$: equational calculi and reduction calculi.

2.1.2 Equational calculi

We wish to define *equational calculi* as systems for performing equational reasoning. We would like the convertibility relation $\equiv$ to satisfy the following properties:

- It should include α-conversion. In other words, if two terms differ only in the names of their bound variables then they are convertible. We write $m =_\alpha m'$ if m is α-convertible to m'.
- It should be a *congruence*: whenever $m_1 \equiv m'_1, \ldots, m_n \equiv m'_n$ and C is an n-ary syntax constructor then we have $\mathsf{C}(m_1, \ldots, m_n) \equiv \mathsf{C}(m'_1, \ldots, m'_n)$.
- It should include a collection of *conversion rules*. Whereas the α rules and the congruence property are fixed by syntactic properties of the language, the conversion rules are arbitrary, and embody the core of the convertibility relation.

We define an *equational calculus* as a calculus induced by an object language $\mathcal{L}$ and a collection of conversion rules, whereby $\equiv$ is the least equivalence relation which contains α-conversion, is a congruence and contains the conversion rules.

2.1.3 Reduction calculi

We define a *reduction calculus* as an object language $\mathcal{L}$ and a *reduction relation* $\longrightarrow$. The convertibility relation is given as $\longleftrightarrow_*$ the reflexive, symmetric, transitive closure of $\longrightarrow$.

We specify the reduction relation via a collection of *reduction rules*. These play a similar role to the conversion rules of equational calculi, but can only be applied left-to-right. Often reduction relations will also have a congruence property, although for calculi with η-expansions this must be weakened. The congruence property for reduction calculi is:

> For each syntax constructor C (with arity n), and $1 \leq i \leq n$, whenever $m_i \longrightarrow m_i'$ we have that $\mathsf{C}(m_1, \ldots, m_i, \ldots, m_n) \longrightarrow \mathsf{C}(m_1, \ldots, m_i', \ldots, m_n)$.

The only constraints we impose on the reduction relation are that:

- It includes the reduction rules.
- It respects α-conversion: if $m \longrightarrow n$ and $m =_\alpha m'$ then there exists n' such that $m' \longrightarrow n'$ and $n =_\alpha n'$.
- It is a subrelation of the congruence closure of the reduction rules (that is, the relation given by closing the reduction rules under the congruence property).

Some reduction rules always decrease the size of a term, and similarly others always increase the size of a term. We call a reduction rule which always decreases the size of a term a *contraction*, and a reduction rule which always increases the size of a term an *expansion*. We call a term which matches the left-hand side of a reduction rule a *redex*. We call e' the *reduct* of the reduction $e \longrightarrow e'$. We write $\longleftrightarrow$ for the symmetric closure of $\longrightarrow$ and $\longrightarrow_*$ for the transitive reflexive closure of $\longrightarrow$.

Each reduction calculus has an underlying equational calculus, whose convertibility relation $\equiv$ is the least equivalence relation which contains α-conversion, is a congruence and contains the reduction rules. Note that $\longrightarrow \subseteq \longleftrightarrow_* \subseteq \equiv$.

2.2 λ-calculi

Each object language gives rise to both equational and reduction calculi. We have already noted that each reduction calculus has an underlying equational calculus given by closing the convertibility relation under α- and congruence rules. One can also obtain a reduction calculus from an equational calculus by *directing* the conversion

$$\begin{tikzcd}
\lambda u \arrow[r, "types"] & \lambda \arrow[d, "\mathbf{1},\times,+"'] \arrow[dr, "T"] & & & \\
& \lambda^{+\mathbf{1}} & \lambda_{ml} \arrow[r] & \lambda_{ml*} \arrow[r] \arrow[d] & \lambda_c \\
& & & \lambda\mathrm{MIL} \arrow[r] & \mathrm{MIL}
\end{tikzcd}$$

Figure 2.1: Relationships between λ-calculi

rules, that is, applying them in only one direction. Sometimes there is a choice. For instance, η-rules can either be instantiated as contractions or as expansions.

In the remainder of this chapter we present equational variants of the calculi of interest. Subsequently, where necessary, corresponding reduction calculi will be obtained by directing conversion rules appropriately.

In general we shall use the convention that lowercase lambda λ (with appropriate annotations) denotes a calculus, and uppercase lambda Λ (with appropriate annotations) denotes the underlying language.

Figure 2.1 illustrates the relationships between the different λ-calculi covered in this chapter. Arrows with a vertical component indicate extensions. Horizontal arrows indicate variations.

2.2.1 The untyped λ-calculus (λu)

The syntax of λ-terms is as follows:

$$m, n ::= x \mid \mathsf{lam}(x, m) \mid \mathsf{app}(m, n)$$

Terms are variables, abstractions or applications. Λu denotes the set of untyped λ-terms. The conversion rules are:

$$\begin{array}{lll}
(\to.\beta) & \mathsf{app}(\mathsf{lam}(x, m), n) \equiv m[x := n] & \\
(\to.\eta) & \mathsf{lam}(x, \mathsf{app}(m, x)) \equiv m, & \text{if } x \notin fv(m)
\end{array}$$

2.2.2 The simply-typed λ-calculus ($\lambda^{\rightarrow}$)

We present our typed calculi by first specifying the syntax of types and *pre-terms*, then giving typing rules which act as constraints on pre-terms to give the set of terms. We further constrain the convertibility relation such that terms are convertible only if they have the same type. We assume a global assignment from variable names to types Γ, such that $\Gamma(x)$ is uniquely defined for each variable $x \in \mathbf{V}$.

The syntax of types and pre-terms is as follows:

$$\begin{array}{ll} \text{types} & A, B ::= O \mid A \rightarrow B \\ \text{pre-terms} & m, n ::= x^A \mid \mathsf{lam}(x^A, m) \mid \mathsf{app}(m, n) \end{array}$$

We assume a single base type O, and the usual function type-constructor. The pre-terms are the same as the terms of the untyped λ-calculus, but with type annotations on the variables. Type annotation are often omitted when they are not necessary. The typing rules are:

$$\frac{}{x^A : A}\Gamma(x) = A \qquad \frac{x : A \quad m : B}{\mathsf{lam}(x^A, m) : A \rightarrow B} \qquad \frac{m : A \rightarrow B \quad n : A}{\mathsf{app}(m, n) : B}$$

lam is the *introduction* syntax constructor for functions — it introduces a term of function type. app is the *elimination* syntax constructor for functions — it *eliminates* a term of function type. In general each type constructor gives rise to an introduction syntax constructor and an elimination syntax constructor. Each elimination syntax constructor gives rise to *elimination contexts* with a hole for the term to be eliminated to be plugged in. Thus function elimination contexts are one-holed contexts of the form:

$$E^{A \rightarrow B}[\ ] ::= \mathsf{app}([\ ], n)$$

In general we shall let $E^A[\ ]$ range over elimination contexts with holes of type A. $E^A[m]$ is the term obtained by plugging m in $E^A[\ ]$. $\Lambda^{\rightarrow}$ denotes the set of simply-typed λ-terms.

Church versus Curry We have chosen a Church-style [Bar92] presentation because it meshes well with our ML implementations, and the MIL language has Church-style type annotations. Most of our development would work equally well in a Curry-style [Bar92]. In the Curry-style one works with untyped, but typeable, terms. Types are assigned via typing contexts. In the Church-style one works with explicitly typed terms.

The conversion rules are the same as for the untyped case (modulo type annotations):

$$\begin{array}{ll} (\to.\beta) & \mathsf{app}(\mathsf{lam}(x,m),n) \equiv m[x:=n] \\ (\to.\eta) & \mathsf{lam}(x,\mathsf{app}(m,x)) \equiv m, \qquad \text{if } x \notin fv(m) \end{array}$$

Notice that the left-hand-side of $\to.\beta$ contains a $\to$-introduction term inside a $\to$-elimination term — in other words an elimination context with an introduction term plugged in the hole. In general β-rules follow this pattern. η-rules are not quite so easy to characterise. In general they capture some notion of *extensionality*. In this case $\to.\eta$ ensures that if $\mathsf{app}(f,m) \equiv \mathsf{app}(f',m)$ for all m, then it must be the case that $f \equiv f'$.

2.2.3 Products and sums

It is natural to extend typed λ-calculi with products and sums. The introduction for products is a pair, and the elimination is a projection. The introduction for sums is an injection, and the elimination is a case split. Generally we use binary products and sums, and sometimes a nullary product — the unit — as well. The syntax of the simply-typed λ-calculus with unit, binary products and sums appears in Figure 2.2. Elimination contexts for products and sums are given by:

$$\begin{aligned} E^{A\times B}[\ ] &::= \mathsf{proj}_1([\ ]) \mid \mathsf{proj}_2([\ ]) \\ E^{A+B}[\ ] &::= \mathsf{case}\ [\ ]\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \end{aligned}$$

Note that there is no elimination for unit as the nullary tuple $*$ has no projections.

The typing rules are those of $\lambda^{\to}$ plus the additional rules for unit, products and sums shown in Figure 2.3.

$\Lambda^{+\mathbf{1}}$ denotes the set of terms of this language. Λ^{+} denotes the subset of $\Lambda^{+\mathbf{1}}$ with unit type removed. $\Lambda^{\times}$ denotes the subset with sum and unit type removed, and $\Lambda^{\times\mathbf{1}}$ the subset with sum type removed.

types	$A, B ::= O \mid A \to B \mid \mathbf{1} \mid A \times B \mid A + B$
pre-terms	$m, n ::= x^A \mid *$
	$\mid \mathsf{lam}(x^A, m) \mid \mathsf{app}(m, n)$
	$\mid \mathsf{pair}(m, n) \mid \mathsf{proj}_1(m) \mid \mathsf{proj}_2(m)$
	$\mid \mathsf{inj}_1(m) \mid \mathsf{inj}_2(m) \mid \mathsf{case}\ m\ \mathsf{of}\ (x_1^A \Rightarrow n_1 \mid x_2^B \Rightarrow n_2)$

Figure 2.2: Syntax of $\lambda^{+\mathbf{1}}$

Unit

$$\frac{}{* : \mathbf{1}}$$

Products

$$\frac{m : A \quad n : B}{\mathsf{pair}(m, n) : A \times B}$$

$$\frac{m : A_1 \times A_2}{\mathsf{proj}_1(m) : A_1} \qquad \frac{m : A_1 \times A_2}{\mathsf{proj}_2(m) : A_2}$$

Sums

$$\frac{m : A}{\mathsf{inj}_1(m) : A + B} \qquad \frac{m : B}{\mathsf{inj}_2(m) : A + B}$$

$$\frac{x_1 : A_1 \quad x_2 : A_2 \quad m : A_1 + A_2 \quad n_1 : B \quad n_2 : B}{\mathsf{case}\ m\ \mathsf{of}\ (x_1^{A_1} \Rightarrow n_1 \mid x_2^{A_2} \Rightarrow n_2) : B}$$

Figure 2.3: Typing rules for **1**, $\times$ and $+$

There are additional conversion rules for the new constructions. The conversion rules for products are straightforward:

$$\begin{array}{rr}
(\times.\beta 1) & \mathsf{proj}_1(\mathsf{pair}(m_1, m_2)) \equiv m_1 \\
(\times.\beta 2) & \mathsf{proj}_2(\mathsf{pair}(m_1, m_2)) \equiv m_2 \\
(\times.\eta) & \mathsf{pair}(\mathsf{proj}_1(m), \mathsf{proj}_2(m)) \equiv m
\end{array}$$

These rules can easily be generalised for n-ary products. Then we can use the fact that unit is a nullary product to obtain the rules for unit. The n-ary counterpart of a pair is a tuple $\mathsf{tuple}(m_1, \ldots, m_n)$, and the n-ary counterpart of a binary projection is an n-ary projection $\mathsf{proj}_i(m)(1 \leq i \leq n)$. The generalised β-rule is:

$$\mathsf{proj}_i(\mathsf{tuple}(m_1, \ldots, m_i, \ldots m_n)) \equiv m_i \tag{2.1}$$

When $n = 0$ this is vacuous. Thus there is no β-rule for unit. The generalised η-rule is:

$$\mathsf{tuple}(\mathsf{proj}_1(m), \ldots, \mathsf{proj}_n(m)) \equiv m \tag{2.2}$$

Setting $n = 0$ gives:

$$(\mathbf{1}.\eta) \qquad\qquad * \equiv m$$

If reading this from left-to-right, we may be concerned as to the origin of m. In principle m could be any term of type unit. However, we only ever apply it from right-to-left. Following others [JG95] we call the right-to-left application of $\mathbf{1}.\eta$ an η-expansion, even though the size of the term does not increase. We justify this by noting that $\mathbf{1}.\eta$ is a special case of Equation (2.2), where $n = 0$. For $n > 0$ applying Equation (2.2) right-to-left does increase the size of the term. One can also derive $\mathbf{1}.\eta$-expansion from categorical considerations [JG95].

Remark In general we call the left-to-right application of an η-rule *contraction*, and the right-to-left application *expansion*. Note that, if in doubt, the left hand side of an η-rule (or β-rule) can always be identified as such, as it must contain an introduction term for the relevant type-constructor.

The conversion rules for sums are as follows:

$$\begin{array}{rl}
(+.\beta 1) & \mathsf{case}\ \mathsf{inj}_1(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \equiv n_1[x_1 := m] \\
(+.\beta 2) & \mathsf{case}\ \mathsf{inj}_2(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \equiv n_2[x_2 := m] \\
(+.\eta) & \mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{inj}_1(x_1) \mid x_2 \Rightarrow \mathsf{inj}_2(x_2)) \equiv m \\
(+.\cdot.\mathrm{CC}) & E^A[\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)] \equiv \mathsf{case}\ m\ \ \mathsf{of}\ \ \begin{array}{l} x_1 \Rightarrow E^A[n_1] \\ \mid\ \ x_2 \Rightarrow E^A[n_2] \end{array}
\end{array}$$

where $A = B \cdot C$ and $\cdot$ is one of $\rightarrow, \times, +$. The $+.\cdot.$CC rules are *commuting conversions*. Applying a commuting conversion does not change the essential structure of a term, although, as in this case, it may duplicate or share contexts. In general commuting conversions arise when the form of elimination terms for a given type constructor includes a subterm whose type is independent of the term being eliminated — in other words, the elimination includes an auxiliary term. For instance, eliminations for sums have the form: $\mathsf{case}\ m\ \mathsf{of}\ (x_1^{A_1} \Rightarrow n_1 \mid x_2^{A_2} \Rightarrow n_2)$. The term being eliminated m has type $A_1 + A_2$, but n_1, n_2 have type B, where B is any type, so they are auxiliary to the elimination. The commuting conversions allow contexts to be moved in or out of the auxiliary terms.

Note that replacing the η-rule $+.\eta$ and the commuting conversion rules $+.\cdot.$CC with the generalised η-rule:

$$\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n[z := \mathsf{inj}_1(x_1)] \mid x_2 \Rightarrow n[z := \mathsf{inj}_2(x_2)]) \equiv n[z := m] \tag{2.3}$$

leaves the convertibility relation unchanged. However, we usually find it more convenient to use the former rules.

Binary sums can be generalised to n-ary sums in a similar way to the way in which binary products can be generalised to n-ary products. In §2.2.4 we discuss unary sums.

2.2.4 The computational metalanguage (λ_{ml})

The computational metalanguage extends the simply-typed λ-calculus with computation types and corresponding syntax constructors. The syntax of types and pre-terms is as follows:

types	$A, B ::= O \mid A \to B \mid TA$
pre-terms	$m, n ::= x^A \mid \mathsf{lam}(x^A, m) \mid \mathsf{app}(m, n) \mid \mathsf{val}(m) \mid \mathsf{let}\ x^A \Leftarrow m\ \mathsf{in}\ n$

The T type constructor is for computation types. The type TA is the type of a computation which returns a value of type A. The introduction $\mathsf{val}(m)$ is just the trivial computation which returns m, and the elimination $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n$ is the composite computation which first performs the computation m and then binds the result to the variable x in the computation n. The typing rules are as follows:

$$\frac{}{x^A : A}\Gamma(x) = A \qquad \frac{m : B}{\mathsf{lam}(x^A, m) : A \to B} \qquad \frac{m : A \to B \quad n : A}{\mathsf{app}(m, n) : B}$$
$$\frac{m : A}{\mathsf{val}(m) : TA} \qquad \frac{m : TA \quad n : TB}{\mathsf{let}\ x^A \Leftarrow m\ \mathsf{in}\ n : TB}$$

A term of the computational metalanguage is a pre-term which is typeable by the above rules. Λml denotes the set of computational metalanguage terms. A *computation* is a term of computation type. A *value* is a term of non-computation type.

We have three additional conversion rules in addition to those for the simply-typed λ-calculus:

$$\begin{array}{rc}
(T.\beta) & \mathsf{let}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n \equiv n[x := m] \\
(T.\eta) & \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{val}(x) \equiv m \\
(T.T.\mathrm{CC}) & \mathsf{let}\ y \Leftarrow (\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n)\ \mathsf{in}\ p \equiv \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p, \quad \text{if } x \notin fv(p)
\end{array}$$

Note that adding unit, products and sums does not cause any complications. We can easily add them in the same manner as we did for the simply-typed λ-calculus.

Unary sums Syntactically the computational metalanguage can be seen as a restriction of the simply-typed λ-calculus with unary sums.

$$\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n \simeq \mathsf{case}\ m\ \mathsf{of}\ (x \Rightarrow n)$$
$$\mathsf{val}(m) \simeq \mathsf{inj}(m)$$

$$
\begin{aligned}
\mathcal{E}'_v(O) &= O \\
\mathcal{E}'_v(A \to B) &= (\mathcal{E}'_v(A) \to \mathcal{E}_v(B)) \\
\\
\mathcal{E}_v(A) &= T(\mathcal{E}'_v(A)) \\
\\
\mathcal{E}_v(x) &= \mathsf{val}(x) \\
\mathcal{E}_v(\mathsf{lam}(x,m)) &= \mathsf{val}(\mathsf{lam}(x,\mathcal{E}_v(m))) \\
\mathcal{E}_v(\mathsf{app}(m,n)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}_v(m)\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow \mathcal{E}_v(n)\ \mathsf{in}\ \mathsf{app}(x,y) \\
&\quad \text{where } x,y \text{ are fresh}
\end{aligned}
$$

Figure 2.4: Call-by-value embedding of $\lambda^{\to}$ into λ_{ml}

The restriction is that let $x \Leftarrow m$ in n must have computation type, whereas the unary sum construction case m of $(x \Rightarrow n)$ can have any type. This in turn restricts the commuting conversions, such that the computational metalanguage has only one commuting conversion $T.T.$CC, whereas the unary case gives rise to one commuting conversion for each type constructor. Filinski [Fil96] has considered a generalised let which has similar properties to unary case.

Related calculi It is common to work with a restricted form of λ_{ml} in which all functions must take values and return computations, thus having type $A \to TB$, and the computation constructor can be applied only to values. Following Sabry and Wadler [SW97] we call this calculus λ_{ml*}. We write $\Lambda ml*$ for the set of all λ_{ml*}-terms. Unlike Sabry and Wadler, we regard both values and computations — rather than just computations — as terms.

λ_{ml*} contains the call-by-value embedding [HD94] of simply-typed λ-calculus into the computational metalanguage. The restriction of function arguments to value types embodies the call-by-value nature of functions. Note that *pure* functions of type $A \to B$ (where B is of value type), *call-by-name* functions of type $TA \to TB$, and *metacomputations* of type $T(TA)$ are all disallowed by λ_{ml*}. The call-by-value embedding function $\mathcal{E}_v$ is defined on simply-typed λ-calculus types and terms. It appears in Figure 2.4. The embedding function $\mathcal{E}_v$ maps simple types to λ_{ml*} computation types, and $\lambda^{\rightarrow}$ terms to λ_{ml*} computation terms. The auxiliary function $\mathcal{E}'_v$ maps simple types to λ_{ml*} value types.

It turns out that the terms of λ_{ml*} are so restricted that we can dispense with the val syntax constructor and computation types, replacing them by simple syntactic separation of values v from terms m. This leaves us with a variant of $\Lambda^{\rightarrow}$ extended with the let-construction, and we have a subset λ_{c*} of Moggi's computational λ-calculus λ_c [Mog89]. Sabry and Wadler discuss in detail the correspondences between λ_{ml}, λ_{ml*}, λ_{c*} and λ_c [SW97].

2.3 MIL

MIL is the *monadic intermediate language* used by the SML.NET compiler. It is essentially an extension of λ_{ml*} restricted so that non-atomic values must be named. We also consider an unrestricted variant which we call λMIL. We distinguish between concrete implementations of MIL and λMIL, and simplified versions we use for exposition. MIL as implemented in the SML.NET compiler contains various features which are not relevant to most of this thesis. We now present simplified MIL and simplified λMIL. Value types are defined by:

$$A, B ::= X \mid Int \mid A\ ref \mid \mathbf{1} \mid A \rightarrow T_{\varepsilon}(B) \mid A \times B \mid A + B \mid \mu X.A$$

Unit, functions, products and sums are covered in §2.2. The integer type *Int* is a base type. A *reference cell* has type *A ref*, where A is the type of the contents of the cell. X ranges over type variables for recursive types and μ is the type constructor for recursive types. $\mu X.A$ is the recursive type in which X is bound to $\mu X.A$ inside A. Recursive types allow datatypes such as lists to be defined. For instance:

$$\mu X.\mathbf{1} + Int \times X$$

is the type of lists of integers.

We will assume that type variables do not occur free. The distinction between *negative* and *positive* occurrences of type variables will be important when we come to consider normalisation with recursive types in Chapter 5. One way of characterising negative and positive occurrences is as follows:

- A free occurrence of a type variable is said to be *positive* in a type A iff it is on the left hand side of an even number of function types.
- A free occurrence of a type variable is said to be *negative* in a type A iff it is on the right hand side of an odd number of function types.

Thus a type A can include both positive and negative occurrences of the free type variable X. More concretely recursive predicates *isNegative* and *isPositive* are defined in Figure 2.5. The predicate *isNegative*(X, A) holds if the type variable X occurs negatively in the type A. The predicate *isPositive*(X, A) holds if the type variable X occurs positively in A.

A computation type $T_\varepsilon(A)$ is parameterised by a finite set of effects ε and a value type A, the return type. Effects range over subsets of $\{\perp, r, w, a\} \uplus \mathbb{E}$, where $\mathbb{E}$ is the set of exceptions. The effects are divergence ($\perp$), reading from a reference cell (r), writing to a reference cell (w), allocating a new reference cell (a) and raising an exception $E \in \mathbb{E}$. Inclusion on sets of effects induces a subtyping relation:

$$\frac{}{A \leq A} \qquad \frac{\varepsilon \subseteq \varepsilon' \quad A \leq A'}{T_\varepsilon(A) \leq T_{\varepsilon'}(A')} \qquad \frac{A \leq A'}{\mu X.A \leq \mu X.A'}$$

$$\frac{A \leq A' \quad B \leq B'}{A \times B \leq A' \times B'} \qquad \frac{A \leq A' \quad B \leq B'}{A + B \leq A' + B'} \qquad \frac{A' \leq A \quad C \leq C'}{A \to C \leq A' \to C'}$$

where A, B range over value types and C over computation types.

The pre-terms of λMIL appear in Figure 2.6. The introduction and elimination constructors for recursive types are, respectively, fold and unfold. Elimination contexts for recursive types are given by:

$$E^{\mu X.A}[\ ] ::= \mathsf{unfold}([\ ])$$

Rather than the usual handle construct, MIL uses Benton and Kennedy's *exceptional syntax* [BK01]. The introduction term raise(E) raises the exception E, as normal. However, the try construct is a generalisation of let. The elimination term

$$
\begin{aligned}
\mathit{isNegative}(X,Y) &= \text{false} \\
\mathit{isNegative}(X,\mathit{Int}) &= \text{false} \\
\mathit{isNegative}(X,A\ \mathit{ref}) &= \mathit{isNegative}(X,A) \\
\mathit{isNegative}(X,\mathbf{1}) &= \text{false} \\
\mathit{isNegative}(X,A \to T_\varepsilon(B)) &= \mathit{isPositive}(X,A) \vee \mathit{isNegative}(X,B) \\
\mathit{isNegative}(X,A \times B) &= \mathit{isNegative}(X,A) \vee \mathit{isNegative}(X,B) \\
\mathit{isNegative}(X,A + B) &= \mathit{isNegative}(X,A) \vee \mathit{isNegative}(X,B) \\
\mathit{isNegative}(X,\mu Y.A) &= \text{false}, && \text{if } X = Y \\
\mathit{isNegative}(X,\mu Y.A) &= \mathit{isNegative}(A), && \text{otherwise} \\[1em]
\mathit{isPositive}(X,Y) &= \text{true}, && \text{if } X = Y \\
\mathit{isPositive}(X,Y) &= \text{false}, && \text{otherwise} \\
\mathit{isPositive}(X,\mathit{Int}) &= \text{false} \\
\mathit{isPositive}(X,A\ \mathit{ref}) &= \mathit{isPositive}(X,A) \\
\mathit{isPositive}(X,\mathbf{1}) &= \text{false} \\
\mathit{isPositive}(X,A \to T_\varepsilon(B)) &= \mathit{isNegative}(X,A) \vee \mathit{isPositive}(X,B) \\
\mathit{isPositive}(X,A \times B) &= \mathit{isPositive}(X,A) \vee \mathit{isPositive}(X,B) \\
\mathit{isPositive}(X,A + B) &= \mathit{isPositive}(X,A) \vee \mathit{isPositive}(X,B) \\
\mathit{isPositive}(X,\mu Y.A) &= \text{false}, && \text{if } X = Y \\
\mathit{isPositive}(X,\mu Y.A) &= \mathit{isPositive}(A), && \text{otherwise}
\end{aligned}
$$

Figure 2.5: Predicates for negative and positive occurrences of type variables

Atoms	$a, b ::= x^A \mid * \mid c^A$
Values	$v, w ::= a$
	$\mid \mathsf{lam}(x^A, m)$
	$\mid \mathsf{pair}(a, b) \mid \mathsf{proj}_1(a) \mid \mathsf{proj}_2(b)$
	$\mid \mathsf{inj}_1(a) \mid \mathsf{inj}_2(a)$
	$\mid \mathsf{fold}_{\mu X.A}(a) \mid \mathsf{unfold}(a)$
Computations	$m, n ::= \mathsf{app}(a, b)$
	$\mid \mathsf{val}(v) \mid \mathsf{raise}(E) \mid \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H$
	$\mid \mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$
	$\mid \mathsf{read}(a) \mid \mathsf{write}(a, b) \mid \mathsf{new}$

Figure 2.6: Pre-terms of MIL

$\mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H$ first performs the computation m. If an exception is not thrown then the result is bound to x, and the computation n is performed. If an exception E is thrown then it is passed to the handler H. Elimination contexts for computations are now given by:

$$E^{T_\varepsilon(A)}[\ ] ::= \mathsf{try}\ x \Leftarrow [\ ]\ \mathsf{in}\ m\ \mathsf{unless}\ H$$

A *handler* H^A is a list of pairs (E, m) associating exceptions with computations of type A. We write $H(E)$ for the first computation in H which is associated with the exception E. Any subsequent pairs associating E to a computation are redundant. If E is not associated with any computation in H, then $H(E)$ denotes $\mathsf{raise}(E)$. A handler H handles an exception E by performing the computation $H(E)$.

Handlers can be composed as follows:

$$H; H' = (map\ (\lambda(E, m).\mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{val}(x)\ \mathsf{unless}\ H')\ H) +\!\!+ H' \qquad (2.4)$$

where *map* is the usual map function as defined in §2.1.

If an exception E is passed to the handler $(H; H')$, then it is first passed to H. If H catches E then $H(E)$ is invoked, but this may raise a further exception which is passed on to H'. If H does not catch E then it is passed on to H'.

The other constructs we have not yet discussed are for managing references.

- new allocates a new reference cell
- $\mathsf{read}(a)$ returns the contents of reference cell a.
- $\mathsf{write}(a,b)$ writes the value b into the reference cell a.

We shall not attempt to perform conversions involving references. Thus, the above syntax constructors can effectively be treated as constants.

We introduce some syntactic sugar:

$$\begin{aligned}
\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n &= \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ \langle\rangle \\
\mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ n &= \mathsf{let}\ x \Leftarrow \mathsf{val}(v)\ \mathsf{in}\ n \\
\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n &= \mathsf{letval}\ f \Leftarrow \mathsf{lam}(x,m)\ \mathsf{in}\ n
\end{aligned}$$

Notice that the only place where non-atomic values can occur is in $\mathsf{val}(v)$. The only way to use a value inside a $\mathsf{val}(v)$ is to bind it to a variable using try. Thus, in MIL, all non-atomic values must be named before being used. The idea is that atoms are small, so can be safely duplicated, whereas non-atomic values are potentially large, so they should be named in case they need to be used more than once.

It is sometimes useful to relax the distinction between atoms and non-atoms. We call the resulting language λMIL. λMIL is a strict extension of λ_{ml*}. We use λMIL to define the typing rules and convertibility relation for MIL. The pre-terms of λMIL appear in Figure 2.7.

The typing rules for λMIL appear in Figure 2.8. MIL has the same typing rules (with appropriate restrictions on values). ΛMIL denotes the set of terms of λMIL terms, and **MIL** denotes the set of terms of MIL.

Now we present the conversion rules for λMIL. The β- and η-conversion rules appear in Figure 2.9. Because of the restriction that case terms must have computation type, there are only two commuting conversions. These appear in Figure 2.10. The first ($T.T$.CC) is for a try inside a try. The second ($+.T$.CC) is for a case inside a try. The convertibility relation for MIL is simply the restriction of the convertibility relation for λMIL to MIL terms.

Values	$v, w ::= x^A \mid * \mid c^A$
	$\mid \mathsf{lam}(x^A, m)$
	$\mid \mathsf{pair}(v, w) \mid \mathsf{proj}_1(v) \mid \mathsf{proj}_2(v)$
	$\mid \mathsf{inj}_1(v) \mid \mathsf{inj}_2(v)$
	$\mid \mathsf{fold}_{\mu X.A}(v) \mid \mathsf{unfold}(v)$
Computations	$m, n ::= \mathsf{app}(v, w)$
	$\mid \mathsf{val}(v) \mid \mathsf{raise}(E) \mid \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H$
	$\mid \mathsf{case}\ v\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$
	$\mid \mathsf{read}(v) \mid \mathsf{write}(v, w) \mid \mathsf{new}$

Figure 2.7: Pre-terms of λMIL

For the remainder of the thesis MIL and λMIL will be used to refer to their simplified versions, except where specified otherwise.

2.4 Normalisation

Normalisation is a process which takes a term e and returns an equivalent term e' in a special form called a *normal form*. We define normalisation with respect to a calculus with object language $\mathcal{L}$ and convertibility relation $\equiv$, and a subset of $\mathcal{L}$, the set of *normal forms* $\mathcal{L}$*-nf*.

- *Normalisation* is the process of taking a term e and obtaining another term e' (if such a term exists) such that $e' \in \mathcal{L}\text{-}nf$ and $e \equiv e'$.

- A function $norm{:}\mathcal{L} \to \mathcal{L}\text{-}nf$ is a *normalisation function* if for all $e \in \mathcal{L}$ we have that if $norm(e)$ terminates, then $norm(e) \equiv e$.

- A normalisation function *norm* is *sound* if whenever $e \equiv e'$ then either $norm(e)$ and $norm(e')$ are both undefined (diverge) or $norm(e) = norm(e')$.

Atoms

$$\frac{}{x^A : A}\ \Gamma(x) = A \qquad \frac{}{* : \mathbf{1}} \qquad \frac{}{c^A : A}$$

Functions

$$\frac{x : A \quad m : T_\varepsilon(B)}{\mathsf{lam}(x^A, m) : A \to T_\varepsilon(B)} \qquad \frac{v : A \to T_\varepsilon(B) \quad w : A}{\mathsf{app}(v, w) : T_\varepsilon(B)}$$

Products

$$\frac{v : A \quad w : B}{\mathsf{pair}(v, w) : A \times B} \qquad \frac{v : A_1 \times A_2}{\mathsf{proj}_1(v) : A_1} \qquad \frac{v : A_1 \times A_2}{\mathsf{proj}_2(v) : A_2}$$

Computations

$$\frac{v : A}{\mathsf{val}(v) : T_\emptyset(A)} \qquad \frac{}{\mathsf{raise}(E) : T_{\{E\}}(A)}$$

$$\frac{x : A \quad m : T_\varepsilon(A) \quad n : T_\varepsilon(B) \quad H : T_\varepsilon(B)}{\mathsf{try}\ x^A \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H : T_\varepsilon(B)}$$

Sums

$$\frac{v : A}{\mathsf{inj}_1(v) : A + B} \qquad \frac{v : B}{\mathsf{inj}_2(v) : A + B}$$

$$\frac{x_1 : A_1 \quad x_2 : A_2 \quad v : A_1 + A_2 \quad n_1 : T_\varepsilon(B) \quad n_2 : T_\varepsilon(B)}{\mathsf{case}\ v\ \mathsf{of}\ (x_1^{A_1} \Rightarrow n_1 \mid x_2^{A_2} \Rightarrow n_2) : T_\varepsilon(B)}$$

Recursive types

$$\frac{v : A[X := \mu X.A]}{\mathsf{fold}_{\mu X.A}(v) : \mu X.A} \qquad \frac{v : \mu X.A}{\mathsf{unfold}(v) : A[X := \mu X.A]}$$

References

$$\frac{}{\mathsf{new} : T_{\{a\}}(A\ \mathit{ref})} \qquad \frac{v : A\ \mathit{ref}}{\mathsf{read}(v) : T_{\{r\}}(A)} \qquad \frac{v : A\ \mathit{ref} \quad w : A}{\mathsf{write}(v, w) : T_{\{w\}}(\mathbf{1})}$$

Subtyping

$$\frac{e : A}{e : A'}\ A \le A'$$

Figure 2.8: Typing rules for λMIL

$$
\begin{array}{rc}
(\rightarrow.\beta) & \mathsf{app}(\mathsf{lam}(x,m),v) \equiv m[x := v] \\
(\rightarrow.\eta) & \mathsf{lam}(x,\mathsf{app}(v,x)) \equiv v, \quad \text{if } x \notin fv(v) \\
(\mathbf{1}.\eta) & * \equiv v \\
(\times.\beta 1) & \mathsf{proj}_1(\mathsf{pair}(v_1,v_2)) \equiv v_1 \\
(\times.\beta 2) & \mathsf{proj}_2(\mathsf{pair}(v_1,v_2)) \equiv v_2 \\
(\times.\eta) & \mathsf{pair}(\mathsf{proj}_1(v),\mathsf{proj}_2(v)) \equiv v \\
(T.\beta) & \mathsf{try}\ x \Leftarrow \mathsf{val}(v)\ \mathsf{in}\ m\ \mathsf{unless}\ H \equiv m[x := v] \\
(T.\eta) & \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{val}(x)\ \mathsf{unless}\ \langle\rangle \equiv m \\
(T_{exn}.\beta) & \mathsf{try}\ x \Leftarrow \mathsf{raise}(E)\ \mathsf{in}\ n\ \mathsf{unless}\ H \equiv H(E) \\
(+.\beta 1) & \mathsf{case}\ \mathsf{inj}_1(v)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \equiv n_1[x_1 := v] \\
(+.\beta 2) & \mathsf{case}\ \mathsf{inj}_2(v)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \equiv n_2[x_2 := v] \\
(+.\eta) & \begin{array}{lll} \mathsf{case}\ v & \mathsf{of} & x_1 \Rightarrow \mathsf{val}(\mathsf{inj}_1(x_1)) \equiv \mathsf{val}(v) \\ & \mid & x_2 \Rightarrow \mathsf{val}(\mathsf{inj}_2(x_2)) \end{array} \\
(\mu.\beta) & \mathsf{unfold}(\mathsf{fold}_{\mu X.A}(v)) \equiv v \\
(\mu.\eta) & \mathsf{fold}_{\mu X.A}(\mathsf{unfold}(v)) \equiv v
\end{array}
$$

Figure 2.9: β- and η-rules for λMIL

$(T.T.\mathrm{CC})$

$$
\begin{array}{l}
\mathsf{try}\ y \Leftarrow (\mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H)\ \mathsf{in}\ p\ \mathsf{unless}\ H' \equiv \\
\qquad\qquad \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ (\mathsf{try}\ y \Leftarrow n\ \mathsf{in}\ p\ \mathsf{unless}\ H')\ \mathsf{unless}\ H;H', \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{if } x \notin fv(p) \text{ and } x \notin fv(H'(E)) \text{ for any } E \in \mathbb{E}
\end{array}
$$

$(+.T.\mathrm{CC})$

$$
\begin{array}{l}
\mathsf{try}\ y \Leftarrow (\mathsf{case}\ v\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))\ \mathsf{in}\ m\ \mathsf{unless}\ H \equiv \\
\qquad\qquad\qquad\qquad \begin{array}{lll} \mathsf{case}\ v & \mathsf{of} & x_1 \Rightarrow \mathsf{try}\ y \Leftarrow n_1\ \mathsf{in}\ m\ \mathsf{unless}\ H \\ & \mid & x_2 \Rightarrow \mathsf{try}\ y \Leftarrow n_2\ \mathsf{in}\ m\ \mathsf{unless}\ H \end{array}
\end{array}
$$

Figure 2.10: CC-rules for λMIL

- The set of normal forms is *canonical* if whenever $e \in \mathcal{L}\text{-}nf$ and $e \equiv e'$ then either $e' \notin \mathcal{L}\text{-}nf$ or $e = e'$.

We shall distinguish between two different approaches to normalisation: *reduction-based* and *equational*. For normalisation by evaluation we will mainly work with equational normalisation. However, in many cases there is a correspondence between the two notions and it can be useful to move back and forth between them.

Normal forms and α-equivalence In defining canonical normal forms and sound normalisation functions we will find it convenient to identify terms up to α-equivalence. For instance, we shall assert that $\mathsf{lam}(x, x) = \mathsf{lam}(y, y)$. Note that technically this is not necessary, but it simplifies the presentation. The alternative would be to define a function for providing a canonical renaming of bound variables. By virtue of the fact that we always generate fresh names (deterministically) for bound variables our ML implementations of normalisation functions do give unique normal forms with respect to α-conversion. In the sequel we identify terms up to α-conversion.

2.4.1 Reduction-based normalisation

Reduction-based normalisation applies to a reduction calculus $(\mathcal{L}, \longrightarrow)$, and is probably the more familiar notion of normalisation. The convertibility relation is $\longleftrightarrow_*$. Normal forms are obtained by repeatedly applying directed reduction rules to a term until no more rules are applicable. The set of normal forms is the set of all irreducible terms.

Typically we will obtain a reduction calculus from an equational calculus, and the reduction rules (and a corresponding reduction relation) from a convertibility relation by directing the conversion rules. In the case of λ-calculi we always read the β-rules and commuting conversions from left-to-right. The η-rules can either be read left-to-right, as contractions, or (with extra side-conditions) from right-to-left, as expansions.

For example, the reduction rules for the simply-typed λ-calculus with β-reduction and η-contraction are given by reading both the conversion rules from left-to-right:

$$
\begin{array}{lrcll}
(\to.\beta) & \mathsf{app}(\mathsf{lam}(x,m),n) & \longrightarrow & m[x:=n] & \\
(\to.\eta) & \mathsf{lam}(x,\mathsf{app}(m,x)) & \longrightarrow & m, & \text{if } x \notin fv(m)
\end{array}
$$

Definition 2.1.

- *A term is* weakly normalising *if it has a normal form.*
- *A* reduction sequence *starting from a term m_1 is a (possibly infinite) sequence of terms $m_1, m_2, \ldots$ such that for all i we have that $m_i \longrightarrow m_{i+1}$.*
- *A term is* strongly normalising *if all reduction sequences starting from that term are finite. In other words, every term has a normal form.*
- *A term m is* confluent *if whenever $m \longrightarrow_* m'$ and $m \longrightarrow_* m''$ there exists m''' such that $m' \longrightarrow_* m'''$ and $m'' \longrightarrow_* m'''$.*

It is natural to extend the notions of normalisation and confluence to entire calculi. A reduction calculus is weakly normalising / strongly normalising / confluent if all terms are weakly normalising / strongly normalising / confluent. If a reduction calculus is weakly normalising then normal forms always exist, and hence it gives rise to a total normalisation function. If a reduction calculus is confluent then normal forms are unique, and hence it gives rise to a sound normalisation function.

In Chapter 3 we shall give some general techniques for proving strong normalisation. As we shall see later, strong normalisation turns out to be a useful stepping stone to proving correctness of normalisation by evaluation algorithms.

2.4.2 Equational normalisation

Equational normalisation is simply our general notion of normalisation applied to an equational calculus. There are many possible choices of normal form. Usually we seek a canonical set of normal forms, and correspondingly a sound normalisation function. Normal forms are typically defined syntactically.

Often normal forms correspond with those of a related reduction calculus. Sometimes, however, additional reduction rules must be introduced in addition to those arising directly from the conversion rules.

Remark For some calculi (such as the untyped λ-calculus) it is not possible to compute a normal form for all terms. However, for most of the calculi we are interested in all terms do have well-defined normal forms. Indeed, it would seem a desirable property of a compiler that every source program compiles to a well-defined executable, and consequently one would hope that every intermediate language term is optimised (normalised) to a well-defined target term.

Remark If a reduction calculus is weakly normalising and confluent then it gives rise to an equational calculus with canonical normal forms. Weak normalisation guarantees that normal forms exist, and confluence guarantees that normal forms are unique up to α-conversion. The normal forms of the equational calculus are defined as the normal forms of the reduction calculus (recall that we identify terms up to α-conversion).

Remark Existence and uniqueness of normal forms are important properties for equational calculi. Notice that there is no counterpart to strong normalisation in equational calculi.

Example: consider the simply-typed λ-calculus with β-reduction and η-contraction. This calculus is strongly (and hence weakly) normalising [GLT89] and also confluent [Bar84]. Thus we can define the canonical normal forms of the equational calculus to be the same as those of the reduction calculus.

Example: consider the simply-typed λ-calculus with products and sums where the reduction rules are obtained by reading the conversion rules from left-to-right. This calculus is not confluent:

$$\begin{tikzcd}
& \mathsf{app}(\mathsf{case}\ y\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{inj}_1(x_1) \mid x_2 \Rightarrow \mathsf{inj}_2(x_2)), z) \arrow[dl, "+.\eta"'] \arrow[d, "+.\mathrm{CC}"] \\
\mathsf{app}(y, z) & \mathsf{case}\ y\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{app}(\mathsf{inj}_1(x_1), z) \mid x_2 \Rightarrow \mathsf{app}(\mathsf{inj}_2(x_2), z))
\end{tikzcd}$$

Hence the reduction-based notion of normal form does not coincide with a canonical equational normal form.

Semantic normalisation [Fil01b] is an instance of canonical equational normalisation in which rather than defining the convertibility relation directly it is induced by a denotational semantics. In other words a normal form m' for a term m is defined as a canonical representative of the equivalence class of terms which have the same denotation as m.

2.4.3 The structure of normal forms

It is often useful to describe the structure of normal forms. One way of doing so is by giving a syntax for normal forms. For instance, a standard example of normal forms for the simply-typed λ-calculus with β- and η-conversion is as follows:

$$\begin{array}{ll} \text{Normal forms} & m ::= n^O \mid \mathsf{lam}(x^A, m) \\ \text{Neutral terms} & n^A ::= x^A \mid \mathsf{app}(n^{B\to A}, m) \end{array}$$

where m ranges over normal forms, and n^A over *neutral terms* of type A. Here, normal forms are *long normal forms* [JG95, Hue76]. In general, *long normal forms* are obtained by generating a reduction relation by reading β- and CC-rules from left-to-right, but η-rules from right-to-left (η-expansion).

$$\begin{array}{llcll} (\to.\beta) & \mathsf{app}(\mathsf{lam}(x,m),n) & \longrightarrow & m[x:=n] & \\ (\to.\eta) & m & \longrightarrow & \mathsf{lam}(x,\mathsf{app}(m,x)), & \text{where } x \notin fv(m) \end{array}$$

The η-rule also has to be further restricted in order to prevent infinite expansion. η-expansion is only applicable to m if the resulting term contains no new β-redexes. The reduction relation is restricted such that it always satisfies the congruence property, except where doing so violates the side-condition on creating new β-redexes. For instance:

$$f^{O\to O} \longrightarrow \mathsf{lam}(x^O, \mathsf{app}(f,x))$$

but we do not have:

$$\mathsf{app}(f^{O\to O}, m) \longrightarrow \mathsf{app}(\mathsf{lam}(x^O, \mathsf{app}(f,x)), m)$$

because $\mathsf{app}(\mathsf{lam}(x^O,\mathsf{app}(f,x)),m)$ is a β-redex.

2.4.4 Normal forms from reductions

If one starts with a reduction calculus, then it is often possible to construct a syntax for normal forms by starting with the syntax for terms and gradually restricting this according to the reduction rules. For example, consider untyped λ-calculus with β-reduction, but no η-rule. The syntax of terms is:

$$m, n ::= x \mid \mathsf{app}(n, m) \mid \mathsf{lam}(x, m)$$

Now, a normal form cannot contain a β-redex, so we must somehow restrict the form of applications. Specifically, an abstraction cannot be applied. We introduce a new syntactic category of *neutral terms* for terms which can be applied. Again these cannot contain β-redexes:

Normal forms	$m ::= x \mid \mathsf{app}(n, m) \mid \mathsf{lam}(x, m)$
Neutral terms	$n ::= x \mid \mathsf{app}(n, m)$

where n ranges over normal forms and m ranges over neutral terms. We can simplify this, as neutral terms coincide with a subset of normal forms, to give:

Normal forms	$m ::= n \mid \mathsf{lam}(x, m)$
Neutral terms	$n ::= x \mid \mathsf{app}(n, m)$

Naturally this technique can be used for equational calculi too, by directing the conversion rules.

Unfortunately, it is not always possible to give a neat syntactic characterisation of normal forms. For instance, if we add η-contraction then this results in a restriction on the form of normal forms occurring inside abstractions. Specifically, $\mathsf{lam}(x, m)$ is only a valid normal form if m is not of the form $\mathsf{app}(n, x)$ where $x \notin fv(n)$. We can explicitly state this as a restriction on the syntax, but it is not possible to express directly in the BNF. Similarly it is not entirely straightforward to construct a normalisation by evaluation algorithm which performs η-contraction. Nevertheless, this technique does provide a general way to obtain the structure of normal forms from a reduction relation.

2.5 Normalisation by evaluation

We present normalisation by evaluation as a method for performing canonical equational normalisation. However, the normal forms produced by canonical equational normalisation will often coincide with those produced by reduction-based normalisation, so we shall sometimes use reduction-based techniques in order to reason about normalisation by evaluation.

Normalisation by evaluation gives a sound normalisation function *norm* for a language $\mathcal{L}$ with convertibility relation $\equiv$ and normal forms $\mathcal{L}\text{-}nf$:

$$\begin{aligned} norm &: \mathcal{L} \to \mathcal{L}\text{-}nf \\ norm(e) &= \downarrow [\![e]\!] \end{aligned}$$

$[\![\cdot]\!]$ is called the *residualising semantics*. $[\![\mathcal{L}]\!]$ denotes a *semantic domain* in which terms are interpreted, and $[\![e]\!]$ the meaning of the term e, where $[\![e]\!] : [\![\mathcal{L}]\!]$. The *reification function* $\downarrow : [\![\mathcal{L}]\!] \to \mathcal{L}\text{-}nf$ ('reify') takes semantic objects to normal forms.

We require that the following properties hold:

$$\textbf{(soundness)} \quad e \equiv e' \implies [\![e]\!] = [\![e']\!] \tag{2.5}$$

$$\textbf{(consistency)} \quad e \equiv norm(e) \tag{2.6}$$

Given the typing constraint for *norm*, consistency ensures that *norm* is a normalisation function. The soundness property stated here, is soundness of the residualising semantics. Note that soundness of the residualising semantics implies soundness of *norm*. Hence, (2.5) and (2.6) guarantee that *norm* is a sound normalisation function.

2.5.1 Example: the free monoid

Our first example of normalisation by evaluation, the free monoid, is rather simple. It does not relate directly to the main body of this thesis, but it does provide a basic illustration of normalisation by evaluation.

Source language

$$(Exp) \quad m, n ::= 1 \mid x \mid m \times n$$

Terms are built up from a distinguished unit, variables and multiplication, which represent the monoid operations

Conversion rules

$$\begin{array}{rc}
\text{(associativity)} & (a \times b) \times c \equiv a \times (b \times c) \\
\text{(right identity)} & a \times 1 \equiv a \\
\text{(left identity)} & 1 \times a \equiv a
\end{array}$$

The conversion rules are simply the monoid laws.

Normal form

$$(\mathit{Exp\text{-}nf}) \quad m ::= 1 \mid x \times m$$

Normal forms result from eliminating units and associating brackets to the right. An alternative perspective is to view terms as binary trees, and normalisation as the process of flattening these to lists. Indeed, a basic list semantics gives rise to a normalisation by evaluation algorithm.

Residualising semantics

$$\begin{aligned}
[\![\mathit{Exp}]\!] &= \mathbf{V}\ \mathit{list} \\
[\![1]\!] &= \langle\rangle \\
[\![x]\!] &= \langle x \rangle \\
[\![m \times n]\!] &= [\![m]\!] +\!\!+ [\![n]\!]
\end{aligned}$$

Reification

$$\begin{aligned}
\downarrow : [\![\mathit{Exp}]\!] &\to \mathit{Exp\text{-}nf} \\
\downarrow \langle\rangle &= 1 \\
\downarrow (x :: xs) &= x \times (\downarrow xs)
\end{aligned}$$

Normalisation

$$\mathit{norm}(m) = \downarrow [\![m]\!]$$

Unit is interpreted as the empty list, a variable is interpreted as a singleton list, and multiplication is interpreted as list concatenation. The reification function simply witnesses the isomorphism between normal forms and lists.

2.5.2 Typed calculi and environments

For typed calculi it is convenient to take the usual approach of partitioning the semantic domain according to type. $\mathcal{L}_A$ denotes the subset of $\mathcal{L}$ consisting of terms of type A (similarly $\mathcal{L}\text{-}nf_A$ denotes the normal form terms of type A). $[\![\, \mathcal{L}_A \,]\!]$, or more concisely $[\![\, A \,]\!]$, denotes the semantic domain used to interpret terms of type A. Correspondingly we define a type-indexed version of the reification function $\downarrow^A : [\![\, A \,]\!] \to \mathcal{L}\text{-}nf_A$. The normalisation function becomes:

$$\begin{aligned} norm &: \mathcal{L} \to \mathcal{L}\text{-}nf \\ norm(e^A) &= \downarrow^A [\![\, e \,]\!] \end{aligned}$$

In the case of λ-calculi we shall use an environment semantics in which the semantics is parameterised by an environment ρ, mapping variable names to semantic objects (with an appropriate type restriction in the case of typed calculi). The term e in environment ρ is interpreted as $[\![\, e \,]\!]_\rho$. For the typed case this gives a normalisation function of the form:

$$\begin{aligned} norm &: \mathcal{L} \to \mathcal{L}\text{-}nf \\ norm(e^A) &= \downarrow^A [\![\, e \,]\!]_\uparrow \end{aligned}$$

where $\uparrow$ is the initial environment. $\uparrow$ might be the empty environment ε, which makes sense if we want only to normalise closed terms. If we want to handle open terms as well, then we can define $\uparrow$ to map each variable to a corresponding semantic object which somehow encapsulates the name and type of the variable. This is the approach we shall usually take.

2.5.3 Example: simply-typed λ-calculus and long normal forms

Our second example of normalisation by evaluation is the one which has been most studied in the literature [BS91, BES98, BES03, Ber93, Fio02, AHS95]. This normalisation by evaluation algorithm forms the basis for most of the normalisation by evaluation algorithms explored in this thesis. It illustrates normalisation by evaluation for typed-calculi using an environment semantics.

Recall from §2.4.3 that long normal forms are given by:

Normal forms	$m ::= n^O \mid \mathsf{lam}(x^A, m)$
Neutral terms	$n^A ::= x^A \mid \mathsf{app}(n^{B\to A}, m)$

We write $\Lambda^{\to}$-*nf* for the set of normal forms and $\Lambda^{\to}$-*ne* for the set of neutral terms. In general we shall use the suffix -*nf* for the set of normal forms of a language, and -*ne* for the set of neutral terms.

We choose a set-theoretic semantics in which the base type is interpreted by the set of normal forms of base type and functions are interpreted by the set-theoretic function space:

$$\begin{aligned}
[\![O]\!] &= \Lambda^{\to}\text{-}nf_O \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!]
\end{aligned}$$

The semantics of terms is standard:

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\mathsf{lam}(x, m)]\!]_\rho &= \lambda s. [\![m]\!]_{\rho[x \mapsto s]} \\
[\![\mathsf{app}(m, n)]\!]_\rho &= [\![m]\!]_\rho([\![n]\!]_\rho)
\end{aligned}$$

The type-indexed $\downarrow$ is defined mutually recursively with another type-indexed function $\uparrow$ (‘reflect’) which *reflects* a neutral term in the semantics.

$$\begin{aligned}
\downarrow^A &: [\![A]\!] \to \Lambda^{\to}\text{-}nf_A \\
\uparrow^A &: \Lambda^{\to}\text{-}ne_A \to [\![A]\!]
\end{aligned}$$

$$\begin{aligned}
\downarrow^O e &= e \\
\downarrow^{A\to B} f &= \mathsf{lam}(x, \downarrow^B (f(\uparrow^A x))), \quad x \text{ fresh}
\end{aligned}$$

$$\begin{aligned}
\uparrow^O e &= e \\
\uparrow^{A\to B} e &= \lambda s. \uparrow^B (\mathsf{app}(e, (\downarrow^A s)))
\end{aligned}$$

To normalise a term e^A we have:

$$norm(e^A) = \downarrow^A [\![e]\!]_{\uparrow}$$

where $\uparrow$ denotes the environment mapping x^A to $\uparrow^A x$.

2.5.4 Reflection

Perhaps the easiest way to motivate the role of reflection is to begin by considering the function $\uparrow$ restricted to variables. In order to define the semantics on open terms we need a suitable initial environment $\uparrow$, which maps a variable x^A to a semantic object $\uparrow^A x$ of type A. For the purposes of normalisation by evaluation the x must be somehow encoded in $\uparrow^A x$ in such a way that it is possible to extract x — in particular reifying it should return the normal form of the variable.[1]

$$\downarrow^A \uparrow^A x = norm(x^A) \tag{2.7}$$

It turns out that reflection is also necessary in the definition of $\downarrow$. In particular, in order to reify a function it must be applied to the semantic representation of a fresh variable. Furthermore, in the definition of $\uparrow$ it is necessary to obtain a semantic representation of other neutral terms apart from variables. Thus $\uparrow$ is defined over all neutral terms, and for any neutral term n^A:

$$\downarrow^A \uparrow^A n = norm(n^A) \tag{2.8}$$

This property follows from the characteristic property of reflection:

$$[\![n^A]\!]_{\uparrow} = \uparrow^A n \tag{2.9}$$

Reflecting a neutral term is the same as interpreting it in the initial environment.

[1]Notice that in general the long normal form of a neutral term can be obtained just using η-expansion. This is because a neutral term contains no β- or CC-redexes, and η-expansion is not allowed to introduce new β- or CC-redexes.

Remark A common mistake is to ask the question of whether normalisation can be performed as reify composed with reflect. Clearly this could not work in general, as reflect is only defined on neutral terms.

Remark A standard normalisation by evaluation method for dealing with constants in λ-calculi, which we will use later, is to interpret constants by reflecting them in the semantics, calling $\uparrow$ at the appropriate type. In effect this amounts to treating such constants as if they are neutral or *uninterpreted*.

2.5.5 Example: untyped λ-calculus and β-normal form

We consider the untyped λ-calculus with just the β-rule. Normal forms are given by:

$$\begin{array}{rl} (\Lambda u\text{-}nf) & m ::= n \mid \mathsf{lam}(x,m) \\ (\Lambda u\text{-}ne) & n ::= x \mid \mathsf{app}(n,m) \end{array}$$

Terms are interpreted in a domain given by a recursive domain equation. Informally, a term is either interpreted as raw syntactic material, or as a function from semantic objects to semantic objects.

$$\begin{array}{rlll} [\![\Lambda u]\!] & \cong \Lambda u\text{-}ne + ([\![\Lambda u]\!] \to [\![\Lambda u]\!]) & & \\ [\![x]\!]_\rho & = \rho(x) & & \\ [\![\mathsf{lam}(x,e)]\!]_\rho & = \lambda s.[\![e]\!]_{\rho[x\mapsto s]} & & \\ [\![\mathsf{app}(e_1,e_2)]\!]_\rho & = f([\![e_2]\!]_\rho), & & \text{if } f = [\![e_1]\!]_\rho \text{ is a function} \\ & = \mathsf{app}(m, \downarrow([\![e_2]\!]_\rho)), & & \text{if } m = [\![e_1]\!]_\rho \text{ is a term} \end{array}$$

$$\begin{array}{rlll} \downarrow : [\![\Lambda u]\!] \to \Lambda u\text{-}nf & & & \\ \downarrow e = e, & & & \text{if } e \text{ is a term} \\ \downarrow f = \mathsf{lam}(x, \downarrow f(x)), & & (x \text{ fresh}) & \text{if } f \text{ is a function} \end{array}$$

$$norm(e) = \downarrow([\![e]\!]_\uparrow)$$

where $\uparrow x = x$.

We do not give a formal account of recursive domain equations. $\cong$ is read as "is isomorphic to". So $[\![\Lambda u]\!]$ is isomorphic to Λu-*nf* $+ ([\![\Lambda u]\!] \to [\![\Lambda u]\!])$. If we were to give a formal semantics we could take $[\![\Lambda u]\!]$ to be a CPO [Win93], and assume that $S \to S'$ denotes the continuous function space between CPOs. In ML programs we simulate recursive domains using datatypes.

```
datatype term = Var of string
              | Lam of string * term | App of term * term

datatype  sem = Neutral of term | Fun of sem -> sem
```

The datatype `term` represents Λu, and `sem` represents $[\![\Lambda u]\!]$.

2.6 Parameterised semantics and compositionality

Generally we want to give a *compositional* semantics for our equational calculi, in that the semantics of a term is defined in terms of the semantics of its component parts. Observe that any term e can be expressed as an n-ary term constructor C applied to n subterms $e_1 \dots e_n$.

Definition 2.2. *A function f on terms of a language is* compositional *if f can be defined such that for any term $e = \mathsf{C}(e_1, \dots, e_n)$ we have that $f(e)$ is a function of $f(e_1), \dots, f(e_n)$.*

At the very least, compositionality of a *semantics* ensures that it is sound with respect to the congruence rules.

Lemma 2.3. *If $[\![\cdot]\!]$ is compositional, C is an n-ary term constructor, and $[\![e_1]\!] = [\![e'_1]\!], \dots, [\![e_n]\!] = [\![e'_n]\!]$ then $[\![\mathsf{C}(e_1, \dots, e_n)]\!] = [\![\mathsf{C}(e'_1, \dots, e'_n)]\!]$.*

Proof.

$$\begin{aligned}
[\![\mathsf{C}(e_1, \dots, e_n)]\!] &= f([\![e_1]\!], \dots, [\![e_n]\!]), \quad \text{for some function } f \\
&= f([\![e'_1]\!], \dots, [\![e'_n]\!]) \\
&= [\![\mathsf{C}(e'_1, \dots, e'_n)]\!]
\end{aligned}$$

□

We capture compositionality explicitly using the notion of a *parameterised semantics*. A parameterised semantics is specified in terms of a set of parameters: one for each term constructor. Instantiating all of these parameters gives a concrete semantics. Different instantiations give different concrete semantics.

Sometimes certain parameters are left *uninterpreted*. The semantics of an n-ary uninterpreted parameter p is given by defining $p(s_1,\ldots,s_n)$ as an n-tuple $(s_1,\ldots,s_n)$ tagged with the parameter p, which we simply write as $p(s_1,\ldots,s_n)$.

2.6.1 Example: the free monoid

We give a parameterised semantics for the free monoid, with parameters *unit*, *elt* and *prod*:

$$\begin{aligned}
[\![1]\!] &= \mathit{unit} \\
[\![x]\!] &= \mathit{elt}(x) \\
[\![e_1 \times e_2]\!] &= \mathit{prod}([\![e_1]\!], [\![e_2]\!])
\end{aligned}$$

In §2.5.1 this was instantiated with the monoid of lists:

$$[\![\mathit{Exp}]\!] = \mathbf{V}\ \mathit{list}$$

$$\begin{aligned}
\mathit{unit} &= \langle\rangle \\
\mathit{elt}(x) &= \langle x \rangle \\
\mathit{prod}(l_1, l_2) &= l_1 +\!\!+\, l_2
\end{aligned}$$

Another alternative is to use the monoid of functions from terms to terms:

$$[\![\mathit{Exp}]\!] = \mathit{Exp} \to \mathit{Exp}$$

$$\begin{aligned}
\mathit{unit} &= \lambda e.e \\
\mathit{elt}(x) &= \lambda e.x \times e \\
\mathit{prod}(f_1, f_2) &= f_1 \circ f_2
\end{aligned}$$

$$\downarrow f = f(1)$$

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ [\![\mathsf{lam}(x,e)]\!]_\rho &= lam(\lambda s.[\![e]\!]_{\rho[x\mapsto s]}) \\ [\![\mathsf{app}(e_1,e_2)]\!]_\rho &= app([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \end{aligned}$$

Figure 2.11: Parameterised semantics for Λu and $\Lambda^{\rightarrow}$

Beylin and Dybjer [BD95] constructed this algorithm from a free monoidal category. It can also be seen as a version of the list-based algorithm in which lists are encoded as partially applied concatenation functions [Hug86, DN01]. The motivation for representing lists as partially applied concatenation functions, is that concatenation becomes a constant time operation, whereas concatenation on linked lists is linear.

2.6.2 Example: the (untyped / simply-typed) λ-calculus

A parameterised semantics for both Λu and $\Lambda^{\rightarrow}$, with parameters *lam* and *app*, is given in Figure 2.11. Note how the abstraction is interpreted as *lam* applied to a function. As well as covering the congruence rules, this also captures the α-rules — the name of the bound variable is irrelevant. In general bound variables give rise to functions in a parameterised semantics. For instance the parameter *let* can be used to interpret let in the computational metalanguage, where $[\![\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n]\!]_\rho = let([\![m]\!]_\rho, \lambda s.[\![n]\!]_{\rho[x\mapsto s]})$.

Remark By leaving all the parameters *uninterpreted*, we obtain *higher-order abstract syntax* [PE88].

The semantics of §2.5.5 can be easily obtained:

$$\begin{aligned} &lam : [\![\Lambda u]\!] \rightarrow [\![\Lambda u]\!] \\ &app : ([\![\Lambda u]\!] \times [\![\Lambda u]\!]) \rightarrow [\![\Lambda u]\!] \end{aligned}$$

$$\begin{aligned} lam(f) &= f \\ app(f,s) &= f(s), && \text{if } f \text{ is a function} \\ &= \mathsf{app}(f, \downarrow s), && \text{if } f \text{ is a term} \end{aligned}$$

Given that the residualising semantics of the simply-typed λ-calculus from §2.5.3 does not use the type annotations on bound variables, the parameters can also be instantiated to give the semantics of §2.5.3:

$$\begin{aligned} lam &: (\llbracket A \rrbracket \to \llbracket B \rrbracket) \to (\llbracket A \rrbracket \to \llbracket B \rrbracket) \\ app &: ((\llbracket A \rrbracket \to \llbracket B \rrbracket) \times \llbracket A \rrbracket) \to \llbracket B \rrbracket \end{aligned}$$

$$\begin{aligned} lam(f) &= f \\ app(f, s) &= f(s) \end{aligned}$$

In Chapter 4 we will consider other normalisation by evaluation algorithms using the same parameterised semantics. In §4.8 we introduce an alternative parameterised semantics for $\Lambda^{\rightarrow}$, which does make use of the type annotations on bound variables. This allows us to dispense with the type-index on $\downarrow$.

Remark In the above examples we have explicitly specified the types of the instantiated parameters. Usually we shall omit the types, as they are easy to infer.

2.7 Proof techniques for normalisation by evaluation

The focus of this thesis is the application of normalisation by evaluation, rather than providing completely formal correctness proofs. However, it can be illuminating to have some appreciation of the proof techniques one might use. We recall the two properties which are needed for correctness of normalisation by evaluation:

$$\begin{aligned} \textbf{(soundness)} &\quad e \equiv e' \implies \llbracket e \rrbracket = \llbracket e' \rrbracket \\ \textbf{(consistency)} &\quad e \equiv norm(e) \end{aligned}$$

Usually soundness is easy to prove by induction on the proof that $e \equiv e'$. The difficult part tends to be proving consistency.

2.7.1 Direct proof

Sometimes we can prove soundness of the semantics and consistency of *norm* directly without using any special techniques. We illustrate this approach with the free monoid using the list semantics.

Soundness of the semantics is straightforward to prove by induction on the proof that $a \equiv b$. The associative law follows from associativity of concatenation and the identity laws follow from the property that the empty list is the identity for list concatenation. In other words, lists satisfy the monoid laws. Consistency is slightly more difficult to prove. It is proved by induction on the structure of terms using the following lemma:

Lemma 2.4. $\downarrow (\llbracket a \rrbracket +\!\!+ \llbracket b \rrbracket) \equiv (\downarrow \llbracket a \rrbracket) * (\downarrow \llbracket b \rrbracket)$

Proof. Observe that $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$ are just lists $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_m \rangle$ respectively. Then:

$$\begin{aligned}
\downarrow (\llbracket a \rrbracket +\!\!+ \llbracket b \rrbracket) &= \downarrow \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle && \text{(expanding out lists)} \\
&= (a_1 * \cdots * (a_n * (b_1 * \cdots * (b_m * 1)))) && \text{(definition of } \downarrow\text{)} \\
&\equiv (a_1 * \cdots * (a_n * 1)) * (b_1 * \cdots * (b_m * 1)) && \text{(conversion rules)} \\
&= (\downarrow \langle a_1, \ldots, a_n \rangle) * (\downarrow \langle b_1, \ldots, b_m \rangle) && \text{(definition of } \downarrow\text{)} \\
&= (\downarrow \llbracket a \rrbracket) * (\downarrow \llbracket b \rrbracket) && \text{(contracting lists)}
\end{aligned}$$

□

Lemma 2.5 (consistency). $e \equiv norm(e)$

Proof. The only interesting case is multiplication:

$$\begin{aligned}
norm(a * b) &= \downarrow (\llbracket a * b \rrbracket) && \text{(definition of } norm\text{)} \\
&= \downarrow (\llbracket a \rrbracket +\!\!+ \llbracket b \rrbracket) && \text{(definition of } \llbracket \cdot \rrbracket\text{)} \\
&\equiv (\downarrow \llbracket a \rrbracket) * (\downarrow \llbracket b \rrbracket) && \text{(Lemma 2.4)} \\
&= norm(a) * norm(b) && \text{(definition of } norm\text{)} \\
&\equiv a * b && \text{(induction hypothesis)}
\end{aligned}$$

□

2.7.2 Using the existence of normal forms

A direct proof of consistency is often non-trivial. However, if we already know that normal forms exist, then the task becomes much easier. After proving soundness of the semantics, we just need to show that normal forms are preserved by *norm*.

Proposition 2.6. *Suppose* $norm(e) = \downarrow [\![e]\!]$. *If:*

1. *For all* $e, e' \in \mathcal{L}$, *if* $e \equiv e'$ *then* $[\![e]\!] = [\![e']\!]$.
2. *For all* $e \in \mathcal{L}$ *there exists* $e' \in \mathcal{L}\text{-}nf$ *such that* $e \equiv e'$.
3. *For all* $e \in \mathcal{L}\text{-}nf$ *we have* $e = norm(e)$.

then norm is a normalisation by evaluation function.

Proof. 1 is soundness. Now we show consistency. By 2, given any e there exists $e' \in \mathcal{L}\text{-}nf$ with $e \equiv e'$. By 3, $e' = norm(e')$, and hence $e \equiv norm(e')$. But by soundness $[\![e]\!] = [\![e']\!]$, and hence $norm(e) = norm(e')$ by definition of *norm*. Thus $e \equiv norm(e)$. □

We illustrate this technique with simply-typed λ-calculus and the normalisation by evaluation algorithm of §2.5.3. Soundness follows by straightforward induction on the proof that $e \equiv e'$ using the following substitution lemma:

Lemma 2.7. $[\![m]\!]_\rho[x \mapsto [\![n]\!]_\rho] = [\![m[x := n]]\!]_\rho$

Proof. Induction on the structure of m. □

It is standard that every term is convertible to a long normal form [JG95, Hue76]. Hence we just need to show for e^A in normal form that $e^A = norm(e^A)$. The proof is by induction on the structure of terms. There are two cases.

λ-abstractions

$$\begin{aligned}
& norm(\mathsf{lam}(x,m))^{A\to B}) \\
& = \downarrow^{A\to B} [\![\mathsf{lam}(x,m)]\!]_\uparrow && \text{(definition of } norm\text{)} \\
& = \mathsf{lam}(x, \downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})(\uparrow^A x))), \quad x \text{ fresh} && \text{(definition of } \downarrow \text{ and } [\![\cdot]\!]\text{)} \\
& = \mathsf{lam}(x, \downarrow^B [\![m]\!]_{\uparrow[x\mapsto \uparrow^A x]}) && \text{(meta-level } \beta\text{-reduction)} \\
& = \mathsf{lam}(x, \downarrow^B [\![m]\!]_\uparrow) && \text{(definition of } \uparrow\text{)} \\
& = \mathsf{lam}(x, norm(m^B)) && \text{(definition of } norm\text{)} \\
& = \mathsf{lam}(x, m) && \text{(induction hypothesis)}
\end{aligned}$$

Note that it is safe to assume that x is chosen as the fresh variable in the definition of reify, as this is the only place we generate bound variables and this variable will never be used elsewhere.

We could be completely formal about fresh-name generation using a method such as: FreshML / FM set theory [SPG03, GP01], term families / de Bruijn levels / de Bruijn indices [BES98, dB72], or a name generation monad [Fil01b]. However, we chose not to because this would significantly complicate the exposition with boring details.

Neutral terms We write $\mathsf{app}(m_1,\ldots,m_k)$ for $\mathsf{app}(\ldots\mathsf{app}(m_1,m_2),\ldots,m_k)$. Neutral terms of type O have the form $\mathsf{app}(x,m_1,\ldots,m_k)$, where $m_1{:}A_1,\ldots,m_k{:}A_k$ are normal:

$$\begin{array}{ll}
\mathit{norm}((\mathsf{app}(x,m_1,\ldots,m_k))^O) & \\
=\downarrow^O [\![(\mathsf{app}(x,m_1,\ldots,m_k))]\!]_\uparrow & \text{(definition of } \mathit{norm}\text{)} \\
= [\![(\mathsf{app}(x,m_1,\ldots,m_k))]\!]_\uparrow & \text{(definition of } \downarrow\text{)} \\
= [\![x]\!]_\uparrow [\![m_1]\!]_\uparrow \ldots [\![m_k]\!]_\uparrow & \text{(definition of } [\![\cdot]\!]\text{)} \\
= (\uparrow^{A_1\to\cdots\to A_n\to 0} x)[\![m_1]\!]_\uparrow \ldots [\![m_k]\!]_\uparrow & \text{(definition of } [\![x]\!]_\uparrow\text{)} \\
=\uparrow^O \mathsf{app}(x,\downarrow^{A_1} [\![m_1]\!]_\uparrow,\ldots,\downarrow^{A_n} [\![m_k]\!]_\uparrow) & \text{(definition of } \uparrow^{A\to B} \\
 & \text{applied } k \text{ times)} \\
=\uparrow^O (\mathsf{app}(x,m_1,\ldots,m_k)) & \text{(definition of } \mathit{norm} \text{ and} \\
 & \text{induction hypothesis} \\
 & \text{applied to each } m_i\text{)} \\
= \mathsf{app}(x,m_1,\ldots,m_k) & \text{(definition of } \uparrow^O\text{)}
\end{array}$$

Hence *norm* is a normalisation by evaluation function for simply-typed λ-calculus with long normal forms.

Proposition 2.6 gives not only that *norm* is a normalisation by evaluation function, but also that it is total. One would expect that if normal forms exist for all terms, then a normalisation by evaluation algorithm should terminate on all inputs. However, proving termination directly is often non-trivial. Typically we prove existence of normal forms in an equational calculus by appealing to a normalisation proof in a reduction calculus. In effect we lift a termination proof in a reduction calculus up to a proof of termination and correctness of a normalisation by evaluation algorithm. We discuss approaches to normalisation in Chapter 3.

Berger et al. [BS91, BES98, BES03] use the existence of normal forms in order to prove correctness of normalisation by evaluation. Our proof of consistency is essentially the same as theirs. Fiore [Fio02] uses the same idea, but works entirely in a categorical framework.

2.7.3 Other proof techniques

The standard normalisation by evaluation algorithm for obtaining long normal forms for the simply-typed λ-calculus has been proved correct using a variety of different techniques including the following:

- Berger [Ber93] starts with a constructive strong normalisation proof, from which he extracts a normalisation by evaluation algorithm.
- Hofmann [Hof99] uses a logical relation between semantic objects and terms.
- Filinski [Fil99b] uses semantic normalisation and a Kripke logical relation to prove correctness of TDPE — a stronger property than correctness of normalisation by evaluation.
- Vestergaard [Ves01] takes a rewriting-theoretic approach using a two-level λ-calculus [NN92].
- A range of categorical techniques [AHS95, CDS98, Fio02].

Berger and Hofmann's techniques are both connected to the method of §2.7.2. First, strong normalisation implies the existence of normal forms. Second, strong normalisation for simply-typed λ-calculus can be proved using a logical relation [GLT89]. We believe that one can view Hofmann's proof as a transformation of the proof of §2.7.2 in which a logical relations proof that normal forms exist has been inlined.

We shall not rigorously prove our algorithms correct, but we shall use Proposition 2.6 to justify the algorithms in Chapter 4 and Chapter 5.

2.8 Corollaries of normalisation by evaluation

Uniqueness of normal forms By definition a normalisation function *norm* which is sound and consistent defines a unique normal form for an equational calculus.

Confluence If in proving correctness of *norm*, the conversion rules are applied only in one direction then this gives rise to a reduction-calculus with unique normal forms, that is, a confluent reduction-calculus. Hence normalisation by evaluation can be used for proving confluence. Coquand and Dybjer [CD97], and Dybjer and Filinski [DF02] use this idea to prove confluence of a combinatory calculus.

Another way of proving confluence is from strong normalisation, correctness of normalisation by evaluation and preservation of normal forms.

Theorem 2.8. *Let $(\mathcal{L}, \longrightarrow)$ be a reduction calculus. If:*

1. *$(\mathcal{L}, \longrightarrow)$ is strongly normalising with normal forms $\mathcal{L}$-nf.*
2. *$norm{:}\mathcal{L} \to \mathcal{L}\text{-}nf$ is a normalisation by evaluation function for $(\mathcal{L}, \longleftrightarrow_*)$*
3. *For all $e \in \mathcal{L}\text{-}nf$ we have $e = norm(e)$.*

then $(\mathcal{L}, \longrightarrow)$ is confluent.

Proof. Suppose $e \longrightarrow_* m$ and $e \longrightarrow_* n$. By 1, $m \longrightarrow_* m'$ and $n \longrightarrow_* n'$ where $m', n' \in \mathcal{L}\text{-}nf$. By 2, $m' = norm(m')$ and $n' = norm(n')$. But $m' \longleftrightarrow_* n'$, and by soundness of *norm* we have that $norm(m') = norm(n')$. Hence $m' = n'$. □

Completeness A semantics is complete if for any e, e':

$$[\![e]\!] = [\![e']\!] \implies e \equiv e'$$

This follows from soundness and consistency as:

$$[\![e]\!] = [\![e']\!] \implies norm(e) = norm(e') \implies e \equiv e'$$

Altenkirch et al. [ADHS01] used normalisation by evaluation in this way to obtain a completeness proof for a categorical model of $\lambda^{+\mathbf{1}}$.

Decidability One of the primary uses of normalisation is for deciding whether two terms are convertible — just compare their normal forms for syntactic equality. If *norm* is total, that is, $norm(e)$ is defined for all $e \in \mathcal{L}$, then $\equiv$ is clearly decidable.

Complexity Decidability of β- and $\beta\eta$-equality for the simply-typed λ-calculus has worst case time complexity [Vor97, Vor04] given by the non-elementary function:

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}cn$$

where n is the size of the term, and c is some constant. Decidability is trivially reducible to normalisation. Hence normalisation is at least as complex. Consequently we do not try to reason about the asymptotic complexity of our normalisation by evaluation algorithms, but instead take an empirical approach to evaluating the efficiency of normalisation by evaluation as compared with other normalisation algorithms.

Chapter 3

Normalisation for the computational metalanguage

In subsequent chapters we investigate normalisation by evaluation for the computational metalanguage and related calculi. In this chapter we consider reduction-based normalisation for the computational metalanguage. This is interesting in its own right, but also allows us to apply Proposition 2.6. We concentrate on showing that the computational metalanguage is strongly normalising, although in order to apply Proposition 2.6 it is actually sufficient to prove weak normalisation (existence of normal forms).

Another property of reduction calculi one is often interested in is confluence. As illustrated in §2.8, confluence can be obtained as a corollary of normalisation by evaluation, so we do not pursue it further. Other proofs of confluence for the computational metalanguage appear in the literature [BBdP98, BHT97].

Remark Although it can be useful to know that reductions can be applied in any order, confluence (or uniqueness of normal forms) is not essential in compiler implementations. Providing that the target code is semantically equivalent to the source, it does not matter that there may be other possible normal forms.

This chapter is concerned with reduction calculi. For the remainder of the chapter read reduction calculus for calculus. We obtain reduction rules for the computational metalanguage by directing β-, CC- and η-rules from left-to-right. The rules appear in

$$\begin{array}{rrcll}
(\to.\beta) & \mathsf{app}(\mathsf{lam}(x,m),n) & \longrightarrow & m[x:=n] & \\
(\to.\eta) & \mathsf{lam}(x,\mathsf{app}(m,x)) & \longrightarrow & m, & \text{if } x \notin fv(m) \\
(T.\beta) & \mathsf{let}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n & \longrightarrow & n[x:=m] & \\
(T.\eta) & \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{val}(x) & \longrightarrow & m & \\
(T.T.\mathrm{CC}) & \mathsf{let}\ y \Leftarrow (\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n)\ \mathsf{in}\ p & \longrightarrow & \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p, & \text{if } x \notin fv(p)
\end{array}$$

Figure 3.1: Reductions for the computational metalanguage

Figure 3.1.

In our reducibility proofs we will need a *substitutivity* result.

Proposition 3.1 (Substitutivity). *If* $m \longrightarrow m'$ *then* $m[x:=n] \longrightarrow m'[x:=n]$.

Proof. Induction on the derivation of $m \longrightarrow m'$. □

The rest of this chapter is structured as follows. In §3.1 we give an overview of our approach to proving strong normalisation for the computational metalanguage. In §3.2 we outline an alternative proof of strong normalisation for the computational metalanguage by translation into a simpler calculus. In §3.3 we give a strong normalisation proof using reducibility and continuations. In §3.4 we give some variations and extensions of reducibility for continuations. In §3.5 we generalise reducibility over continuations to reducibility over frame stacks. Finally, in §3.6 we discuss some related work.

3.1 Strong normalisation and $(-)^{\top\top}$

We shall prove that the computational metalanguage is strongly normalising. In §3.2 we outline a combinatorial proof using a translation into the simply-typed λ-calculus extended with a commuting conversion corresponding to $T.T.$CC. Similar proofs by translation appear in the literature [BBdP98, BHT97]. This successfully establishes termination, but only by relating λ_{ml} to some other system for which we happen to have a result to hand.

A direct proof of strong normalisation is problematic for the same reason that it is for the simply-typed λ-calculus: one of the reductions for computations performs substitution of one term within another. Thus a reduction step may make a term grow larger, and create subterms not present before. The consequence is that straightforward induction over the structure of terms or types is not enough to prove termination. In §3.3 we present a semantic proof of strong normalisation for λ_{ml}, by adapting a standard technique from the λ-calculus. We define an auxiliary notion of *reducibility* at every type, that is linked to strong normalisation, but amenable to induction over the structure of types. Roughly, reducibility is the logical predicate induced by strong normalisation at ground types. We can show that all reducible terms are strongly normalising, and then the fundamental theorem of logical relations ensures that in fact all terms are reducible.

Our presentation of reducibility follows the style in Chapter 6 of Girard et al.'s book [GLT89]. Our addition is to find a suitable definition for reducibility at computation types. A first informal attempt might be to echo the definition for functions:

(Bad 1) Term m of type TA is reducible if for all reducible n of type TB, the term $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n$ is reducible.

This is not inductive over types, as the definition of reducibility at type TA depends on reducibility at type TB, which may be more complex. We can try to patch this:

(Bad 2) Term m of type TA is reducible if for all strongly normalising n of type TB, the term $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n$ is strongly normalising.

This is now inductive, but in practice too weak to handle substitution. We need to look more closely at the contexts in which computation terms like m can be used. These *continuations* are nestings of $\mathsf{let}\ x \Leftarrow [\ ]\ \mathsf{in}\ n$, and give us our successful definition of reducibility:

(Good 1) Term m of type TA is reducible if for all reducible continuations K, the application $K @ m$ is strongly normalising.

Here application means plugging term m into the hole [ ] within K. Of course, we now have to define reducibility for continuations:

(Good 2) Continuation K accepting terms of type TA is reducible if for all reducible v of type A, the application $K @ \mathsf{val}(v)$ is strongly normalising.

This term $\mathsf{val}(v)$ is the trivial computation returning value v. By moving to the simpler value type A we avoid a potential circularity, and so get a notion of reducibility defined by induction on types. What is more, the characterisation by continuations is strong enough that the remainder of the strong normalisation proof goes through without undue difficulty.

Looking beyond reducibility, this jump over continuations offers a quite general method to leap-frog concepts from value type A up to computation type TA, whether or not we know the nature of T. If we write $K \top m$ when K applied to m strongly normalises, then for any predicate $\phi \subseteq A$ we define in turn:

$$\phi^{\top} = \{ K \mid K \top \mathsf{val}(v) \text{ for all } v \in \phi \}$$
$$\phi^{\top\top} = \{ M \mid K \top m \text{ for all } K \in \phi^{\top} \} \subseteq TA$$

This is our operation of $\top\top$-lifting: to take a predicate ϕ on value type A and return another $\phi^{\top\top}$ on the computation type TA, by a "leap-frog" over $\phi^{\top}$ on continuations.

We believe that the use of $\top\top$-lifting in the metalanguage λ_{ml} is original. It was inspired by similar constructions applied to specific notions of computation; it is also related to Pitts's $\top\top$-closure, and that in turn has analogues in earlier work on reducibility. §3.6 discusses this further.

3.2 Strong normalisation by translation

Our first proof of strong normalisation for λ_{ml} is by translation into a simpler calculus. In general we can prove strong normalisation for a calculus λ_1 by translation to another calculus λ_2 if the translation preserves reductions and the target calculus is already known to be strongly normalising.

We use the translation Φ in Figure 3.2 which strips out all computation types and terms; essentially by instantiating the type constructor T as the identity. The target calculus λ_{assoc} has exactly the types and terms of the simply-typed λ-calculus together with one extra reduction rule:

$$(\mathit{assoc}) \qquad \mathsf{app}(\mathsf{lam}(y,n),\mathsf{app}(\mathsf{lam}(x,m),l)) \longrightarrow$$
$$\mathsf{app}(\mathsf{lam}(x,\mathsf{app}(\mathsf{lam}(y,n),m)),l) \qquad \text{if } x \notin \mathit{fv}(n)$$

Types

$$
\begin{aligned}
\Phi(O) &= O \\
\Phi(TA) &= \Phi(A) \\
\Phi(A \to B) &= \Phi(A) \to \Phi(B)
\end{aligned}
$$

Terms

$$
\begin{aligned}
\Phi(x) &= x \\
\Phi(\mathsf{lam}(x, m)) &= \mathsf{lam}(x, \Phi(m)) \\
\Phi(\mathsf{app}(m, n)) &= \mathsf{app}(\Phi(m), \Phi(n)) \\
\Phi(\mathsf{val}(m)) &= \Phi(m) \\
\Phi(\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n) &= \mathsf{app}(\mathsf{lam}(x, \Phi(n)), \Phi(m))
\end{aligned}
$$

Figure 3.2: Translation Φ from λ_{ml} to λ_{assoc}.

These two terms are β-interconvertible, so *assoc*-reduction is admissible within the theory of the λ-calculus, but seems not very widely used. It is, for example, an instance of Sabry and Felleisen's reduction β_{lift}, that captures one kind of "administrative" reduction for code written in continuation-passing style [SF93, Definition 7].

The following result confirms that translation properly respects the structure and behaviour of λ_{ml} terms.

Lemma 3.2. *The translation Φ preserves types, substitution and reduction steps.*

(i) *If $m : A$ then $\Phi(m) : \Phi(A)$.*

(ii) $\Phi(m[x := n]) = \Phi(m)[x := \Phi(n)]$.

(iii) *If $m \to m'$ in λ_{ml} then $\Phi(m) \to \Phi(m')$ in λ_{assoc}.*

Proof. Parts (i) and (ii) follow by induction on the derivation of $m : A$ and the structure of m respectively. For part (iii) we observe that each reduction of λ_{ml} maps to a single reduction of λ_{assoc}:

- $\Phi(\to.\beta)$ and $\Phi(\to.\eta)$ are exactly β and η;

- $\Phi(T.\beta)$ is also β;
- $\Phi(T.\eta)$ is a special case of β; and
- $\Phi(T.T.\mathrm{CC})$ is *assoc*.

The proof for $T.\beta$ and $T.\eta$ uses result (ii) on substitution. □

Because of the addition of *assoc*-reduction, it is not immediate that the calculus λ_{assoc} is strongly normalising. However, it is close enough to base a combinatorial proof on the known β-normalisation, counting reduction steps of different kinds. We define three measures on lambda-terms: $s(m)$ is an asymmetrically weighted measure of term size; $b(m)$ counts β-reductions; and $f(m)$ combines the two.

$$\begin{aligned}
s(x) &= 1 \\
s(\mathsf{lam}(x,m)) &= s(m) \\
s(\mathsf{app}(m,n)) &= s(m) + 2s(n)
\end{aligned}$$

$$\begin{aligned}
b(m) &= \text{length of longest } \beta\text{-reduction sequence from } m \\
f(n) &= \langle b(m), s(m) \rangle \text{ lexicographically ordered.}
\end{aligned}$$

Both η and *assoc* decrease measure $s(m)$, which shows that they are strongly normalising on their own. Measure $b(m)$ is well-defined, as we know that β is strongly normalising, and naturally β decreases $b(m)$. The measure $f(m)$ is then enough to prove that λ_{assoc} is strongly normalising provided we can show that η and *assoc*-reductions do not increase $b(m)$.

For η this is straightforward, as it introduces no new β-redexes. The case of *assoc*-reduction requires more sophistication. What we must show is that for any $m \rightarrow_{assoc} m'$, if m' has a β-reduction sequence ρ' to normal form, then m has a matching sequence ρ that is at least as long. The following reduction diagram illustrates the need to consider complete sequences.

$$\begin{tikzcd}
\mathsf{app}(\mathsf{lam}(y,n),\mathsf{app}(\mathsf{lam}(x,m),l)) \arrow[r, "assoc"] \arrow[d, "\beta"'] & \mathsf{app}(\mathsf{lam}(x,\mathsf{app}(\mathsf{lam}(y,n),m)),l) \arrow[d, "\beta"] \\
n \arrow[d] & \mathsf{app}(\mathsf{lam}(x,n),l) \arrow[d] \\
? & \mathsf{app}(\mathsf{lam}(x,n),l')
\end{tikzcd}$$

where $x,y \notin fv(m) \cup fv(n)$.

Given a reduction sequence ρ' on the right we attempt to construct another ρ on the left, when it happens that variable x does not appear in m. Applying function $\mathsf{lam}(y,n)$ on both sides seems a natural match, but the resulting right-hand term $\mathsf{app}(\mathsf{lam}(x,n),l)$ may have reductions in l not available on the left. The solution is to postpone the left-hand reduction as long as possible, which we do by annotating terms to keep track of what reductions are pending. We omit the details of the proof — the final result is that for every complete sequence ρ' on the right, there is a matching ρ on the left, but individual reductions may be reordered.

Theorem 3.3. *Both λ_{assoc} and λ_{ml} are strongly normalising.*

Proof. By Lemma 3.2(iii) any infinite reduction sequence in λ_{ml} translates to an infinite reduction sequence λ_{assoc}. But there are no such sequences as every λ_{assoc} reduction decreases the well-founded measure $f(m)$. Thus every term in λ_{assoc} and λ_{ml} strongly normalises. □

3.3 Reducibility with continuations

Our second strong normalisation proof extends Tait's type-directed reducibility approach [Tai67], making use of ᴛᴛ-lifting. We follow closely the style of Girard et al. [GLT89, Chapter 6].

As explained earlier, the key step is to find an appropriate definition of reducibility for computation types, which we do by introducing a mechanism for managing continuations.

3.3.1 Continuations

Informally, a continuation should capture how the result of a computation might be used in a larger program. Our formal definition is structured to support inductive proof about these uses.

- A *term abstraction* $(x)n$ of type $TA \multimap TB$ is a computation term n of type TB with a distinguished free variable x of type A.
- A *continuation* K is a finite list of term abstractions, with length $|K|$.

$$K \ ::= Id \mid K \circ (x)n \qquad\qquad \begin{aligned} |Id| &= 0 \\ |K \circ (x)n| &= |K| + 1 \end{aligned}$$

- Continuations have types assigned using the following rules:

$$Id : TA \multimap TA \qquad\qquad \frac{(x)n : TA \multimap TB \qquad K : TB \multimap TC}{K \circ (x)n : TA \multimap TC} \ .$$

- We apply a continuation of type $TA \multimap TB$ to a computation term m of type TA by wrapping m in *let*-statements that use it:

$$\begin{aligned} Id \,@\, m &= m \\ (K \circ (x)n) \,@\, m &= K \,@\, (\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n) \end{aligned}$$

  Notice that when $|K| > 1$ this is a genuine nested stack of computations, not just simple sequencing: i.e.

$$\mathsf{let}\ x_1 \Leftarrow (\mathsf{let}\ x_2 \Leftarrow (\ldots(\mathsf{let}\ x_n \Leftarrow m\ \mathsf{in}\ n_n))\ldots\ \mathsf{in}\ n_2)\ \mathsf{in}\ n_1$$

  rather than

$$\mathsf{let}\ x_1 \Leftarrow m_1\ \mathsf{in}\ \mathsf{let}\ x_2 \Leftarrow m_2\ \mathsf{in}\ \ldots in \mathsf{let}\ x_n \Leftarrow m_n\ \mathsf{in}\ n\ .$$

- We define a notion of reduction on continuations:

$$K \longrightarrow K' \quad \overset{def}{\iff} \quad \forall m\,.\, K \,@\, m \to K' \,@\, m \quad \iff \quad K \,@\, x \to K' \,@\, x$$

$m \in \mathsf{red}_O$	if $m : O$ is strongly normalising
$f \in \mathsf{red}_{A\to B}$	if $\mathsf{app}(f,m) \in \mathsf{red}_B$ for all $m \in \mathsf{red}_A$
$p \in \mathsf{red}_{A\times B}$	if $\mathsf{proj}_1(p) \in \mathsf{red}_A$ and $\mathsf{proj}_2(p) \in \mathsf{red}_B$
$m \in \mathsf{red}_{TA}$	if $K \mathbin{@} m$ is strongly normalising for all $K \in \mathsf{red}^{\top}_{TA}$
$K \in \mathsf{red}^{\top}_{TA}$	if $K \mathbin{@} \mathsf{val}(v)$ is strongly normalising for all $v \in \mathsf{red}_A$.

Figure 3.3: Reducibility for λ_{ml}

where the right-hand equivalence follows from Proposition 3.1. A continuation K is *strongly normalising* if all reduction sequences starting from K are finite; and in this case we write $max(K)$ for the length of the longest.

Lemma 3.4. *If $K \longrightarrow K'$, for continuations K and K', then $|K'| \le |K|$.*

Proof. Suppose $K = Id \circ (x_1)n_1 \circ \cdots \circ (x_k)n_k$. Then its application $K \mathbin{@} x = \mathsf{let}\ x_1 \Leftarrow (\ldots(\mathsf{let}\ x_k \Leftarrow x\ \mathsf{in}\ n_k)\ldots)\ \mathsf{in}\ n_1$ and there are only two reductions that might change the length of K.

- $T.\eta$ where $n_i = \mathsf{val}(x_i)$ for some i. Then $K \to K'$ where $K' = Id \circ (x_1)n_1 \circ \cdots \circ (x_{i-1})n_{i-1} \circ (x_{i+1})n_{i+1} \circ \cdots \circ (x_k)n_k$ and $|K'| = |K| - 1$.
- $T.T.$CC may occur at position i for $1 \le i < n$ to give $K' = (x_1)n_1 \circ \cdots \circ (x_{i-1})n_i \circ (x_{i+1})(\mathsf{let}\ x_i \Leftarrow n_{i+1}\ \mathsf{in}\ n_i) \circ (x_{i+2})n_{i+2} \circ \cdots \circ (x_k)n_k$. Again $|K'| = |K| - 1$.

Hence $|K'| \le |K|$ as required. □

3.3.2 Reducibility and activity

Figure 3.3 defines two sets by induction on the structure of types: reducible terms red_A of type A, and reducible continuations $\mathsf{red}^{\top}_{TA}$ of type $TA \multimap TB$ for some B. As described in the introduction, for computations we use $\mathsf{red}_{TA} = \mathsf{red}^{\top\top}_{A}$.

Reduction	**Rewrite context**	**Active term**
$\to.\beta$	$\mathsf{app}(-, n)$	$\mathsf{lam}(x, m)$
$\to.\eta$	$-$	$\mathsf{lam}(x, \mathsf{app}(m, x))$
$\times.\beta i$	$\mathsf{proj}_i(-)$	$\mathsf{pair}(m, n)$
$\times.\eta$	$-$	$\mathsf{pair}(\mathsf{proj}_1(m), \mathsf{proj}_2(m))$
$T.\beta$	$\mathsf{let}\ x \Leftarrow -\ \mathsf{in}\ m$	$\mathsf{val}(n)$
$T.\eta$	$\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ -$	$\mathsf{val}(x)$
$T.T.\mathrm{CC}$	$\mathsf{let}\ y \Leftarrow -\ \mathsf{in}\ n$	$\mathsf{let}\ x \Leftarrow l\ \mathsf{in}\ m$

Figure 3.4: Activity for λ_{ml}

We also need to classify some terms as *inactive*[1]; we do this by decomposing every reduction into a rewrite context with a hole that must be plugged with a term of a particular form (see Figure 3.4).

From this we define:

- Term m is *active* if $R[m]$ is a redex for at least one of the rewrite contexts.
- Term m is *inactive* if $R[m]$ is not a redex for any of the rewrite contexts.

The inactive terms are those of the form x, $\mathsf{app}(m, n)$, $\mathsf{proj}_1(m)$ and $\mathsf{proj}_2(m)$; i.e. computation types add no new inactive terms.

The basic properties of reducibility now follow **(CR 1)**–**(CR 4)** of [GLT89].

Theorem 3.5. *For every term m of type A, the following hold.*

(i) *If $m \in \mathsf{red}_A$, then m is strongly normalising.*

(ii) *If $m \in \mathsf{red}_A$ and $m \to m'$, then $m' \in \mathsf{red}_A$.*

(iii) *If m is inactive, and whenever $m \to m'$ then $m' \in \mathsf{red}_A$, then $m \in \mathsf{red}_A$.*

[1] Girard et al [GLT89] (and the paper on which this chapter is based [LS05]) use the word *neutral*. But we have already used *neutral* for a stronger notion 2.4.3, so we use the word *inactive* instead. In the terminology of this thesis, we have that the term m is neutral iff it is inactive and all active subterms of m are in normal form.

(iv) *If m is inactive and normal then $m \in \mathsf{red}_A$.*

Proof. Part (iv) is a trivial consequence of (iii), so we need only prove (i)–(iii), which we do by induction over types. The proof for ground, function and product types proceeds as normal [GLT89].

Ground type

(i) Say $m \in \mathsf{red}_O$. Then m is SN by definition.

(ii) Suppose $m \in \mathsf{red}_O$ and $m \to m'$. Then m' is SN and hence reducible.

(iii) Take $m : O$ inactive with $m' \in \mathsf{red}_O$ whenever $m \to m'$. Then m' is SN whenever $m \to m'$. Thus m is SN and $m \in \mathsf{red}_O$.

Function types

(i) Say $m \in \mathsf{red}_{A \to B}$. By the induction hypothesis (iv) $x \in \mathsf{red}_A$ and by definition $\mathsf{app}(m, x) \in \mathsf{red}_B$. Now by the induction hypothesis (i) $\mathsf{app}(m, x)$ is SN and hence m is SN.

(ii) Suppose $m \in \mathsf{red}_{A \to B}$ and $m \to m'$. Whenever $n \in \mathsf{red}_A$ we have $\mathsf{app}(m, n) \in \mathsf{red}_B$. By the induction hypothesis (ii) $\mathsf{app}(m', n) \in \mathsf{red}_B$. Thus $m' \in \mathsf{red}_{A \to B}$.

(iii) Take $m : A \to B$ inactive with $m' \in \mathsf{red}_{A \to B}$ whenever $m \to m'$. Suppose $n \in \mathsf{red}_A$. We prove by induction on $max(n)$ that $\mathsf{app}(m, n) \in \mathsf{red}_B$. $\mathsf{app}(m, n)$ may reduce as follows:

- $\mathsf{app}(m', n)$, where $m \to m'$, which is reducible as $m' \in \mathsf{red}_{A \to B}$.
- $\mathsf{app}(m, n')$, where $n \to n'$, which is reducible by the induction hypothesis.

There are no other possibilities as m is inactive. Hence m is reducible.

Product types

(i) Say $m \in \mathsf{red}_{A\times B}$. Then $\mathsf{proj}_1(m) \in \mathsf{red}_A$ and by the induction hypothesis (i) $\mathsf{proj}_1(m)$ is SN. Hence m is SN.

(ii) Suppose $m \in \mathsf{red}_{A_1\times A_2}$ and $m \to m'$. Then $\mathsf{proj}_i(m) \in \mathsf{red}_{A_i}$, for $i = 1, 2$. By the induction hypothesis (ii) $\mathsf{proj}_i(m') \in \mathsf{red}_{A_i}$. Hence $m' \in \mathsf{red}_{A_1\times A_2}$.

(iii) Take $m : A_1 \times A_2$ inactive with $m' \in \mathsf{red}_{A_1\times A_2}$ whenever $m \to m'$. We prove by induction on $max(m)$ that $\mathsf{proj}_i(m) \in \mathsf{red}_{A_i}$ for $i = 1, 2$. As m is inactive, $\mathsf{proj}_i(m)$ can only reduce to $\mathsf{proj}_i(m')$, where $m \to m'$, which is reducible as $m' \in \mathsf{red}_{A_1\times A_2}$. Hence m is reducible.

Computation types

(i) Say $m \in \mathsf{red}_{TA}$. By the induction hypothesis (i), every $n \in \mathsf{red}_A$ has n and hence $\mathsf{val}(n)$ SN. Thus $Id{:}TA \multimap TA$ is reducible and m is SN as required.

(ii) Suppose $m \in \mathsf{red}_{TA}$ and $m \to m'$. For all $K \in \mathsf{red}^{\top}_{TA}$, application $K @ m$ is SN, and $K @ m \to K @ m'$, so $K @ m'$ is SN too and hence m' is reducible.

(iii) Take $m : TA$ inactive with $m' \in \mathsf{red}_{TA}$ whenever $m \to m'$. We have to show that $K @ m$ is SN for each $K \in \mathsf{red}^{\top}_{TA}$. First, we have that $K @ \mathsf{val}(x)$ is SN as $x \in \mathsf{red}_A$ by the induction hypothesis (iv). Hence K itself is SN, and we can work by induction on $max(K)$. Application $K @ m$ may reduce as follows:

- $K @ m'$, where $m \to m'$, which is SN by reducibility of K and m'.
- $K' @ m$, where $K \to K'$. Now, given any $n \in \mathsf{red}_A$, we have that $K @ \mathsf{val}(n)$ is SN as K is reducible; and $K @ \mathsf{val}(n) \to K' @ \mathsf{val}(n)$, so $K @ \mathsf{val}(n)$ is also SN. Thus K' is reducible with $max(K') < max(K)$, so by the induction hypothesis $K' @ m$ is SN.

There are no other possibilities as m is inactive. Hence $K @ m$ is SN, and m is reducible.

$\square$

3.3.3 Reducibility theorem

We show that all terms are reducible, and hence strongly normalising, by induction on their syntactic structure. This requires an appropriate lemma for each term constructor.

Functions

Lemma 3.6. *If $m : A \to B$ and $n : A$ are reducible, then so is* $\mathsf{app}(m, n)$.

Proof. By definition of reducibility on functions. □

Lemma 3.7. *If $m[x := n] : B$ is reducible for all reducible $n : A$ then* $\mathsf{lam}(x, m) : A \to B$ *is reducible.*

Proof. Suppose $n \in \mathsf{red}_A$. We show by induction on $max(m) + max(n)$ that the term $\mathsf{app}(\mathsf{lam}(x, m), n)$ is reducible. $\mathsf{app}(\mathsf{lam}(x, m), n)$ may reduce as follows:

- $\mathsf{app}(\mathsf{lam}(x, m'), n)$, where $m \to m'$, which is reducible by Theorem 3.5(ii) and the induction hypothesis.
- $\mathsf{app}(\mathsf{lam}(x, m), n')$, where $n \to n'$, which is reducible by Theorem 3.5(ii) and the induction hypothesis.
- $m[x := n]$, which is reducible by hypothesis.

By Theorem 3.5(iii), $\mathsf{app}(\mathsf{lam}(x, m), n)$ is reducible. Hence $\mathsf{lam}(x, m)$ is reducible. □

Products

Lemma 3.8. *If $m : A_1 \times A_2$ is reducible, then* $\mathsf{proj}_i(m)$ *is reducible.*

Proof. By definition of reducibility on products. □

Lemma 3.9. *If $m_1 : A_1$ and $m_2 : A_2$ are reducible, then so is* $\mathsf{pair}(m_1, m_2)$.

Proof. We show by induction on $max(m_1) + max(m_2)$ that $\mathsf{proj}_i(\mathsf{pair}(m_1, m_2))$ is reducible. $\mathsf{proj}_i(\mathsf{pair}(m_1, m_2))$ may reduce as follows:

- $\mathsf{proj}_i(\mathsf{pair}(m_1{}', m_2))$, where $m_1 \to m_1{}'$, which is reducible by Theorem 3.5(ii) and the induction hypothesis.
- $\mathsf{proj}_i(\mathsf{pair}(m_1, m_2{}'))$, where $m_2 \to m_2{}'$, which is reducible by Theorem 3.5(ii) and the induction hypothesis.
- m_i, which is reducible by hypothesis.

By Theorem 3.5(iii), $\mathsf{proj}_i(\mathsf{pair}(m_1, m_2))$ is reducible. Hence $\mathsf{pair}(m_1, m_2)$ is reducible. □

Computations

Lemma 3.10. *If $n : A$ is reducible, then so is $\mathsf{val}(n)$.*

Proof. Let K be a reducible continuation. By definition $K @ \mathsf{val}(n)$ is SN as n is reducible. Hence $\mathsf{val}(n)$ is reducible. □

We now wish to show that formation of let-terms preserves reducibility. That will be Lemma 3.12, but we first need a result on the strong normalisation of let-terms in context. This is the key component of our overall proof, and is where our attention to the stack-like structure of continuations pays off: the challenging case is the commuting conversion $T.T.$CC, which does not change its component terms; but does alter the continuation stack length, and this gives enough traction to maintain the induction proof.

Lemma 3.11. *Let $x : A$ be a variable, $m : A, n : TB$ be terms and $K : TB \multimap TC$ a continuation, such that m and $K @ n[x := m]$ are strongly normalising. Then, the term $K @ (\mathsf{let}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n)$ is strongly normalising.*

Proof. We show by induction on $|K| + max(K @ n) + max(m)$ that the reducts of $K @ (\mathsf{let}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n)$ are all SN. The interesting reductions are as follows:

- $T.\beta$ giving $K @ n[x := m]$, which is SN by hypothesis.
- $T.\eta$ when $N = \mathsf{val}(x)$, giving $K @ \mathsf{val}(m)$. But $K @ \mathsf{val}(m) = K @ n[x := m]$, which is again SN by hypothesis.

- $T.T.$CC in the case where $K = K' \circ (y)p$ with $x \notin fv(p)$; giving the reduct $K' @$ $(\mathsf{let}\, x \Leftarrow \mathsf{val}(m)\, \mathsf{in}\, (\mathsf{let}\, y \Leftarrow n\, \mathsf{in}\, p))$. We aim to apply the induction hypothesis with K' and $(\mathsf{let}\, y \Leftarrow n\, \mathsf{in}\, p)$ for K and n. Now

$$\begin{aligned} K' @ (\mathsf{let}\, y \Leftarrow n\, \mathsf{in}\, p)[x := m] &= K' @ (\mathsf{let}\, y \Leftarrow n[x := m]\, \mathsf{in}\, p) \\ &= K @ (n[x := m]) \end{aligned}$$

  which is SN by hypothesis. Also

$$|K'| + max(K' @ (\mathsf{let}\, y \Leftarrow n\, \mathsf{in}\, p)) + max(m) < |K| + max(K @ n) + max(m)$$

  as $|K'| < |K|$ and $(K' @ (\mathsf{let}\, y \Leftarrow n\, \mathsf{in}\, p)) = (K @ n)$. Applying the induction hypothesis gives that $K' @ (\mathsf{let}\, x \Leftarrow \mathsf{val}(m)\, \mathsf{in}\, (\mathsf{let}\, y \Leftarrow m\, \mathsf{in}\, p))$ is SN as required.

Other reductions are confined to $K @ n$ or m, and can be treated by the induction hypothesis, decreasing $max(K @ n)$ or $max(m)$ respectively. □

We are now in a position to state and prove a lemma on reducibility for let-terms.

Lemma 3.12. *If $m : TA$ is reducible and $N : TB$ with $n[x := p]$ reducible for all reducible $p : A$, then $(\mathsf{let}\, x \Leftarrow m\, \mathsf{in}\, n)$ is reducible.*

Proof. Let $K : TB \multimap TC$ be a reducible continuation. We need to show that $K @$ $(\mathsf{let}\, x \Leftarrow m\, \mathsf{in}\, n)$ is SN. Now for any $p : A$ reducible, $K @ n[x := p]$ is SN by reducibility of K and $n[x := p]$. But p is also SN, by Theorem 3.5(i), and so Lemma 3.11 shows that $K @ (\mathsf{let}\, x \Leftarrow \mathsf{val}(p)\, \mathsf{in}\, n)$ is SN too. Thus $K \circ (x)n$ is reducible and applying to reducible m gives that $K @ (\mathsf{let}\, x \Leftarrow m\, \mathsf{in}\, n)$ is SN. □

We finally move towards the desired result via a stronger result on substitutions into open terms.

Theorem 3.13. *Let m be any term, with free variables $x_1 : A_1, \ldots, x_k : A_k$. If $p_1 : A_1, \ldots, p_k : A_k$ are reducible then $m[x_1 := p_1, \ldots, x_k := p_k]$ is reducible.*

Proof. By induction on the structure of terms:

- x: $x[\vec{x} := \vec{p}] =$

  - x, if $x \neq x_i$ for $1 \leq i \leq k$, which is reducible by Theorem3.5(iv).
  - p_i, if $x = x_i$ for some i, which is reducible by hypothesis.

- $\mathsf{app}(m,n)$: By the induction hypothesis $m[\vec{x}{:=}\vec{p}]$ and $n[\vec{x}{:=}\vec{p}]$ are reducible, and by Lemma 3.6 so is $\mathsf{app}(m,n)[\vec{x}{:=}\vec{p}]$.
- $\mathsf{lam}(x,m)$: By the induction hypothesis $m[\vec{x}{:=}\vec{p}, x{:=}n]$ is reducible for all reducible n, and by Lemma 3.7 so is $\mathsf{lam}(x,m)[\vec{x}{:=}\vec{p}]$.
- $\mathsf{proj}_i(m)$: By the induction hypothesis $m[\vec{x}{:=}\vec{p}]$ is reducible, and by Lemma 3.8 so is $\mathsf{proj}_i(m)[\vec{x}{:=}\vec{p}]$.
- $\mathsf{pair}(m,n)$: By the induction hypothesis $m[\vec{x}{:=}\vec{p}]$ and $n[\vec{x}{:=}\vec{p}]$ are reducible, and by Lemma 3.9 so is $\mathsf{pair}(m,n)[\vec{x}{:=}\vec{p}]$.
- $\mathsf{val}(m)$: By the induction hypothesis $m[\vec{x}{:=}\vec{p}]$ is reducible, and by Lemma 3.10 so is $\mathsf{val}(m)[\vec{x}{:=}\vec{p}] = \mathsf{val}(m[\vec{x}{:=}\vec{p}])$.
- $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n$: By the induction hypothesis $m[\vec{x}{:=}\vec{p}]$ is reducible and $n[\vec{x}{:=}\vec{p}, x{:=}l]$ is reducible for all reducible l. Lemma 3.12 then gives that the term $(\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n)[\vec{x}{:=}\vec{p}] = \mathsf{let}\ x \Leftarrow m[\vec{x}{:=}\vec{p}]\ \mathsf{in}\ n[\vec{x}{:=}\vec{p}]$ is reducible too.

□

Theorem 3.14. *Each term m of λ_{ml} is reducible, and hence strongly normalising.*

Proof. Apply Theorem 3.13 with $p_i = x_i$, where the x_i are all reducible by Theorem 3.5(iv). This gives us that m is reducible, and by Theorem 3.5(i) also strongly normalising. □

Remark Notice the difference between reducibility lemmas for functions and products versus those for computations. In the case of functions and products, the reducibility lemma for the elimination follows immediately from the definition of reducibility, whereas the reducibility lemma for the introduction is non-trivial. Conversely in the case of computations, the reducibility lemma for the introduction follows immediately from the definition of reducibility, whereas the reducibility lemma for the elimination is non-trivial.

Adding η-expansion The strong normalisation proofs in this chapter are for calculi in which the η-rule is oriented as a contraction. The standard extensional normalisation by evaluation algorithms give long normal forms, which arise from η-expansion. Standard techniques can be used to treat η-expansion orthogonally to β- and CC-reduction (for example, see [AJ04]).

3.4 Variations on reducibility with continuations

In this section we apply $\top\top$-lifting to some extensions of λ_{ml}: with sum types, with exceptions; and in the computational lambda-calculus λ_c. Both sums and exceptions have existing normalisation results in the standard lambda-calculus (for example, [dG02] and [Lil99, Theorem 6.1]); we know of no prior proofs for them in λ_{ml}. More important, though, is to see how $\top\top$-lifting adapts to these features. The key step is to extend our formalised continuations with new kinds of observation. Once this is done, we can use these to lift predicates to computation types. The case of reducibility, and hence a proof of strong normalisation, then goes through as usual.

3.4.1 Reducibility for Sums

Prawitz showed how to extend the reducibility method[2] to sums [Pra71]. He worked in the context of proof theory, but the Curry-Howard isomorphism transfers this across to the simply-typed λ-calculus with sums. The method is quite intricate: for a term m of sum type to be reducible, not only must the immediate subterms of m be reducible, but also a certain class of subterms of m' must be reducible whenever m reduces to m'. This significantly complicates a general proof with sums.

In the computational metalanguage, we have an opportunity to simplify things by restricting attention to sums with a case construct in which each branch must be a computation. In fact, this is just the case construct of MIL. The reductions for sums are:

[2]Prawitz's *strong validity* corresponds to Girard's notion of reducibility.

$$
\begin{array}{lll}
(+.\beta i) & \mathsf{case}\ \mathsf{inj}_i(m)\ \ \mathsf{of}\ \ x_1 \Rightarrow n_1 & \longrightarrow\ n_i[x_i := m] \\
& \qquad\qquad\quad\ \ \mid\ \ x_2 \Rightarrow n_2 & \\
(+.\eta) & \mathsf{case}\ m\ \ \mathsf{of}\ \ x_1 \Rightarrow \mathsf{inj}_1(x_1) & \longrightarrow\ m \\
& \qquad\qquad\ \mid\ \ x_2 \Rightarrow \mathsf{inj}_2(x_2) & \\
(+.\mathrm{CC}) & \mathsf{let}\ y \Leftarrow \left(\mathsf{case}\ l\ \begin{array}{ll} \mathsf{of} & x_1 \Rightarrow m_1 \\ \mid & x_2 \Rightarrow m_2 \end{array} \right) \mathsf{in}\ n & \longrightarrow \\
& \mathsf{case}\ l\ \ \begin{array}{ll} \mathsf{of} & x_1 \Rightarrow \mathsf{let}\ y \Leftarrow m_1\ \mathsf{in}\ n \\ \mid & x_2 \Rightarrow \mathsf{let}\ y \Leftarrow m_2\ \mathsf{in}\ n \end{array} & \text{if } x_1, x_2 \notin fv(n)
\end{array}
$$

To record possible uses of sum terms, we introduce *sum continuations*:

- A *sum abstraction* $((x_1)n_1, (x_2)n_2)$ of type $A + B \multimap TC$ is a pair of term abstractions $(x_1)n_1$ of type $A \multimap TC$ and $(x_2)n_2$ of type $B \multimap TC$.

- A *sum continuation* S is a sum abstraction inside a continuation.

$$S\ ::= K \circ ((x_1)n_1, (x_2)n_2)$$

- Sum continuations are typed as follows:

$$\frac{((x_1)n_1, (x_2)n_2) : A + B \multimap TC \qquad K : TC \multimap TD}{K \circ ((x_1)n_1, (x_2)n_2) : A + B \multimap TD}$$

- We apply a sum continuation of type $A + B \multimap TC$ to a sum term m of type $A + B$ as follows:

$$(K \circ ((x_1)n_1, (x_2)n_2)) @ m = K @ (\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))$$

- We extend reduction on continuations to reduction on sum continuations:

$$S \to S' \quad \overset{def}{\Longleftrightarrow} \quad \forall m\,.\, S @ m \to S' @ m \quad \Longleftrightarrow \quad S @ x \to S' @ x$$

  A sum continuation S is *strongly normalising* if all reduction sequences starting from S are finite.

Lemma 3.15. *If $S \to S'$, for continuations S and S', then $|S'| \leq |S|$.*

As with computations, we extend reducibility to sums via a leap-frog over reducibility for their continuations:

- Sum continuation $S : A + B \multimap TC$ is reducible if:
  - $S @ (\mathsf{inj}_1(m))$ is strongly normalising for all reducible $m : A$ and
  - $S @ (\mathsf{inj}_2(n))$ is strongly normalising for all reducible $n : B$.
- Sum term $m : A + B$ is reducible if $S @ m$ is strongly normalising for all reducible sum continuations S of type $(A + B) \multimap TC$.

This reducibility is sufficient to prove strong normalisation for λ_{ml} with sums in the manner of §3.3.3. First we need to extend the proof of Theorem 3.5 to sum types:

Proof.

(i) Say $m \in \mathsf{red}_{A+B}$. Suppose $n_1{:}A, n_2{:}B$ are reducible. By the induction hypothesis (i) n_1, n_2 are SN. By induction on $max(n_1)$:

$$\begin{array}{lll} \mathsf{case}\ \mathsf{inj}_1(n_1) & \mathsf{of} & x_1 \Rightarrow \mathsf{val}(\mathsf{inj}_1(x_1)) \\ & \mid & x_2 \Rightarrow \mathsf{val}(\mathsf{inj}_2(x_2)) \end{array}$$

is SN and by induction on $max(n_2)$:

$$\begin{array}{lll} \mathsf{case}\ \mathsf{inj}_2(n_2) & \mathsf{of} & x_1 \Rightarrow \mathsf{val}(\mathsf{inj}_1(x_1)) \\ & \mid & x_2 \Rightarrow \mathsf{val}(\mathsf{inj}_2(x_2)) \end{array}$$

is SN. Thus $Id \circ ((x_1)\mathsf{val}(\mathsf{inj}_1(x_1)), (x_2)\mathsf{val}(\mathsf{inj}_1(x_2)))$ is reducible and:

$$\begin{array}{lll} \mathsf{case}\ m & \mathsf{of} & x_1 \Rightarrow \mathsf{val}(\mathsf{inj}_1(x_1)) \\ & \mid & x_2 \Rightarrow \mathsf{val}(\mathsf{inj}_2(x_2)) \end{array}$$

and hence m is SN as required.

(ii) Suppose $m \in \mathsf{red}_{A+B}$ and $m \to m'$. For all reducible $S : A + B \multimap TB$, application $S @ m$ is SN, and $S @ m \to S @ m'$, so $S @ m'$ is SN too and hence S' is reducible.

(iii) Take $m : A_1 + A_2$ inactive with $m' \in \mathsf{red}_{A_1+A_2}$ whenever $m \rightarrow m'$. We have to show that $S @ m$ is SN for each reducible $S : A_1 + A_2 \multimap TB$. First, we have that $S @ \mathsf{inj}_i(x_i)$ is SN as $x_i \in \mathsf{red}_{A_i}$ by the induction hypothesis (iv). Hence S itself is SN, and we can work by induction on $max(S)$. Application $S @ m$ may reduce as follows:

- $S @ m'$, where $m \rightarrow m'$, which is SN by reducibility of S and m'.
- $S' @ m$, where $S \rightarrow K'$. Now given any $l_i \in \mathsf{red}_{A_i}$, $S @ \mathsf{inj}_i(l_i) \rightarrow S' @ \mathsf{inj}_i(l_i)$ which is SN by reducibility of S. Thus S' is reducible with $max(S') < max(S)$, and by the induction hypothesis $S' @ m$ is SN.

There are no other possibilities as m is inactive. Hence $S @ m$ is SN, and m is reducible.

□

Now we just need to prove appropriate reducibility lemmas for sums.

Lemma 3.16. *If $m_i : A_i$ is reducible, then so is $\mathsf{inj}_i(m_i)$.*

Proof. Straightforward from the definition of reducibility for sum continuations. □

Just like for computations, the difficult part is the elimination term constructor, namely case. This time we need a suitably-crafted closure property for strong normalisation under $+.\beta i$-expansion.

Lemma 3.17. *Let $x_1, x_2 : A_1, A_2$ be variables, $m : A_i, n_1, n_2 : TB$ be terms and $K : TB \multimap TC$ a continuation, such that m and $K @ n_i[x_i := m]$ are strongly normalising. Then $K @ (\mathsf{case}\ \mathsf{inj}_i(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))$ is strongly normalising.*

The proof is similar to that of Lemma 3.11, except that one proves by induction on $|K| + max(K @ n_1) + max(K @ n_2) + max(m)$ that the reducts of

$$K @ (\mathsf{case}\ \mathsf{inj}_i(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))$$

are all SN.

We can then state and prove the reducibility lemma for case-terms.

Lemma 3.18. *If $m : A_1 + A_2$ is reducible and $n_1, n_2 : TB$ with $n_i[x_i := l]$ reducible for all reducible $l : A_i$, for $i = \{1,2\}$ then* $\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$ *is reducible.*

Proof. Let $K : TB \multimap TC$ be a reducible continuation. We need to show that $K @ (\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))$ is SN. Now for any $l_i : A_i$ reducible, $K @ n_i[x_i := l_i]$ is SN by reducibility of K and $n_i[x_i := l]$. But l_i is also SN, by Theorem 3.5(i), and so Lemma 3.17 shows that $K @ (\mathsf{case}\ \mathsf{inj}_i(l_i)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))$ is SN too. Thus $K \circ ((x_1)n_1, (x_2)n_2)$ is reducible and applying to reducible m gives that $K @ (\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))$ is SN. □

The strong normalisation result now follows from a straightforward extension of Theorem 3.13 with sums.

To apply this to a more general case construction, we can move to *frame stacks*: nested collections of elimination contexts for any type constructor [Pit00]. Frame stacks generalise continuations, and in §3.5 we use them to give a leap-frog definition of reducibility not just for computations, but also for sums, products and function types. This in turn gives a proof of strong normalisation for λ_{ml} with full sums, as well as the simply-typed lambda-calculus with sums.

One special case of this brings us full circle: λ_{ml} trivially embeds into the simply-typed λ-calculus with *unary* sums.

$$\mathsf{val}(m) \mapsto \mathsf{inj}(m) \qquad\qquad \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n \mapsto \mathsf{case}\ m\ \mathsf{of}\ (x \Rightarrow n)$$

As discussed in §2.2.4, the two languages are essentially the same, except that λ_{ml} has tighter typing rules and admits fewer reductions. Frame stacks and ᴛᴛ-reducibility then provide strong normalisation for both calculi.

3.4.2 Reducibility for Exceptions

Extending the computational metalanguage with the exceptional syntax of MIL (§2.3), and directing the conversion rules we obtain the new reduction rules:

$$\begin{array}{llcl}
(T.\beta) & \mathsf{try}\ x \Leftarrow \mathsf{val}(v)\ \mathsf{in}\ m\ \mathsf{unless}\ H & \longrightarrow & m[x := v] \\
(T_{exn}.\beta) & \mathsf{try}\ x \Leftarrow \mathsf{raise}(E)\ \mathsf{in}\ m\ \mathsf{unless}\ H & \longrightarrow & H(E) \\
(T.\eta) & \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{val}(x)\ \mathsf{unless}\ H & \longrightarrow & m \\
(T.T.\mathrm{CC}) & \mathsf{try}\ y \Leftarrow (\mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H)\ \mathsf{in}\ p\ \mathsf{unless}\ H' & \longrightarrow & \\
& \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ (\mathsf{try}\ y \Leftarrow n\ \mathsf{in}\ p\ \mathsf{unless}\ H')\ \mathsf{unless}\ H; H', & & \\
& \text{if } x \notin fv(p) \text{ and } x \notin fv(H'(E)) \text{ for any } E \in \mathbb{E} & &
\end{array}$$

Recall that let is a special case of try, with the empty handler $\langle\rangle$:

$$\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n = \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ \langle\rangle$$

For ᴛᴛ-lifting in this calculus, we generalise continuations to cover the observable behaviour of exception raising, by associating a handler to every step of the continuation.

$$\begin{gathered}
K ::= Id \mid K \circ ((x)n, H) \\
(K \circ ((x)n, H)) @ m \;=\; K @ (\mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H)
\end{gathered}$$

Computation types are extended to include exception annotations as in MIL. We now say that continuation $K : T_\varepsilon(A) \multimap T_{\varepsilon'}(B)$ is reducible if:

- $K @ (\mathsf{val}(m))$ is strongly normalising for all reducible $m : A$, and in addition
- $K @ (\mathsf{raise}(E))$ is strongly normalising for all exceptions $E \in \varepsilon$

Building ᴛᴛ-reducibility on this is enough to give strong normalisation for λ_{ml} with exceptions, with a proof in the manner of §3.3.3.

The proof for exceptions is effectively a combination of the original proof for computations, and that for sums. We concentrate on the key lemma. In fact it makes sense to divide this into two parts: one for $T.\beta$-closure, and the other for $T_{exn}.\beta$-closure. The first part is very similar to Lemma 3.11.

Lemma 3.19. *Let $x : A$ be a variable, $m : A, n_0, \ldots, n_k : T_{\varepsilon'}(B)$ be terms and $K : T_{\varepsilon'}(B) \multimap T_{\varepsilon''}(C)$ a continuation. Let $H = \langle (E_1, n_1), \ldots (E_k, n_k) \rangle$ be a handler for exceptions $E_1 \ldots E_k$. Suppose that $m, n_1, \ldots, n_k$ and $K @ n_0[x := m]$ are strongly normalising. Then, the term $K @ (\mathsf{try}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n_0\ \mathsf{unless}\ H)$ is also strongly normalising.*

The only difference is the presence of the exception annotations and the handler. The proof is a straightforward extension of that of Lemma 3.11. The second part is slightly more involved, because the effect annotations become important.

Lemma 3.20. *Let $x : A$ be a variable, $m : A, n_0, \ldots, n_k : T_{\varepsilon'}(B)$ be terms and $K : T_{\varepsilon'}(B) \multimap T_{\varepsilon''}(C)$ a continuation. Let $H = \langle (E_1, n_1), \ldots (E_k, n_k) \rangle$ be a handler for exceptions $E_1 \ldots E_k$. Let ε be a finite set of exceptions. Suppose that:*

- *$m, n_0, n_1, \ldots, n_k$ are strongly normalising,*
- *for $1 \leq i \leq k$ we have that $K @ n_i$ is strongly normalising, and*
- *for all exceptions $E \in (\varepsilon - \{E_1, \ldots E_k\})$, we have that $K @ \mathsf{raise}(E)$ is strongly normalising.*

Then, for all exceptions $E \in \varepsilon$, the term $K @ (\mathsf{try}\ x \Leftarrow \mathsf{raise}(E)\ \mathsf{in}\ n_0\ \mathsf{unless}\ H)$ is also strongly normalising.

Here the proof is by induction on $|K| + max(m) + max(n_0) + max(K @ n_1) + \cdots + max(K @ n_k) + \sum_{E \in (\varepsilon - \{E_1, \ldots E_k\})} (max(K @ \mathsf{raise}(E)))$. Note the restriction of the universal quantifier to a finite set of exceptions. This is not a problem, as the type system only supports a finite number of exceptions for a given term.

The rest of the strong normalisation proof is straightforward. Note that strong normalisation does not hold if, as in ML (and the full version of MIL), exceptions are allowed to carry values [Lil99].

3.4.3 Reducibility for the computational λ-calculus

Strong normalisation for λ_{ml} implies strong normalisation for the subcalculus λ_{ml*}. However, despite the close correspondence between λ_{ml*} and λ_c [SW97], we do not immediately get strong normalisation for λ_c. The reason is the existence of two additional reduction rules in λ_c:

$let.1$	$\mathsf{app}(p, m)$	$\longrightarrow$	$\mathsf{let}\ x \Leftarrow p\ \mathsf{in}\ \mathsf{app}(x, m)$	if $x \notin fv(m)$
$let.2$	$\mathsf{app}(v, q)$	$\longrightarrow$	$\mathsf{let}\ y \Leftarrow q\ \mathsf{in}\ \mathsf{app}(v, y)$	if $y \notin fv(v)$

where p, q range over non-values, and v ranges over values. We can adapt our proof, again using continuations in a leap-frog definition of reducibility:

Ground value	$v \in \mathsf{red}_O$	if v is strongly normalising
Function value	$v \in \mathsf{red}_{A \to B}$	if, for all $m \in \mathsf{red}_A \cup \mathsf{red}_A^{\top\top}$, $\mathsf{app}(v, m) \in \mathsf{red}_B^{\top\top}$
Continuation	$K \in \mathsf{red}_A^{\top}$	if, for all $v \in \mathsf{red}_A$, $K @ v$ is strongly normalising
Non-value	$p \in \mathsf{red}_A^{\top\top}$	if, for all $K \in \mathsf{red}_A^{\top}$, $K @ p$ is strongly normalising

The distinction between values and non-values is crucial. There is no explicit computation type constructor in λ_c, but non-values are always computations. Thus red_A is reducible values of type A, and $\mathsf{red}_A^{\top\top}$ is reducible non-values of type A, playing the role of red_{TA}. This $\top\top$-reducibility leads to a proof of strong normalisation for λ_c, accounting for both additional reductions.

3.5 Reducibility with frame stacks

We can view continuations as a mechanism for absorbing $T.T$.CC-reductions. In the key lemma, Lemma 3.11, $T.T$.CC-reduction does not change the main premises of the hypothesis (m and $K @ n[x := m]$ are strongly normalising), but simply reduces the size of the continuation. Frame stacks provide a more general mechanism for absorbing commuting conversions. We illustrate the use of frame stacks using an extension of λ_{ml} with sums in which instead of the MIL typing rules we use the more general typing rules of Figure 2.3.

Definition 3.21 (frame stacks).

$$\begin{array}{lrl}
(\textit{frames}) & F & ::= \mathsf{app}([\ ], n) \mid \mathsf{proj}_i([\ ]) \\
 & & \mid \mathsf{let}\ x \Leftarrow [\ ]\ \mathsf{in}\ n \\
 & & \mid \mathsf{case}\ [\ ]\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \\
(\textit{frame stacks}) & S & ::= Id \mid S \circ F
\end{array}$$

$$\begin{array}{lrl}
(\textit{stack length}) & |Id| & = O \\
 & |S \circ F| & = |S| + 1
\end{array}$$

$$\begin{array}{lrl}
(\textit{plugging}) & Id[m] & = m \\
 & (S \circ F)[m] & = S[(F[m])]
\end{array}$$

Frames are just elimination contexts — which when plugged with a corresponding introduction term result in β-redexes. A frame stack is a collection of nested elimination contexts. Note that our notation has changed from the last section. Instead of term abstractions we now have frames. Instead of continuation application we have frame stack plugging. The change of notation highlights the syntactic aspects of the frame stack approach.

The reduction rules appear in Figure 3.5. By considering a term m plugged into an arbitrary frame stack S we are able to reason by induction. Performing a CC-reduction on S always reduces the size of S. In effect frame stacks allow us to capture the essence of β-reduction whilst absorbing CC-reductions in the frame stack. The key step in proving strong normalisation is to obtain strong normalisation closure properties with respect to each β-expansion over all possible frame stacks. The closure properties capture the interactions between β-reductions and CC-reductions. Once we have proved these closure properties, it is relatively straightforward to prove that all terms are reducible, and hence strongly normalising, using the usual logical relations pattern.

Analogously to continuations we define reduction on frame stacks.

Definition 3.22 (frame stack reduction).

$$S \to S' \quad \overset{def}{\iff} \quad \forall m. S[m] \to S'[m] \quad \iff \quad S[x] \to S'[x]$$

$$
\begin{array}{rrcl}
(\to.\beta) & \mathsf{app}(\mathsf{lam}(x,m),n) & \longrightarrow & m[x:=n] \\
(\to.\eta) & \mathsf{lam}(x,\mathsf{app}(m,x)) & \longrightarrow & m, \qquad \text{if } x \notin \mathit{fv}(m) \\
(\times.\beta 1) & \mathsf{proj}_1(\mathsf{pair}(m_1,m_2)) & \longrightarrow & m_1 \\
(\times.\beta 2) & \mathsf{proj}_2(\mathsf{pair}(m_1,m_2)) & \longrightarrow & m_2 \\
(+.\beta 1) & \mathsf{case}\ \mathsf{inj}_1(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) & \longrightarrow & n_1[x_1:=m] \\
(+.\beta 2) & \mathsf{case}\ \mathsf{inj}_2(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) & \longrightarrow & n_2[x_2:=m] \\
(+.\eta) & \mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{inj}_1(x_1) \mid x_2 \Rightarrow \mathsf{inj}_2(x_2)) & \longrightarrow & m \\
(+.\cdot.\mathrm{CC}) & F[\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)] & \longrightarrow & \mathsf{case}\ m\ \ \mathsf{of}\ \ x_1 \Rightarrow F[n_1] \\
 & & & \qquad\qquad\ \mid\ \ x_2 \Rightarrow F[n_2] \\
(T.\beta) & \mathsf{let}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n & \longrightarrow & n[x:=m] \\
(T.\eta) & \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{val}(x) & \longrightarrow & m \\
(T.T.\mathrm{CC}) & \mathsf{let}\ y \Leftarrow (\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n)\ \mathsf{in}\ p & \equiv & \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p, \\
 & & & \text{if } x \notin \mathit{fv}(p)
\end{array}
$$

Figure 3.5: Reductions for an extension of λ_{ml} with sums and products

where the right-hand equivalence follows from Proposition 3.1. A frame stack S is *strongly normalising* if all reduction sequences starting from S are finite.

Lemma 3.23. *If $S \to S'$, for frame stacks S, S', then $|S'| \leq |S|$.*

The proof of this lemma is similar to that of Lemma 3.4.

Reducibility is defined on terms and stack frames.

Definition 3.24 (reducibility).

- *Id is reducible.*
- *$S \circ \mathsf{app}([\ ],n) : (A \to B) \multimap C$ is reducible if S and n are reducible.*
- *$S \circ \mathsf{proj}_i([\ ]) : (A \times B) \multimap C$ is reducible if S is reducible.*
- *$S : TA \multimap C$ is reducible if $S[\mathsf{val}(m)]$ is strongly normalising for all reducible $m : A$.*

- *$S : (A+B) \multimap C$ is reducible if $S\,[\mathsf{inj}_1(m)]$ is strongly normalising for all reducible $m : A$, and $S\,[\mathsf{inj}_2(n)]$ is strongly normalising for all reducible $n : B$.*

- *m:A is reducible if $S\,[m]$ is strongly normalising for all reducible $S : A \multimap C$.*

Lemma 3.25. *If $m : A$ is reducible then m is strongly normalising.*

Proof. Follows immediately from reducibility of Id and the definition of reducibility on terms. □

Lemma 3.26. *$x : A$ is reducible.*

Proof. By induction on A.

O: The only frame stack with a hole of base type is the identity. Clearly x is SN.

$A \to B$: Suppose $S : (A \to B) \multimap C$ is reducible. Then either $S = Id$, or $S = S' \circ n$ where $S' : B \multimap C$ and $n : A$ are reducible. The first case is trivial. For the second case we need to show that $S' \circ n[x] = S'[\mathsf{app}(x,n)]$ is SN. By the induction hypothesis and the definition of reducibility $S'[x']$ is SN, and thus S' is SN. By Lemma 3.25 n is SN. Now, by inspection, $\mathsf{app}(x,n)$ cannot interact with S'. Hence by induction on $max(S') + max(n)$, $S'[\mathsf{app}(x,n)]$ is SN.

$A \times B$: Suppose $S : (A \times B) \multimap C$ is reducible. Then either $S = Id$, $S = S' \circ \mathsf{proj}_1([\ ])$ where $S' : A \multimap C$, or $S = S' \circ \mathsf{proj}_2([\ ])$ where $S' : B \multimap C$. The first case is trivial. For the second case we need to show that $S' \circ \mathsf{proj}_1([\ ])[x] = S'[\mathsf{proj}_1(x)]$ is SN. By the induction hypothesis and the definition of reducibility $S'[x']$ is SN, and thus S' is SN. Now, by inspection, $\mathsf{proj}_1(x)$ cannot interact with S'. Hence by induction on $max(S')$, $S'[\mathsf{proj}_1(x)]$ is SN. The third case is symmetric to the second.

TA: By the induction hypothesis $x' : A$ is reducible. Thus given any reducible $S : TA \multimap TB$, $S\,[\mathsf{val}(x')]$ is SN. Hence by substitutivity $S\,[x]$ is SN.

$A+B$: By the induction hypothesis $x' : A$ is reducible. Thus given any reducible $S : (A+B) \multimap C$, $S\,[\mathsf{inj}_1(x')]$ is SN. Hence by substitutivity $S\,[x]$ is SN.

□

Lemma 3.27. *If $S : A \multimap C$ is reducible then S is strongly normalising.*

Proof. Immediate corollary of Lemma 3.26. □

Each type constructor has an associated β-rule. Each β-rule gives rise to an SN-closure property, which we use in the corresponding part of the proof of our main theorem. These closure properties generalise Lemma 3.11.

Lemma 3.28 (SN-closure).

$\to$ *If $S[m[x:=n]]$ and n are strongly normalising then $S[\mathsf{app}(\mathsf{lam}(x,m),n)]$ is strongly normalising.*

$\times.1$ *If $S[m]$ and n are strongly normalising then $S[\mathsf{proj}_1(\mathsf{pair}(m,n))]$ is strongly normalising.*

$\times.2$ *If $S[n]$ and m are strongly normalising then $S[\mathsf{proj}_2(\mathsf{pair}(m,n))]$ is strongly normalising.*

T *If $S[n[x:=m]]$ and m are strongly normalising then $S[\mathsf{let}\ x \Leftarrow \mathsf{val}(m)\ \mathsf{in}\ n]$ is strongly normalising.*

$+.1$ *If $S[n_1[x_1:=m]]$, $S[n_2]$ and m are strongly normalising then*
$S[\mathsf{case}\ \mathsf{inj}_1(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)]$ is strongly normalising.

$+.2$ *If $S[n_2[x_2:=m]]$, $S[n_1]$ and m are strongly normalising then*
$S[\mathsf{case}\ \mathsf{inj}_2(m)\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)]$ is strongly normalising.

Proof.

$\to, \times.1, \times.2$: By induction on $max(S) + max(m) + max(n)$.

T: By induction on $|S| + max(S[n[x:=m]]) + max(m)$.

$+.1$: By induction on $|S| + max(S[n_1[x_1:=m]]) + max(S[n_2]) + max(m)$.

$+.2$: By induction on $|S| + max(S[n_2[x_2:=m]]) + max(S[n_1]) + max(m)$.

□

Now we can obtain similar reducibility-closure properties for each type constructor.

Lemma 3.29 (reducibility-closure).

$\rightarrow$ *If* $m[x := n]$ *is reducible for all reducible* n*, then* $\mathsf{lam}(x, m)$ *is reducible.*

$\times$ *If* m, n *are reducible, then* $\mathsf{pair}(m, n)$ *is reducible.*

T *If* m *is reducible, and* $n[x := p]$ *is reducible for all reducible* p*, then* $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n$ *is reducible.*

$+$ *If* m *is reducible,* $n_1[x_1 := l]$ *is reducible for all reducible* l*, and* $n_2[x_2 := p]$ *is reducible for all reducible* p*, then* $\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$ *is reducible.*

Proof. Each property follows from the corresponding part of Lemma 3.28 using Lemmas 3.25-3.27. □

Theorem 3.30. *Let* m *be any term. Suppose* $x_1 : A_1, \ldots, x_k : A_k$ *includes all the free variables of* m*. If* $p_1 : A_1, \ldots, p_k : A_n$ *are reducible then* $m[x_1 := p_1, \ldots, x_k := p_k]$ *is reducible.*

Proof. By induction on the structure of terms using Lemma 3.29. □

Theorem 3.31 (strong normalisation). *All terms are strongly normalising.*

Proof. Let m be a term with free variables $x_1, \ldots, x_k$. By Lemma 3.26, $x_1, \ldots, x_k$ are reducible. Hence, by Theorem 3.30, m is strongly normalising. □

3.6 Related Work

Existing proofs of strong normalisation for λ_{ml} are based on translations into other calculi, which are already known to be strongly normalising. Benton et al., working from a logical perspective, translate λ_{ml} into a λ-calculus with sums, and then invoke the result of Prawitz [BBdP98]. In a report on *monadic type systems* — a generalisation of pure type systems and the computational metalanguage — Barthe et al. [BHT97] prove strong normalisation by translation into a λ-calculus with an extra reduction β'. They then use *finiteness of developments* [Bar84] to show that this target calculus is strongly normalising. Hatcliff and Danvy [HD94] state that T-reductions are strongly normalising, although they do not indicate a specific proof method. These approaches do not use reducibility, although Benton et al. mention a draft adaptation of Prawitz's reducibility for sums to handle computation types.

In [Pit00], Pitts employs $\top\top$-*closure* to define an operational form of relational parametricity for a polymorphic PCF. Here the computational effect is nontermination, and $(-)^{\top\top}$ leads to an operational analogue of the semantic concept of "admissible" relations. Abadi in [Aba00] investigates further the connection between $\top\top$-closure and admissibility.

The notion of $\top\top$-closed is different from our lifting: it expresses a property of a set of terms at a single type, whereas we lift a predicate ϕ on terms of type A to $\phi^{\top\top}$ on terms of a different type TA. However, the concept is clearly related, and the closure operation makes some appearance in the literature on reducibility, in connection with *saturation* and *saturated* sets of terms. Loosely, saturation is the property one wishes candidates for reducibility to satisfy; and this can sometimes be expressed as $\top\top$-closure. Examples include Girard's reducibility candidates for linear logic [Gir87, pp. 72–73] and Parigot's work on $\lambda\mu$ and classical natural deduction [Par97, pp. 1469–1471]. For Girard the relevant continuations are the linear duals $A^{\perp}$, while for Parigot they are applicative contexts, lists of arguments in normal form $\mathcal{N}^{<\omega}$. We conjecture that in their style our $\top\top$-lifting could be presented as an insertion $\{[v] \mid v : \mathsf{red}_A\}$ followed by saturation (although we then lose the notion of reducible continuations).

Malliès and Vouillon use *biorthogonality* in their work on ideal models for types; this is a closure operation based on an orthogonality relation matching our $K \top m$ [VM04a, VM04b]. They make a case for the importance of orthogonality, highlighting the connection to reducibility. They also deconstruct contexts into frame stacks for finer analysis: elsewhere, Vouillon notes the correspondence between different forms of continuation and possible observations [Vou04].

Chapter 4

Normalisation by evaluation

In this chapter we describe techniques for working with normalisation by evaluation. We have already seen, in §2.6, how a parameterised semantics allows us to neatly encapsulate different variants of a semantics. It is natural to implement parameterised semantics in SML using the module system [DF02], or in Haskell using type classes [AU04].

We use a parameterised semantics as the basis for investigating a number of variants of normalisation by evaluation for the computational metalanguage. In doing so we highlight some general techniques which are useful for applying normalisation by evaluation.

The first two sections of this chapter focus on the the computational aspect of the computational metalanguage. In §4.1 we begin with a standard monadic semantics. The monadic parameters are instantiated with appropriate *residualising monads* which contain enough syntactic information to retrieve a normal form: a continuation monad and a state-based accumulation monad. These monads are used to encapsulate computation bindings. In §4.2 we characterise residualising monads, using the proof technique of §2.7.2 to derive equations which residualising monads must satisfy.

In §4.3 we observe that *pure* residualising monads can actually be viewed as artefacts of normalisation by evaluation, and it is in fact sufficient to use the internal monad of a sufficiently expressive metalanguage. Specifically, the metalanguage should either include support for delimited continuations (corresponding to the continuation monad) or state (corresponding to the accumulation monad). The side-effecting operations appear in the definition of the reification and reflection functions but not in the semantics itself. The ability to move syntactic information, and also computation, back and forth between the semantics and the reification function, proves to be a powerful tool in the application of normalisation by evaluation.

The rest of this chapter explores other aspects of normalisation by evaluation. In §4.4 we consider how to restrict normalisation by evaluation in order to prevent η-expansion, and how to perform η-contraction instead. In §4.5 we discuss sums. The usual TDPE approach of greedily eliminating sums as soon as they are introduced is taken, and an implementation using delimited continuations is given. In §4.6 we present a new algorithm which uses binding trees, a zipper structure, and a single reference cell. In §4.7 we outline how to handle polymorphism and recursive types. In §4.8 we show how types can be embedded in the semantics. This allows the type index on the reify function to be dispensed with. Finally, in §4.9 we discuss the practicalities of implementing normalisation by evaluation in ML.

4.1 The computational metalanguage using monads

We begin with a straightforward parameterised semantics for the computational metalanguage (with parameters: *lam*, *app*, *val*, *let*):

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\mathsf{lam}(x^A, e)]\!]_\rho &= lam(\lambda s^A. [\![e]\!]_{\rho[x \mapsto s]}) \\
[\![\mathsf{app}(e_1, e_2)]\!]_\rho &= app([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![\mathsf{val}(e)]\!]_\rho &= val([\![e]\!]_\rho) \\
[\![\mathsf{let}\ x^A \Leftarrow e_1\ \mathsf{in}\ e_2]\!]_\rho &= let([\![e_1]\!]_\rho, \lambda s^A. [\![e_2]\!]_{\rho[x \mapsto s]})
\end{aligned}$$

We restrict ourselves to a monadic residualising semantics:

$$\begin{aligned}
[\![O]\!] &= \Lambda ml_O \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![TA]\!] &= Comp([\![A]\!])
\end{aligned}$$

Comp is a further parameter corresponding to the particular choice of monad we use. We call *Comp*, *val* and *let* the monadic parameters.

The monad laws The residualising semantics must satisfy the monad laws:

$$let(val(v), f) = f\ v \tag{4.1}$$

$$let(s, \lambda v.val(v)) = s \tag{4.2}$$

$$let(let(s, \lambda v.s'), \lambda v'.s'') = let(s, \lambda v.let(s', \lambda v'.s'')) \tag{4.3}$$

The monad laws are just the semantic counterparts of the monadic conversion rules, or in other words the monadic conversion rules expressed in higher order abstract syntax. These laws ensure that the residualising semantics is sound.

Remark The parameters *Comp*, *val*, and *let* correspond to the standard categorical notion of a Kleisli triple $(\mathrm{T}, \eta, \_^*)$ where:

$$\begin{aligned} Comp &= \mathrm{T} \\ val(v) &= \eta(v) \\ let(s, f) &= f^*\ s \end{aligned}$$

Normal forms for the computational metalanguage are given by:

Normal forms	$m ::= n^0 \mid \mathsf{lam}(x^A, m) \mid \mathsf{val}(m) \mid \mathsf{let}\ y^A \Leftarrow n^{TA}\ \mathsf{in}\ m$
Neutral terms	$n^A ::= x^A \mid \mathsf{app}(n^{B \to A}, m)$

The non-computational part of the algorithm is standard:

$$\begin{aligned} lam(f) &= f \\ app(f, s) &= f\ s \end{aligned}$$

$$\begin{aligned} \downarrow^O e &= e \\ \downarrow^{A \to B} f &= \mathsf{lam}(x^A, \downarrow (f(\uparrow^A x))), \quad x \text{ fresh} \\ \downarrow^{TA} c &= \ldots \end{aligned}$$

$$\begin{aligned} \uparrow^O e &= e \\ \uparrow^{A \to B} e &= \lambda s.\ \uparrow^B (\mathsf{app}(e, (\downarrow^A s))) \\ \uparrow^{TA} e &= \ldots \end{aligned}$$

The monad needs to contain enough syntactic information in order to be able to store the bindings associated with the let construct. The $T.T.$CC rule allows us to view the bindings associated with a computation as a list. In essence, the monad should encapsulate a stack of bindings.

One option is to use the continuation monad with answer domain the set of normal form terms:

$$\begin{aligned} \mathit{Comp}([\![A]\!]) &= ([\![A]\!] \to \Lambda\mathit{ml\text{-}nf}) \to \Lambda\mathit{ml\text{-}nf} \\ \mathit{val}(s) &= \lambda\kappa.\kappa\ s \\ \mathit{let}(t,f) &= \lambda\kappa.t(\lambda s.f\ s\ \kappa) \end{aligned}$$

$$\downarrow^{TA} t = t(\lambda s.\mathsf{val}(\downarrow^{A} s))$$

$$\uparrow^{TA} e = \lambda\kappa.\mathsf{let}\ x^{A} \Leftarrow e\ \mathsf{in}\ \kappa(\uparrow^{A} x), \quad x \text{ fresh}$$

The semantics is a standard continuation semantics. The reification function just passes the identity continuation. The reflection function uses the continuation to gather together let-bindings. The important aspect of this monad is that the answer domain is syntactic. It is this property which allows the list of bindings to be represented.

Another alternative, with a more explicit representation for the stack of bindings, is a state-based monad such as the accumulation monad over $\mathbf{V} \times \Lambda\mathit{ml}$ lists:

$$\begin{aligned} \mathit{Comp}([\![A]\!]) &= \langle \mathbf{V} \times \Lambda\mathit{ml} \rangle \times [\![A]\!] \\ \mathit{val}(s) &= (\langle\rangle, s) \\ \mathit{let}((bs,s),f) &= (bs +\!\!+ bs', s'), \quad \text{where } (bs',s') = f\ s \end{aligned}$$

$$\begin{aligned} \downarrow^{TA} (\langle\rangle, s) &= \mathsf{val}(\downarrow^{A} s) \\ \downarrow^{TB} ((x^{A}, e) :: bs, s) &= \mathsf{let}\ x^{A} \Leftarrow e\ \mathsf{in} \downarrow^{TB} (bs, s) \end{aligned}$$

$$\uparrow^{TA} e = (\langle x^{A}, e\rangle, \uparrow^{A} x), \quad x \text{ fresh}$$

Here a computation is a pair consisting of a list of computation bindings and a semantic value. Lifting a value to a computation just gives a trivial computation with no bindings, whereas sequencing uses the value of the first computation to generate another computation, then concatenates the lists of bindings. Reifying a computation simply wraps the reified value inside its bindings. Reflecting a computation term creates a new binding. Another possibility is to accumulate bindings using higher order functions.

Remark It should be noted that there is not a single *canonical* choice of residualising monad. Accumulation monads and continuation monads are both equally valid.

4.2 Characterising the residualising monad

The choice of continuation monad versus accumulation monad in the residualising semantics raises the question of what constraints the residualising monad should satisfy. Filinski [Fil01b] gives one possible answer in the setting of the computational λ-calculus. We arrive at a slightly different answer motivated by the proof technique of §2.7.2. We know from Chapter 3 that normal forms exist. Assuming soundness of the residualising semantics, in order to prove correctness of *norm* all that remains is to prove that $norm(e) = e$ for all normal forms e. Extending the inductive proof of §2.7.2 to the computational metalanguage we have two new cases corresponding to the computational syntax constructors:

$$norm(\mathsf{val}(m)^{TA}) = \mathsf{val}(m) \tag{4.4}$$
$$norm(\mathsf{let}\ x^{A} \Leftarrow n^{TA}\ \mathsf{in}\ m) = \mathsf{let}\ x^{A} \Leftarrow n^{TA}\ \mathsf{in}\ m \tag{4.5}$$

These equations can be rewritten in terms of the monadic parameters. Expanding the left-hand side of the first equation gives:

$$\downarrow^{TA} val([\![m]\!]_{\uparrow}) = \mathsf{val}(m)$$

Using the induction hypothesis the right-hand side expands to:

$$\downarrow^{TA} val([\![m]\!]_{\uparrow}) = \mathsf{val}(\downarrow^{A} [\![m]\!]_{\uparrow})$$

Moving to the second equation, and expanding the left-hand side:

$$\downarrow^{TB} \mathit{let}([\![n^{TA}]\!]_{\uparrow}, \lambda s.[\![m]\!]_{\uparrow[x \mapsto s]}) = \mathsf{let}\ x^A \Leftarrow n^{TA}\ \mathsf{in}\ m$$

By the reflection equation (2.9) this can be rewritten as:

$$\downarrow^{TB} \mathit{let}(\uparrow^{TA} n, \lambda s.[\![m]\!]_{\uparrow[x \mapsto s]}) = \mathsf{let}\ x^A \Leftarrow n^{TA}\ \mathsf{in}\ m$$

Expanding the right-hand side using the induction hypothesis:

$$\downarrow^{TB} \mathit{let}(\uparrow^{TA} n, \lambda s.[\![m]\!]_{\uparrow[x \mapsto s]}) = \mathsf{let}\ x^A \Leftarrow n^{TA}\ \mathsf{in} \downarrow^{TB} [\![m]\!]_{\uparrow}$$

Hence it is a necessary and sufficient condition that the residualising monad supports extensions of the reification and reflection functions such that the following equations hold:

$$\downarrow^{TA} \mathit{val}([\![m]\!]_{\uparrow}) = \mathsf{val}(\downarrow^{A} [\![m]\!]_{\uparrow}) \tag{4.6}$$

$$\downarrow^{TB} \mathit{let}(\uparrow^{TA} n, \lambda s.[\![m]\!]_{\uparrow[x \mapsto s]}) = \mathsf{let}\ x^A \Leftarrow n^{TA}\ \mathsf{in} \downarrow^{TB} [\![m]\!]_{\uparrow} \tag{4.7}$$

These equations play a similar role to Filinski's *bind* and *collect* [Fil01b]. It is straightforward to verify that these equations are satisfied by the normalisation by evaluation algorithms using either the continuation monad or the accumulation monad. We say that a residualising monad is *valid* if it satisfies (4.6) and (4.7).

Proposition 4.1. *The continuation monad of §4.1 is valid.*

Proof.
(4.6):

$$\begin{aligned}\downarrow^{TA} \mathit{val}([\![m]\!]_{\uparrow}) &= \downarrow^{TA} \lambda\kappa.\kappa([\![m]\!]_{\uparrow})\\ &= (\lambda\kappa.\kappa([\![m]\!]_{\uparrow}))(\lambda s.\mathsf{val}(\downarrow^{A} s))\\ &= \mathsf{val}(\downarrow^{A} [\![m]\!]_{\uparrow})\end{aligned}$$

(4.7):

$$\begin{aligned}\downarrow^{TB} \mathit{let}(\uparrow^{TA} n, \lambda s.[\![m]\!]_{\uparrow[x \mapsto s]}) &= \downarrow^{TB} \lambda\kappa.\uparrow^{TA} n(\lambda s.[\![m]\!]_{\uparrow[x \mapsto s]}\kappa)\\ &= \downarrow^{TB} \lambda\kappa.\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ \kappa[\![m]\!]_{\uparrow}\\ &= \mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ (\lambda s.\mathsf{val}(\downarrow^{A} s))(\mathsf{val}([\![m]\!]_{\uparrow}))\\ &= \mathsf{let}\ x^A \Leftarrow n^{TA}\ \mathsf{in} \downarrow^{TB} [\![m]\!]_{\uparrow}\end{aligned}$$

□

Proposition 4.2. *The accumulation monad of §4.1 is valid.*

Proof.
(4.6):

$$\begin{aligned}\downarrow^{TA} val([\![m]\!]_{\uparrow}) &= \downarrow^{TA} (\langle\rangle, [\![m]\!]_{\uparrow}) \\ &= \mathsf{val}(\downarrow^{A} [\![m]\!]_{\uparrow})\end{aligned}$$

(4.7):

$$\begin{aligned}\downarrow^{TB} let(\uparrow^{TA} n, \lambda s.[\![m]\!]_{\uparrow[x\mapsto s]}) &= \downarrow^{TB} let((\langle(x^A, n)\rangle, \uparrow^{A} x), \lambda s.[\![m]\!]_{\uparrow[x\mapsto s]}) \\ &= \downarrow^{TB} ((x^A, n) :: bs', s'), \qquad \text{where } (bs', s') = [\![m]\!]^{A}_{\uparrow} x \\ &= \mathsf{let}\ x^A \Leftarrow n^{TA}\ \mathsf{in} \downarrow^{TB} [\![m]\!]_{\uparrow}\end{aligned}$$

□

4.3 The computational metalanguage using side-effects

It is apparent, just by inspecting actual TDPE implementations [Dan98, DF02], that if our metalanguage is 'sufficiently expressive' then we can in fact use its internal monad. Extra expressive power is required in the form of side-effecting operations used in the definition of ↓. For instance, if the metalanguage supports state, then rather than using an accumulation monad, it is possible to construct a global list of bindings.

4.3.1 State and delimited continuations

It is necessary to modify our notion of metalanguage. Up to this point it has been reasonable to view the metalanguage as a *pure* mathematical language. Apart from name generation, and occasionally non-termination, our algorithms have not allowed side-effects. We now wish to incorporate state and control effects in the metalanguage. Given that our implementation language is going to be Standard ML extended with first-class control, this is what we shall base our metalanguage on. Thus, we shall now assume our metalanguage is call-by-value and supports state and first-class control.

State is handled in the metalanguage using reference cells. Following ML, reference cells can be:

- created: $\mathit{ref}(v)$ returns a fresh reference cell initialised with the value v.
- assigned to: $x := v$ assigns the value v to the cell x.
- dereferenced: $!x$ gives the contents of reference cell x.

State provides a side-effecting alternative to the accumulation monad. Similarly we can construct a side-effecting alternative to the continuation monad — *delimited continuations*. Delimited continuations are manipulated using the shift and reset control operators [DF90, DF92]. The reset operator $\lessdot\cdot\gtrdot$ delimits the start of a continuation. We call the continuation up to the reset an *initial continuation*. The shift operator $\mathcal{S}$ takes a λ-abstraction as an argument, and delimits the end of a continuation. The operation $\mathcal{S}(\lambda\kappa.e)$ behaves as follows:

- First, the continuation delimited by the nearest enclosing reset (initial continuation) and the shift is bound to κ.
- Second, the expression e (which may depend on κ) is evaluated.
- Third, the resulting value is passed to the initial continuation.

For example, consider:

$$100 + \lessdot 1 + \mathcal{S}(\lambda\kappa.\kappa(2) + \kappa(3))\gtrdot = 107$$

The reset operator sets the initial continuation to $100 + [\ ]$. Inside the second argument to the shift operator the variable κ is bound to $1 + [\ ]$. Then $\kappa(2) + \kappa(3) = (1 + [2]) + (1 + [3]) = 7$ is passed to the initial continuation $100 + [7] = 107$.

Here is another example:

$$100 + \lessdot\text{if } (\mathcal{S}(\lambda\kappa.\kappa(\text{true}) + \kappa(\text{false}))) \text{ then } 1 \text{ else } 2\gtrdot = 103$$

The initial continuation is $100 + [\ ]$. The delimited continuation κ is then bound to if $[\ ]$ then 1 else 2. Finally, $\kappa(\text{true}) + \kappa(\text{false}) = 1 + 2 = 3$ is passed to the initial continuation to give $100 + [3] = 103$.

In fact the argument to shift need not be a λ-abstraction, although it must be a function. The argument can always be η-expanded to give a λ abstraction: $\mathcal{S}m = \mathcal{S}(\lambda\kappa.m\ \kappa)$.

Formally, shift and reset are defined by their denotational semantics, which is given by their CPS transforms [DF90, DF92]. We want to be able to use equational reasoning (in direct-style) on expressions involving shift and reset. We take advantage of Kameyama and Hasegawa's axiomatisation of delimited continuations [KH03]. The axioms for shift and reset are as follows:

(β_v)	$(\lambda x.m)v = m[x := v]$	
(η_v)	$\lambda x.vx = v,$	if x is not free in v
(β_Ω)	$(\lambda x.P[x])m = P[m],$	if x is not free in P
(reset-value)	$\langle v \rangle = v$	
(reset-lift)	$\langle (\lambda x.m)\langle n \rangle \rangle = (\lambda x.\langle m \rangle)\langle n \rangle$	
($\mathcal{S}$-elim)	$\mathcal{S}(\lambda\kappa.\kappa\ m) = m,$	if κ is not free in m
(reset-$\mathcal{S}$)	$\langle P[\mathcal{S}m] \rangle = \langle m(\lambda x.\langle P[x] \rangle) \rangle,$	if x is not free in P
($\mathcal{S}$-reset)	$\mathcal{S}(\lambda\kappa.\langle m \rangle) = \mathcal{S}(\lambda\kappa.m)$	

where x ranges over variables (in the metalanguage), v ranges over values (in the metalanguage), m, n range over terms (in the metalanguage), and f ranges over *pure evaluation contexts* (in the metalanguage).

Implicitly we are assuming the metalanguage is an extension of the call-by-value λ-calculus. Values in the metalanguage are variables, constants and λ-abstractions. Pure evaluation contexts are given by the grammar:

$$P ::= [\,] \mid P\ m \mid v\ P$$

To avoid confusion when considering metalanguage terms which contain object language terms, in the sequel we restrict x, v, m, n to range only over object language entities and not over metalanguage entities.

Call-by-value via thunks Filinski [Fil94] implements shift and reset as functions in SML/NJ using call/cc and a single reference cell. Note that $\langle\cdot\rangle$ cannot be defined directly as a call-by-value function, as the operation $\langle s \rangle$ would then be ill-defined. The argument s must not be evaluated until the start of the delimited continuation associated with the reset has been registered. Filinski's solution is to delay the evaluation of the argument to reset by passing it as a *thunk* — a function from unit to some other type [HD97]. As we use ML as our implementation language, we take the same approach and define the call-by-value function $<\cdot>$ as follows:

$$<t> = \lessdot t() \gtrdot$$

where t is a thunk. We use $<\lambda().s>$ in place of $\lessdot s \gtrdot$. For instance, the last example becomes:

$$100 + <\lambda().\text{if } (\mathcal{S}(\lambda\kappa.\kappa(\text{true}) + \kappa(\text{false}))) \text{ then } 1 \text{ else } 2> = 103$$

4.3.2 Normalisation by evaluation with side-effects

We assume that the metalanguage supports both delimited continuations and state. This is not an unreasonable assumption for an implementation language. For instance, SML/NJ supports state and call/cc, and delimited continuations are implementable in terms of state and call/cc [Fil94].

Using the internal monad of the metalanguage (in other words the identity monad with side-effects), the monadic parameters are:

$$\begin{aligned} Comp([\![A]\!]) &= {}^{\circ}[\![A]\!] \\ val(s) &= s \\ let(val(s), f) &= f\ s \end{aligned}$$

Notice that this looks very much like the identity monad. Indeed, this definition of the semantics does make sense in a pure metalanguage, and can be used to "run" programs in such a setting. However, in order to use this definition for normalisation by evaluation it is crucial that the metalanguage has side-effects. We use the ${}^{\circ}(\cdot)$ operator to introduce the possibility of side-effects.

Reification and reflection are defined abstractly in terms of side-effecting parameters *bind* and *collect*:

$$\begin{aligned}
\downarrow^{A} &: [\![A]\!] \rightarrow \Lambda\mathit{ml\text{-}nf}_{A} \\
\uparrow^{A} &: \Lambda\mathit{ml\text{-}ne}_{A} \rightarrow [\![A]\!]
\end{aligned}$$

$$\begin{aligned}
\downarrow^{O} e &= e \\
\downarrow^{A\rightarrow B} f &= \mathsf{lam}(x^{A}, \downarrow^{B} (f(\uparrow^{A} x))), \quad x \text{ fresh} \\
\downarrow^{TA} s &= \mathit{collect}(\mathsf{val}(\downarrow^{A} s))
\end{aligned}$$

$$\begin{aligned}
\uparrow^{O} e &= e \\
\uparrow^{A\rightarrow B} e &= \lambda s^{A}.\, \uparrow^{B} (\mathsf{app}(e, (\downarrow^{A} s))) \\
\uparrow^{TA} e &= \mathit{bind}^{TA}(e) \\
\mathit{norm}(e^{A}) &= \downarrow^{A} ([\![e]\!]_{\uparrow})
\end{aligned}$$

Informally, *bind* registers a ‘let-binding’ (using side-effects) and *collect* extracts all of the registered bindings (using side-effects). We have two concrete implementations of *bind* and *collect* in mind: one using delimited continuations, and the other using state. Inspired by TDPE implementations and Filinski’s monadic reflection operations [Fil96] we try the following definitions using delimited continuations:

$$\begin{aligned}
\mathit{collect}(e) &= <\lambda().e> \\
\mathit{bind}^{TA}(e) &= \mathcal{S}(\lambda\kappa.\mathsf{let}\ x^{A} \Leftarrow e\ \mathsf{in}\ <\lambda().\kappa(\uparrow^{A} x)>), \quad x \text{ fresh}
\end{aligned}$$

This cannot work because shift may be called outside of any enclosing reset. For instance, consider normalising $\mathsf{lam}(x^{TO}, x)$. This is interpreted as a function and then reified at function type:

$$\downarrow^{TO\rightarrow TO} f = \mathsf{lam}(x^{TO}, \downarrow^{TO} (f(\uparrow^{TO} x))), \quad x \text{ fresh}$$

The argument is reflected at computation type and shift is invoked. The reset which was supposed to delimit the start of the continuation captured by the shift operator occurs inside the call to reify at type TO. In order to solve the problem, the call to *bind* (which contains the shift operator) needs to be delayed until reify is called. This can be achieved using thunks as we did with the reset operator. The monadic parameters are changed as follows:

$$\begin{aligned} \mathit{Comp}([\![A]\!]) &= \mathbf{1} \to {}^{\circ}[\![A]\!] \\ \mathit{val}(s) &= \lambda().s \\ \mathit{let}(s, f) &= \lambda().f\ (s()) \end{aligned}$$

This is reminiscent of Fürhmann and Thielecke's delaying transform [FT04, Section 6], in that we delay the evaluation of the body of a function, rather than the argument passed to it. Normalisation by evaluation is given by:

$$\begin{aligned} &\downarrow^{A}:[\![A]\!] \to \Lambda \mathit{ml\text{-}nf}_{A} \\ &\uparrow^{A}:\Lambda \mathit{ml\text{-}ne}_{A} \to [\![A]\!] \end{aligned}$$

$$\begin{aligned} \downarrow^{O} e &= e \\ \downarrow^{A\to B} f &= \mathsf{lam}(x^{A}, \downarrow^{B} (f(\uparrow^{A} x))), \quad x \text{ fresh} \\ \downarrow^{TA} s &= \mathit{collect}(\lambda().\mathsf{val}(\downarrow^{A} s())) \\ \\ \uparrow^{O} e &= e \\ \uparrow^{A\to B} e &= \lambda s^{A}.\uparrow^{B} (\mathsf{app}(e, (\downarrow^{A} s))) \\ \uparrow^{TA} e &= \lambda().\mathit{bind}^{TA}(e) \\ \mathit{norm}(e^{A}) &= \downarrow^{A} ([\![e]\!]_{\uparrow}) \end{aligned}$$

Now we derive counterparts to (4.6) and (4.7) in terms of *bind* and *collect*. The left-hand-side of Equation (4.6) can be rewritten as:

$$\begin{aligned} \downarrow^{TA} \mathit{val}([\![m]\!]_{\uparrow}) &= \downarrow^{TA} \lambda().[\![m]\!]_{\uparrow} \\ &= \mathit{collect}(\lambda().\mathsf{val}(\downarrow^{A} [\![m]\!]_{\uparrow})) \\ &= \mathit{collect}(\lambda().\mathsf{val}(m)) \end{aligned}$$

and by the induction hypothesis, the right-hand-side can be rewritten as:

$$\mathsf{val}(\downarrow^{A} [\![m]\!]_{\uparrow}) = \mathsf{val}(m)$$

Similarly the left-hand-side of Equation (4.7) can be rewritten as:

$$\begin{aligned}\downarrow^{TB} \mathit{let}(\uparrow^{TA} n, \lambda s.[\![m]\!]_{\uparrow[x\mapsto s]}) &= \mathit{collect}(\lambda().\mathsf{val}(\downarrow^{B} ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ ((\uparrow^{TA} n)\ ()))\ ()))\\ &= \mathit{collect}(\lambda().\mathsf{val}(\downarrow^{B} ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ (\mathit{bind}^{TA}(n)))\ ()))\end{aligned}$$

Thus *bind* and *collect* must satisfy the following equations:

$$\mathit{collect}(\lambda().\mathsf{val}(m)) = \mathsf{val}(m) \tag{4.8}$$

$$\mathit{collect}(\lambda().\mathsf{val}(\downarrow^{B} ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ (\mathit{bind}^{TA}(n))\ ()))) = \mathsf{let}\ x^{A} \Leftarrow n^{TA}\ \mathsf{in}\ \downarrow^{TB} [\![m]\!]_{\uparrow} \tag{4.9}$$

Remark As well as these equations being satisfied it is also necessary that the side-effects are suitably constrained. For example, the overall behaviour of the program should not leave any "dangling" side-effects. We shall not dwell on precisely what the constraints on side-effects should be, but claim that our side-effects are well-behaved. Any side-effects we introduce are always consumed and none of our side-effects are externally visible (unlike I/O operations, for instance).

The *bind* and *collect* parameters can be instantiated (rather concisely) using delimited continuations, or alternatively using global state.

Using delimited continuations to manage the bindings:

$$\begin{aligned}&\mathit{collect}(f) = {<}f{>}\\ &\mathit{bind}^{TA}(e) = \mathcal{S}(\lambda\kappa.\mathsf{let}\ x^{A} \Leftarrow e\ \mathsf{in}\ {<}\lambda().\kappa(\uparrow^{A} x){>}), \quad x \text{ fresh}\end{aligned}$$

Using state we can implement the bindings directly as an updateable list *bindings*.

```
$collect(f) =$
    let $bs_0 = (!bindings)$
    $bindings := \langle\rangle$
    let $e = f()$
    let $e' = wrap(rev(!bindings), e)$
    $bindings := bs_0$
    return $e'$

$bind^{TA}(e) =$
    $bindings := (x^{A}, e) :: (!bindings)$
```

return $\uparrow^A x$, where x is fresh

$$wrap(\langle\rangle, e) = e$$
$$wrap((x^A, m) :: bs, e) = \mathsf{let}\ x^A \Leftarrow m\ \mathsf{in}\ wrap(bs, e)$$

- The auxiliary function $wrap(bs, e)$ outputs the bindings bs and the term e, as a nested sequence of lets.
- $collect(f)$ saves the bindings and resets them to be empty. Then $f()$ is invoked and the result bound to e, which may have the side-effect of creating new bindings. Any new bindings are wrapped around e. (Recall from §2.1 that *rev* simply reverses a list.) Before exiting, the bindings are restored to their initial value.
- $bind^{TA}(e)$ creates a new binding (x^A, e).

Remark Because of the save / restore pattern in *collect*, the initial value (before running the algorithm) of *bindings* is unimportant. In our implementations it is the empty list.

We now outline how to verify that both the continuations-based and state-based versions of *bind* and *collect* satisfy (4.8) and (4.9).

Continuations We make use of Kameyama and Hasegawa's axiomatisation of shift and reset as described in §4.3.1. Recall that values in the metalanguage (§4.3.1) are given by variables, constants and lambda abstractions. In particular, observe that object language terms can be regarded as constants. Similarly object language contexts can be regarded as lambda abstractions. For instance:

- $\mathsf{app}(x, y)$ is an object language term, and a metalanguage constant.
- $\mathsf{val}([\ ])$ is an object language context, and a metalanguage lambda abstraction: $\lambda s.\mathsf{val}(s)$.

This observation enables us to apply Kameyama and Hasegawa's axioms to metalanguage terms which contain object language terms (and contexts).

(4.8):

$$\begin{aligned}
&\mathit{collect}(\lambda().\mathsf{val}(m))\\
&= <\lambda().\mathsf{val}(m)>\\
&= \mathsf{val}(m)\\
&\qquad\text{(reset-value)}
\end{aligned}$$

(4.9):

$$\begin{aligned}
&\mathit{collect}(\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ (\mathit{bind}^{TA}(n))\ ())))\\
&= <\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ (\mathit{bind}^{TA}(n))\ ()))>\\
&= <\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ (\mathcal{S}(\lambda\kappa.\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ k(\uparrow^A x)))\ ()))>\\
&= <\lambda().(\lambda\kappa.\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ \kappa(\uparrow^A x))\ (\lambda r.<\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ r\ ()))>)>\\
&\qquad\text{(reset-}\mathcal{S}\text{, using that } \mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ [\ ]\ ()))\\
&\qquad\text{is a pure evaluation context of the form } v\ (v\ ((v\ [\ ])\ m)))\\
&= <\lambda().\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ (\lambda r.<\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ r\ ()))>)\ (\uparrow^A x))>\\
&= <\lambda().\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ <\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![m]\!]_{\uparrow[x\mapsto s]})\ (\uparrow^A x)\ ()))>>\\
&\qquad(\beta_v\text{, using that } \uparrow^A x \text{ is always a value})\\
&= <\lambda().\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ <\lambda().\mathsf{val}(\downarrow^B ([\![m]\!]_{\uparrow}\ ()))>>\\
&\qquad(\beta_v\text{, again using that } \uparrow^A x \text{ is always a value})\\
&= <\lambda().\mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ \downarrow^{TB} [\![m]\!]_{\uparrow}>
\end{aligned}$$

Unfortunately, we are left with the outer enclosing reset. We claim that, although it uses side-effects internally, the function *norm* has no externally visible side-effects (including non-termination). We leave the formalisation and proof of this claim as future work. Assuming our claim holds, then the contents of the reset evaluates to a value; hence by (reset-value) the reset can be removed and (4.9) follows.

Global state Here, the equational rules are more complex. We omit the details, because they are somewhat involved, but in principle this could be formalised.

(4.8):

$\mathit{collect}(\lambda().\mathsf{val}(m))$

```
=
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(m)$
        let $e' = wrap(rev(!bs), e)$
        $bs := bs_0$
        return $e'$
=
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(m)$
        let $e' = wrap(rev(\langle\rangle), e)$
        $bs := bs_0$
        return $e'$
=
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(m)$
        let $e' = e$
        $bs := bs_0$
        return $e'$
=
        let $e = \mathsf{val}(m)$
        let $e' = e$
        return $e'$
=
        $\mathsf{val}(m)$
```

(4.9):

```
    $collect(\lambda().\mathsf{val}(\downarrow^B ((\lambda s.[\![ m ]\!]_{\uparrow[x \mapsto s]}) (bind^{TA}(n)) ())))$
=
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(\downarrow^B ((\lambda s.[\![ m ]\!]_{\uparrow[x \mapsto s]}) (bind^{TA}(n)) ()))$
        let $e' = wrap(rev(!bs), e)$
        $bs := bs_0$
        return $e'$
=
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $s = bind^{TA}(n)$
        let $e = \mathsf{val}(\downarrow^B ([\![ m ]\!]_{\uparrow[x\mapsto s]}\ ()))$
        let $e' = wrap(rev(!bs), e)$
        $bs := bs_0$
        return $e'$
$=$
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        $bs := (x^A, n) :: (!bs)$
        let $s = \uparrow^A x$
        let $e = \mathsf{val}(\downarrow^B ([\![ m ]\!]_{\uparrow[x\mapsto s]}\ ()))$
        let $e' = wrap(rev((!bs)), e)$
        $bs := bs_0$
        return $e'$
$=$
        let $bs_0 = (!bs)$
        $bs := \langle(x^A, n)\rangle$
        let $e = \mathsf{val}(\downarrow^B ([\![ m ]\!]_{\uparrow}\ ()))$
        let $e' = wrap(rev(!bs), e)$
        $bs := bs_0$
        return $e'$
$=$
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(\downarrow^B ([\![ m ]\!]_{\uparrow}\ ()))$
        let $e' = wrap((x^A, n) :: rev(!bs), e)$
        $bs := bs_0$
        return $e'$
$=$
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(\downarrow^B ([\![ m ]\!]_{\uparrow}\ ()))$
        let $e' = \mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ wrap(rev(!bs), e)$
        $bs := bs_0$
        return $e'$
$=$
        let $bs_0 = (!bs)$
        $bs := \langle\rangle$
        let $e = \mathsf{val}(\downarrow^B ([\![ m ]\!]_{\uparrow}\ ()))$
        let $e' = wrap(rev(!bs), e)$
        $bs := bs_0$
```

$$\begin{array}{ll} & \text{return } \mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ e' \\ = & \\ & \mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ \mathit{collect}(\lambda().\mathsf{val}(\downarrow^B ([\![m]\!]_\uparrow ()))) \\ = & \\ & \mathsf{let}\ x^A \Leftarrow n\ \mathsf{in}\ \downarrow^{TB} [\![m]\!]_\uparrow \end{array}$$

It is interesting to note how much longer the derivations are here (even without all the details), than in the case of delimited continuations or in the case of monads. This highlights the gap between reasoning about functional programs and reasoning about imperative programs.

Relation to monadic reflection The algorithm with delimited continuations is directly justified by Filinski's monadic reflection operations. The idea is that monadic values can be *reflected* (μ) as side-effecting computations, and conversely (delayed) side-effecting computations can be *reified* ($[\cdot]$) as monadic values. In particular for the continuation monad, these operations are defined as:

$$\begin{aligned} \mu(c) &= \mathcal{S}(c) \\ [t] &= \lambda\kappa.<\lambda().\kappa(t())> \end{aligned}$$

We have essentially used μ to reflect a computation (in $\uparrow^{TA}$), and then used $[\cdot]$ to reify the computation (in $\downarrow^{TA}$). μ introduces side-effects and $[\cdot]$ consumes them. It is sound to use this technique to transform a pure program using monads into a side-effecting program providing that whenever side-effects are introduced they are also consumed. Using a thunk to encapsulate the computation, allows us to force the evaluation of the computation once we know that the side-effects are going to be consumed.

Note that the thunks are necessary even if the side-effect is not delimited continuations. For instance, consider the term $\mathsf{lam}(x^{TA}, c^{\mathbf{1}})$, where c is a constant. Calling reify at type $TA \to \mathbf{1}$ calls reflect at type TA. Now, this is where the side-effect must register a binding. But there is no call to collect, so this side-effect cannot be consumed. It is unsound to leave side-effects dangling like this. If, on the other hand, the computation had been delayed, then it would never be evaluated (as this could happen only if collect were called).

Turning things on their head, we can view the use of monadic operations in the definition of the semantics as a technique for transferring (side-effecting) functionality from $\downarrow$ into the residualising semantics[1]. It is not particularly surprising that side-effects and suitable (pure) monads are interchangeable: Moggi's original motivation for using monads was to model computational effects. What is remarkable is that if we use side-effects then they need only be introduced in the definition of $\downarrow$.

Our monadic normalisation by evaluation algorithms for the computational metalanguage are similar to Filinski's normalisation by evaluation algorithms for the computational λ-calculus [Fil01b]. The shift and reset operators have long been used to implement partial evaluation [LD94]. Sumii and Kobayashi showed how to use state instead [SK01]. Danvy [Dan98] presents TDPE using delimited continuations, then using state. These algorithms are similar to our effect-based normalisation by evaluation algorithms for the computational metalanguage.

4.3.3 Restriction to λ_{ml*}

If we restrict ourselves to λ_{ml*}, then the typing restriction ensures that each computation will be used, and by a judicious repositioning of the *collect* operation we can remove the need for computations to be delayed. The monadic parameters then become:

$$\begin{aligned}
Comp(\llbracket A \rrbracket) &= {}^{\circ}\llbracket A \rrbracket \\
val(s) &= s \\
let(val(s), f) &= f\ s
\end{aligned}$$

and the normalisation by evaluation algorithm becomes:

[1]This pattern of transferring functionality between $\downarrow$ and the residualising semantics works both ways, and is a useful tool in the application of normalisation by evaluation.

$$\begin{aligned}
\downarrow^{A} &: [\![A]\!] \to \Lambda\mathit{ml\text{-}nf}_{A} \\
\uparrow^{A} &: \Lambda\mathit{ml\text{-}ne}_{A} \to [\![A]\!]
\end{aligned}$$

$$\begin{aligned}
\downarrow^{O} e &= e \\
\downarrow^{A\to B} f &= \mathsf{lam}(x^{A}, \mathit{collect}(\lambda().\downarrow^{B}(f(\uparrow^{A} x)))), \quad x \text{ fresh} \\
\downarrow^{TA} s &= \mathsf{val}(\downarrow^{A} s())
\end{aligned}$$

$$\begin{aligned}
\uparrow^{O} e &= e \\
\uparrow^{A\to B} e &= \lambda s^{A}.\uparrow^{B}(\mathsf{app}(e, (\downarrow^{A} s))) \\
\uparrow^{TA} e &= \mathit{bind}^{TA}(e) \\
\mathit{norm}(e^{A}) &= \mathit{collect}(\lambda().\downarrow^{A}([\![e]\!]_{\uparrow}))
\end{aligned}$$

The only place a computation can be introduced is through a top-level function application or when applying the function inside $\downarrow^{A\to B}$. In both cases the side-effects are consumed by *collect*.

4.4 Alternatives to η-expansion

Reduction calculi which include η-expansions but not η-reductions have a number of desirable properties. In particular they are usually confluent, whilst corresponding calculi with some η-contractions in place of expansion are not. For instance, consider $\Lambda^{\times\mathbf{1}}$ with the reduction rules:

$$\begin{array}{rrcll}
(\to.\beta) & \mathsf{app}(\mathsf{lam}(x,m),n) & \longrightarrow & m[x := n] & \\
(\to.\eta) & \mathsf{lam}(x,\mathsf{app}(m,x)) & \longrightarrow & m, & \text{if } x \notin \mathit{fv}(m) \\
(\times.\beta i) & \mathsf{proj}_{i}(\mathsf{pair}(m_1,m_2)) & \longrightarrow & m_i & \\
(\times.\eta) & \mathsf{pair}(\mathsf{proj}_1(m),\mathsf{proj}_2(m)) & \longrightarrow & m & \\
(\mathbf{1}.\eta) & m & \longrightarrow & * &
\end{array}$$

Now:

$$\mathsf{lam}(x, \mathsf{app}(f^{A \to \mathbf{1}}, x)) \longrightarrow \mathsf{lam}(x, *)$$

which is irreducible, and hence normal, but:

$$\mathsf{lam}(x, \mathsf{app}(f^{A \to \mathbf{1}}, x)) \longrightarrow f$$

which is also irreducible, and hence normal. Thus this calculus is not confluent[2].

Recall from §2.2.3 that $\mathbf{1}.\eta$ is an expansion. It might seem natural to have a calculus with only η-contractions. But this does not appear to be possible in the case of $\mathbf{1}$. If $\to.\eta$ and $\times.\eta$ are read as expansions instead of reductions then this does lead to a confluent calculus.

Directing η-rules as η-expansions generally gives rise to simple syntactic characterisations of normal forms. In turn, this gives rise to rather natural normalisation by evaluation algorithms. In contrast, directing η-rules as η-contractions, leads to somewhat unnatural side-conditions on the structural rules for normal forms. In turn these manifest themselves as more complicated normalisation by evaluation algorithms. To illustrate the difference in normal forms consider normal forms for simply-typed λ-calculus. Recall from §2.4.3 that long normal forms (which arise from η-expansions) are given by:

Normal forms	$m ::= n^O \mid \mathsf{lam}(x^A, m)$
Neutral terms	$n^A ::= x^A \mid \mathsf{app}(n^{B \to A}, m)$

Normal forms for the calculus with no η-rules are the same except a neutral term of any type (not just O) is also a normal form:

Normal forms	$m ::= n^A \mid \mathsf{lam}(x^A, m)$
Neutral terms	$n^A ::= x^A \mid \mathsf{app}(n^{B \to A}, m)$

Normal forms for the calculus with η-contraction in place of η-expansion are not so easily characterised. They are the same as for the calculus without η-rules, but with

[2]Curien and Di Cosmo [CD91] show how to make this calculus confluent by adding a family of rewrite rules arising from type-isomorphisms involving $\mathbf{1}$.

a side-condition on the body of lambda abstractions — namely subterms of the form $\mathsf{lam}(x^A, \mathsf{app}(n, x))$ such that $x \notin \mathit{fv}(n)$ are not admitted.

Despite the advantages of η-expansion, there are also disadvantages. The obvious disadvantage is the blow-up in the size of normal forms — the bigger the types, the bigger the normal form. This can be particularly problematic for sums, as it can lead to an exponential increase in the size of terms[3]. Recursive types are even more problematic. η-expansion does not terminate for recursive types (see §4.5).

One option is to perform η-contraction instead. An easier alternative is to suppress η-expansion, and not perform any η-contraction. This has the advantage of simplicity and reducing blow-up in the size of terms, but the disadvantage of not identifying as many terms as in the presence of η-rules. For more complicated calculi (involving sums, for instance), it is sometimes necessary to apply η-rules in both directions in order to obtain canonical normal forms.

In Chapter 2 we have already seen examples of normalisation by evaluation which do not perform η-expansion. We illustrate how such an algorithm can be derived from the standard normalisation by evaluation algorithm for simply-typed λ-calculus using simple program transformations. We then show how through further program transformation it is possible to incorporate η-contractions.

Terminology We refer to normalisation by evaluation algorithms which perform normalisation with respect to all conversion rules (β, η and CC) as *extensional* or *η-NBE* algorithms. We refer to normalisation by evaluation algorithms which perform normalisation with respect to just β- and CC-rules as *intensional* or *β-NBE* or *βCC-NBE* algorithms.

4.4.1 Suppressing η-expansion

We begin with the normalisation by evaluation algorithm of §2.5.3 using the parameterised semantics of Figure 2.11. The first step is to *defunctionalise* [Rey98, DN01] the reflection function. Defunctionalisation is a program transformation which converts

[3]In fact sums are problematic for other reasons. Even with η-expansion and the usual reduction rules, we do not obtain a confluent reduction calculus.

$$
\begin{aligned}
[\![O]\!] &= \Lambda^{\rightarrow}_{O} \\
[\![A \rightarrow B]\!] &= ([\![A]\!] \rightarrow [\![B]\!]) + \Lambda^{\rightarrow}_{A\rightarrow B} \\[1em]
app(f, s) &= f\ s \\
app(e, s) &= \mathsf{app}(e, \downarrow s) \\[1em]
\downarrow^{A} &: [\![A]\!] \rightarrow \Lambda^{\rightarrow}\text{-}nf_{A} \\
\uparrow^{A} &: \Lambda^{\rightarrow}\text{-}ne_{A} \rightarrow [\![A]\!] \\[1em]
\downarrow^{O} e &= e \\
\downarrow^{A\rightarrow B} f &= \mathsf{lam}(x^{A}, \downarrow^{B} (app(f, \uparrow^{A} x))), \quad x^{A} \text{ fresh} \\[1em]
\uparrow^{O} e &= e^{O} \\
\uparrow^{A\rightarrow B} e &= e^{A\rightarrow B} \\[1em]
norm(e^{A}) &= \downarrow^{A} ([\![e]\!]_{\uparrow})
\end{aligned}
$$

Figure 4.1: NBE for $\lambda^{\rightarrow}$ with defunctionalised reflection

higher order programs into first-order programs. Typically one defunctionalises the whole program. Banerjee et al. [BHR01] give a correctness proof for whole-program defunctionalisation. In our case we only defunctionalise the λ-abstraction introduced by the reflection function at function type:

$$\uparrow^{A\rightarrow B} e = \lambda s. \uparrow^{B} (\mathsf{app}(e, (\downarrow^{A} s)))$$

The free (meta)variables in this λ-abstraction are e, A, B. Thus, this abstraction can be encoded by the term $e^{A\rightarrow B}$. Wherever this function is applied, in the original program, the application is replaced, in the transformed program, by the body of the abstraction with appropriate values substituted in for the free and bound variables:

$$
\begin{aligned}
[\![O]\!] &= \Lambda^{\rightarrow}_{O} \\
[\![A \rightarrow B]\!] &= ([\![A]\!] \rightarrow [\![B]\!]) + \Lambda^{\rightarrow}_{A\rightarrow B}
\end{aligned}
$$

$$
\begin{aligned}
app(f, s) &= f\ s \\
app(e, s) &= \mathsf{app}(e, \downarrow s)
\end{aligned}
$$

$$
\begin{aligned}
\downarrow^{A} &: [\![A]\!] \rightarrow \Lambda^{\rightarrow}\text{-}nf_{A} \\
\uparrow^{A} &: \Lambda^{\rightarrow}\text{-}ne_{A} \rightarrow [\![A]\!]
\end{aligned}
$$

$$
\begin{aligned}
\downarrow^{O} e &= e \\
\downarrow^{A\rightarrow B} f &= \mathsf{lam}(x^{A}, \downarrow^{B} (f\ x^{A})), \quad x^{A} \text{ fresh} \\
\downarrow^{A\rightarrow B} e &= \mathsf{lam}(x^{A}, \downarrow^{B} (\mathsf{app}(e, \downarrow^{A} x^{A}))), \quad x^{A} \text{ fresh}
\end{aligned}
$$

$$
\begin{aligned}
\uparrow^{O} e &= e^{O} \\
\uparrow^{A\rightarrow B} e &= e^{A\rightarrow B}
\end{aligned}
$$

$$
norm(e^{A}) = \downarrow^{A} ([\![e]\!]_{\uparrow})
$$

Figure 4.2: NBE for $\lambda^{\rightarrow}$ with explicit η-expansion

$$
\uparrow^{B} (\mathsf{app}(e, (\downarrow^{A} s)))
$$

The definition of $[\![A \rightarrow B]\!]$ is modified to contain terms of type $A \rightarrow B$ (arising from calling the reflect function at type $A \rightarrow B$) in addition to functions from $[\![A]\!]$ to $[\![B]\!]$. Correspondingly the *app* parameter is extended to handle application of terms as well as functions. The algorithm appears in Figure 4.1. We distinguish between terms and functions in the semantics by letting e range over terms and f over functions.

Defunctionalising reflect has the effect of delaying the reflection operation until the resulting value is used. Inlining reflection and application gives the algorithm of Figure 4.2. It now becomes apparent exactly where η-expansion takes place. Reifying

$$[\![\, \Lambda u \,]\!] \cong \Lambda u\text{-}ne + ([\![\, \Lambda u \,]\!] \to [\![\, \Lambda u \,]\!])$$

$$\begin{aligned} app(f,s) &= f\ s \\ app(e,s) &= \mathsf{app}(e, \downarrow s) \end{aligned}$$

$$\begin{aligned} \downarrow &: [\![\, \Lambda u \,]\!] \to \Lambda u\text{-}nf \\ \uparrow &: \Lambda u\text{-}ne \to [\![\, \Lambda u \,]\!] \end{aligned}$$

$$\begin{aligned} \downarrow e &= e \\ \downarrow f &= \mathsf{lam}(x, \downarrow (f\ x)), \quad x \text{ fresh} \end{aligned}$$

$$\uparrow e = e$$

$$norm(e) = \downarrow ([\![\, e \,]\!]_{\uparrow})$$

Figure 4.3: The standard β-NBE algorithm for untyped λ-calculus

a term at function type η-expands it.

Note that the semantics after defunctionalisation is no longer sound with respect to $\beta\eta$-conversion (extensional), but is sound with respect to just β-conversion (intensional). We can reintroduce extensionality by defining the following equation on semantic objects:

$$e^A = \uparrow^A e \tag{4.10}$$

where $\uparrow$ is defined as in the original algorithm:

$$\uparrow^A : \Lambda^{\to}\text{-}ne_A \to [\![\, A \,]\!]$$

$$\begin{aligned} \uparrow^O e &= e \\ \uparrow^{A\to B} e &= \lambda s.\ \uparrow^B (\mathsf{app}(e, (\downarrow^A s))) \end{aligned}$$

Alternatively the η-expansion clauses can be stripped out of the reification function:

$$\begin{aligned}\downarrow^{O} e &= e \\ \downarrow^{A\to B} f &= \mathsf{lam}(x^{A}, \downarrow^{B} (f\ x)), \quad x \text{ fresh}\end{aligned}$$

$$norm(e^{A}) = \downarrow^{A} (\llbracket e \rrbracket_{\uparrow})$$

This gives a β-NBE algorithm for simply-typed λ-calculus. Observe that the type parameters are redundant and we can coalesce the semantics into a single untyped domain S:

$$S \simeq \Lambda^{\to} + (S \to S)$$

Then by simply erasing the type annotations we obtain the usual β-NBE algorithm for untyped λ-calculus. This algorithm is part of normalisation by evaluation folklore but has only recently been studied formally. Aehlig and Joachimski [AJ04] use rewriting theory. Filinski and Rohde [FR04] give a domain theoretic account. The algorithm appears in Figure 4.3.

The transformation into the form of Figure 4.2 can easily be extended to other type constructors. The general method is:

- Extend the semantics of each type to include terms of that type.
- Modify each elimination parameter to handle syntax.
- Define reflection as the identity.
- Introduce an η-expansion clause for each type constructor in the reification function.

An intensional NBE algorithm is then obtained by dropping all of the η-expansion clauses.

4.4.2 Performing η-contraction

One way of modifying a normalisation by evaluation algorithm to perform η-contraction is simply to check whenever an η-redex is created in the output and reduce it. The only redexes these η-contractions can create are more η-redexes. There

$$\llbracket \Lambda u \rrbracket \cong (\Lambda u + (\mathbf{V} \to \Lambda u)) \times (\llbracket \Lambda u \rrbracket \to \llbracket \Lambda u \rrbracket)$$

$$\begin{aligned} \mathit{lam}(f) &= (\lambda x.\mathsf{lam}(x, \downarrow (f(\uparrow x))), f) \\ \mathit{app}((\cdot, f), s) &= f\ s \end{aligned}$$

$$\begin{aligned} \downarrow (e, f) &= e \\ \downarrow (g, f) &= g\ x, \quad x \text{ fresh} \end{aligned}$$

$$\uparrow e = (e, \lambda v. \uparrow \mathsf{app}(e, v))$$

$$\mathit{norm}(e) = \downarrow (\llbracket e \rrbracket_{\uparrow})$$

Figure 4.4: Glued β-NBE for untyped λ-calculus

are two problems with this approach. The first is that η-contraction takes place in $\downarrow$ and outside of the semantics, and hence the semantics is not sound with respect to η-contraction. From a practical perspective this is not a problem, but one might hope that it would be possible to ‘move the η-contraction back into the semantics’. The second problem is that the naïve way of detecting $\to.\eta$-redexes is potentially rather inefficient. It involves traversing the entire body of the relevant lambda in order to establish that the bound variable only occurs once.

We solve the first problem by moving to a semantics in which every semantic component always has a syntactic representation *glued* to it. This is reminiscent of Coquand and Dybjer’s take on normalisation by evaluation for simply-typed combinatory logic [CD97]. The algorithm appears in Figure 4.4.

An easy way of showing that the glued algorithm is equivalent to the standard one is to defunctionalise both versions and observe that the resulting algorithm is the same in each case (Figure 4.5).

We can adjust the algorithm of Figure 4.4 to perform η-contraction simply by modifying the abstraction parameter as follows:

$$[\![\Lambda u]\!] \cong \Lambda u + (\mathbf{V} \times \Lambda u \times (\mathbf{V} \to [\![\Lambda u]\!]))$$

$$\begin{aligned}
\downarrow e &= e \\
\downarrow c &= \mathsf{lam}(x, \downarrow app(c, x)), \quad x \text{ fresh}
\end{aligned}$$

$$\begin{aligned}
[\![x]\!]_\rho &= \rho\ x \\
[\![\mathsf{lam}(x, m)]\!]_\rho &= (x, m, \rho) \\
[\![\mathsf{app}(m, n)]\!]_\rho &= app([\![m]\!]_\rho, [\![n]\!]_\rho)
\end{aligned}$$

$$\begin{aligned}
app(e, s) &= \mathsf{app}(e, \downarrow s) \\
app((x, e, \rho), s) &= [\![e]\!]_{\rho[x \mapsto s]}
\end{aligned}$$

Figure 4.5: Defunctionalised β-NBE for untyped λ-calculus

$$lam(f) = (\lambda x.eta\text{-}reduce(\mathsf{lam}(x, \downarrow (f(\uparrow x)))), f)$$

$$\begin{aligned}
eta\text{-}reduce(\mathsf{lam}(x, \mathsf{app}(m, x))) &= m, &\text{if } x \notin fv(m) \\
eta\text{-}reduce(\mathsf{lam}(x, \mathsf{app}(m, x))) &= \mathsf{lam}(x, \mathsf{app}(m, x)), &\text{otherwise}
\end{aligned}$$

The second problem can be solved by augmenting the semantics with variable count information. We omit the details, but we have implemented such an algorithm in ML. A simpler approach, which may be more efficient in practice would be to rely on the invariant that bound variables are unique, and maintain a global variable count.

It is straightforward to extend our approach to η-contraction in order to handle other type constructors such as computations and products. In fact, functions are harder to deal with than other constructors, as the $\to.\eta$-rule depends on counting occurrences, whereas other η-rules do not. In practice, it may be easier simply to perform the η-reduction as part of the reify function, rather than using the gluing construction (though technically this is not normalisation by evaluation).

4.5 Sums

Adding sums to normalisation by evaluation algorithms is an interesting non-trivial problem. The primary difficulty is in defining a suitable normal form. If we forget about η-conversion then this solves the problem. However, the η-rule allows us to perform some rather useful optimisations such as:

In the presence of the η-rule we cannot obtain unique normal forms for all convertible terms simply by applying all the rules in one direction. This is a good example of a situation in which the 'reduction-free' nature of normalisation by evaluation is important. In other words it makes more sense to think about equational normal forms rather than reduction-based normal forms (although we believe it is possible to extend the reduction calculus in such a way that the two notions can be made to coincide).

4.5.1 Greedy elimination of sums

Perhaps the simplest solution to the problem of sums is the one adopted by the TDPE community. This approach is motivated by ease of implementation using the shift/reset control operators. It relies on sum terms always being bound to a variable. This rules out the simply-typed λ-calculus extended with sums, as here we have terms such as:

$$\mathsf{lam}(f^{A\to(B+C)}, \mathsf{lam}(x^A, \mathsf{app}(f, x)))$$

where the application is of sum type. However the restricted version of the computational metalanguage λ_{ml*} and the computational λ-calculus can easily be made to satisfy this property. Also this approach does not handle sum constants or open terms (although both of these features can easily be simulated by an extra abstraction). The idea is that a case split is introduced immediately inside each context in which a sum variable is bound. The resulting normal forms are relatively easy to define (compared to the alternatives). Filinski formalises the idea in the setting of the computational λ-calculus [Fil01b].

The approach has the distinct disadvantage of producing normal forms which are exponentially large in the number of nested sum variables, and potentially contain a large amount of redundancy. It is not dissimilar to representing a boolean formula as a truth table. Balat and Danvy [BD02] show how to eliminate some of the redundancy by using memoisation to prevent nested case splits on the same guard.

In his thesis Ghani used categorical techniques to study normalisation properties of the simply-typed λ-calculus extended with sums [Gha95].

Altenkirch et al. [ADHS01] give a characterisation of normal forms for simply-typed λ-calculus with sums using an extended calculus supporting multiple simultaneous case splits.[4] A key advantage of these normal forms is that they remove a certain amount of redundancy. Altenkirch et al. define and prove correct a normalisation algorithm for obtaining their normal forms. Unfortunately it is not immediately clear how to implement their algorithm as it depends on abstract categorical concepts. Nevertheless we have implemented it using a modified version of the standard TDPE approach. We have also implemented variations which do not depend on first-class control operators.

Balat, Di Cosmo and Fiore [BCF04] have recently given an alternative treatment of Altenkirch et al.'s approach in which rather than extending the calculus they simply quotient out the order in which case splits occur. They also provide an implementation which takes advantage of the more general set/cupto [GRR98] control operators instead of shift/reset.

We now give details of how to incorporate sums into normalisation by evaluation algorithms. We shall concentrate on the basic approach of Filinski, but indicate how it can be adapted to remove redundancy. We also present a version in which we use global state and an appropriate data structure in place of control operators.

We call the language $\Lambda ml*$ extended with sums and products $\Lambda ml*^{+}$. We call the corresponding equational calculus (with all the usual β, η and CC rules) λ^{+}_{ml*}. There are several choices as to how to add sums and products into $\Lambda ml*$. In particular we can choose whether the associated elimination terms must be computations or not. The advantage of asserting that eliminations must be of computation type, is that this ensures intermediate subterms are named. We insist that *case* terms must have computation type. It would seem natural to do the same for projections, but as projections do not involve any auxiliary subterms this would require changing rather than restricting our

[4]Ghani hinted in his thesis [Gha95] that multiple simultaneous case splits might be useful.

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\mathsf{lam}(x^A, e)]\!]_\rho &= lam(\lambda s^A.[\![e]\!]_{\rho[x\mapsto s]}) \\
[\![\mathsf{app}(e_1, e_2)]\!]_\rho &= app([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![\mathsf{val}(e)]\!]_\rho &= val([\![e]\!]_\rho) \\
[\![\mathsf{let}\ x^A \Leftarrow e_1\ \mathsf{in}\ e_2]\!]_\rho &= let([\![e_1]\!]_\rho, \lambda v^A.[\![e_2]\!]_{\rho[x\mapsto v]}) \\
[\![\mathsf{inj}_1(e)]\!]_\rho &= inj_1([\![e]\!]_\rho) \\
[\![\mathsf{inj}_2(e)]\!]_\rho &= inj_2([\![e]\!]_\rho) \\
[\![\mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)]\!]_\rho &= case\begin{pmatrix} [\![m]\!]_\rho, \\ \lambda v_1.[\![n_1]\!]_{\rho[x_1\mapsto v_1]}, \\ \lambda v_2.[\![n_2]\!]_{\rho[x_2\mapsto v_2]} \end{pmatrix}
\end{aligned}$$

Figure 4.6: Parameterised semantics for $\Lambda ml{*}^+$

existing typing rules. Also note that MIL treats projections as values. Thus the typing rules for $\Lambda ml{*}^+$ are the same as those of $\Lambda ml{*}$ plus those of Figure 2.3 with the added restriction that case terms must have computation type:

$$\frac{x_1 : A_1 \quad x_2 : A_2 \quad m : A_1 + A_2 \quad n_1 : TB \quad n_2 : TB}{\mathsf{case}\ m\ \mathsf{of}\ (x_1^{A_1} \Rightarrow n_1 \mid x_2^{A_2} \Rightarrow n_2) : TB}$$

A parameterised semantics for $\Lambda ml{*}^+$ appears in Figure 4.6.

Filinski defines normal forms using a *quarantined context* [Fil01b]. This ensures that case splits are performed on sum variables as soon as they are introduced. We achieve the same goal for λ^+_{ml*} using a grammar annotated with types and bound variables:

Normal values	$v, w ::= u^0 \mid \mathsf{lam}(x^A, m[x^A]) \mid \mathsf{pair}(v, w) \mid \mathsf{inj}_1(v) \mid \mathsf{inj}_2(v)$
Neutral values	$u^A ::= x^A \mid \mathsf{proj}_1(u^{A\times B}) \mid \mathsf{proj}_2(u^{B\times A})$
Split computations	$m[x^{A+B}] ::= \mathsf{case}\ x\ \mathsf{of}\ (x_1^A \Rightarrow m[x_1^A] \mid x_2^B \Rightarrow m[x_2^B])$
Plain computations	$m[x^A] ::= \mathsf{val}(v) \mid \mathsf{let}\ y^B \Leftarrow \mathsf{app}(u, v)\ \mathsf{in}\ m[y^B]$

The set of normal forms $\Lambda ml{*}^+$-*nf* is the union of the normal values, the split computations and the plain computations. The set of neutral terms $\Lambda ml{*}^+$-*ne* is just the set of neutral values.

The residualising semantics is standard:

$$\begin{aligned}
[\![O]\!] &= \Lambda ml_O \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![A + B]\!] &= [\![A]\!] + [\![B]\!] \\
[\![TA]\!] &= Comp([\![A]\!])
\end{aligned}$$

We use the internal monad of the metalanguage:

$$\begin{aligned}
Comp([\![A]\!]) &= {}^{\circ}[\![A]\!] \\
val(s) &= s \\
let(val(s), f) &= f\ s
\end{aligned}$$

and the method of §4.3.3 in order to avoid the need for thunks:

$$\begin{aligned}
\downarrow^{O} e &= e \\
\downarrow^{A \to TB} f &= \mathsf{lam}(x^A, collect(\lambda().\downarrow^{TB} (f(\uparrow^{A} x)))), \quad x^A \text{ fresh} \\
\downarrow^{A \times B} (s, s') &= \mathsf{pair}(\downarrow^{A} s, \downarrow^{B} s') \\
\downarrow^{A_1 + A_2} inj_i(s) &= \mathsf{inj}_i(\downarrow^{A_i} s) \\
\downarrow^{TA} s &= \mathsf{val}(\downarrow^{A} s)
\end{aligned}$$

$$\begin{aligned}
\uparrow^{O} e &= e \\
\uparrow^{A \to TB} e &= \lambda s^A. \uparrow^{TB} (\mathsf{app}(e, (\downarrow^{A} s))) \\
\uparrow^{A \times B} e &= (\uparrow^{A} \mathsf{proj}_1(e), \uparrow^{B} \mathsf{proj}_2(e)) \\
\uparrow^{A_1 + A_2} e &= bind^{A_1 + A_2}(e) \\
\uparrow^{TA} e &= bind^{TA}(e)
\end{aligned}$$

$$norm(e^{TA}) = collect(\lambda(). \downarrow^{TA} ([\![e]\!]_{\uparrow}))$$

The *bind* and *collect* functions must satisfy the additional equation:

$$\begin{aligned}
&collect(\lambda(). \\
&\quad \mathsf{val}(\downarrow^{B} (\lambda s.\mathsf{case}\ s\ \mathsf{of}\ s_1 \Rightarrow [\![m_1]\!]_{\uparrow[x_1 \mapsto s_1]} \mid s_2 \Rightarrow [\![m_2]\!]_{\uparrow[x_2 \mapsto s_2]})(bind^{A_1 + A_2}(u)))) \\
&\qquad = \mathsf{case}\ u\ \mathsf{of}\ (x_1^{A_1} \Rightarrow \downarrow^{TB} [\![m_1]\!]_{\uparrow} \mid x_2^{A_2} \Rightarrow \downarrow^{TB} [\![m_2]\!]_{\uparrow}) \qquad (4.11)
\end{aligned}$$

4.5.2 Sums using delimited continuations

Assuming the metalanguage supports delimited continuations we can define *bind* on computations, and *collect* as before, and extend *bind* to sums:

$$
\begin{aligned}
\mathit{collect}(f) &= \langle f \rangle \\
\mathit{bind}^{TA}(e) &= \mathcal{S}(\lambda\kappa.\mathsf{let}\ x^A \Leftarrow e\ \mathsf{in}\ \langle\lambda().\kappa(\uparrow^A x)\rangle), \\
&\quad x \text{ fresh} \\
\mathit{bind}^{A_1+A_2}(e) &= \mathcal{S}(\lambda\kappa.\mathsf{case}\ e\ \mathsf{of}\ (x_1 \Rightarrow \langle\lambda().\kappa(\uparrow^{A_1} x_1)\rangle \mid x_2 \Rightarrow \langle\lambda().\kappa(\uparrow^{A_2} x_2)\rangle)), \\
&\quad x_1, x_2 \text{ fresh}
\end{aligned}
$$

In $\mathit{bind}^{A_1+A_2}(e)$, the shift operation inserts a case split at the start of the current context and then follows both branches of the sum. As usual the reset operator marks the start of a new context in which a variable has been bound.

4.6 Sums using global state

As shown in §4.5.2, sums can be dealt with using a straightforward modification of the normalisation by evaluation algorithm for the computational metalanguage, providing we use delimited continuations. However, it is not immediately clear how to do the same thing in the state-based setting. The challenge is to define *bind* at sum types. We want to return both the left injection and the right injection of a sum. This is easy to do using shift and reset, as the same continuation can be invoked twice, but in the state-based setting we do not have access to first class continuations.

Other authors have also considered using global state (or an accumulation monad) in place of delimited continuations (or a continuation monad).

Filinski [Fil01b] wrote:

> Products could be added to an accumulation-based interpretations without too much trouble, but sums apparently require the full power of applying a single continuation multiple times.

Sumii and Kobayashi [SK01] wrote:

> Note that state-based let-insertion does not subsume continuation-based PE [[LD94]], whose goal is not only context propagation in let-expressions but also context *duplication* in conditional expressions.[footnote]

and then in the footnote:

> It is possible as well to treat conditional expressions by using state instead of continuations (Zhe Yang, personal communication, January 2000), but it remains to see whether this “state-based if-insertion” is correct and efficient, because it is more complex than state-based let-insertion and because it duplicates some static computation.

Our solution is to first return the left injection, but to record (using global state) that we still need to visit the right injection. Rather than a binding list we require a binding tree. Which branch we are currently in is recorded in the binding tree along with the bindings. The binding tree is reset every time *collect* is called. The thunk passed to *collect* is invoked repeatedly. Each invocation builds another path through the binding tree, and returns the computation to be plugged in the hole at the end of that path. Eventually the entire binding tree is assembled, along with a collection of computation terms. The binding tree is output, with each hole plugged by the appropriate computation term.

4.6.1 Binding trees

We describe binding trees using the datatype:

$$\mathcal{B} = \mathsf{hole} \mid \mathsf{comp}\ (e^{TA}, x^{A}, \mathcal{B}) \mid \mathsf{sum}\ (v^{A_1+A_2}, x_1^{A_1}, \mathcal{B}_1, x_2^{A_2}, \mathcal{B}_2, bool)$$

and the plugging operation, which plugs a binding tree *t* containing *k* holes, using a list *es* of *k* computation terms:

$$\begin{aligned} t[es] = {} & \text{let } (e', \_) = plugTree(t, es) \\ & \text{in } e' \end{aligned}$$

where *plugTree* is defined as:

$$\begin{aligned}
\mathit{plugTree}(\mathsf{hole}, e :: es) &= (e, es) \\
\mathit{plugTree}(\mathsf{comp}\ (e, x, t), es) &= \text{let } (e', es') = \mathit{plugTree}(t, es) \\
&\quad\ \text{in } (\mathsf{let}\ x \Leftarrow e\ \mathsf{in}\ e', es') \\
\mathit{plugTree}(\mathsf{sum}\ (v, x_1, t_1, x_2, t_2, \_), es) &= \text{let } (e_1, es') = \mathit{plugTree}(t_1, es) \\
&\qquad\ (e_2, es'') = \mathit{plugTree}(t_2, es') \\
&\quad\ \text{in } (\mathsf{case}\ v\ \mathsf{of}\ (x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2), es'')
\end{aligned}$$

Binding trees are constructed from holes, computation bindings and sum bindings. The plugging operation $t[es]$ is used to obtain the term represented by t with all the holes plugged with terms from es.

It is sometimes useful to grow a tree by plugging the holes with other trees. We overload the plugging operation as follows:

$$\begin{aligned}
t[ts] = {} & \text{let } (t', \_) = \mathit{plugTree}(t, ts) \\
& \text{in } t'
\end{aligned}$$

where *plugTree* is defined as:

$$\begin{aligned}
\mathit{plugTree}(\mathsf{hole}, t :: ts) &= (t, ts) \\
\mathit{plugTree}(\mathsf{comp}\ (e, x, t), ts) &= \text{let } (t', ts') = \mathit{plugTree}(t, ts) \\
&\quad\ \text{in } (\mathsf{comp}\ (e, x, t'), ts') \\
\mathit{plugTree}(\mathsf{sum}\ (v, x_1, t_1, x_2, t_2, b), ts) &= \text{let } (t'_1, ts') = \mathit{plugTree}(t_1, ts) \\
&\qquad\ (t'_2, ts'') = \mathit{plugTree}(t_2, ts') \\
&\quad\ \text{in } (\mathsf{sum}\ (v, x_1, t'_1, x_2, t'_2, b), ts'')
\end{aligned}$$

4.6.2 The zipper structure

We will need to be able to construct the binding tree incrementally by moving up and down the tree structure and modifying nodes. One way to achieve this is to use a mutable data structure. A much cleaner approach, particularly for an implementation in a functional programming language, is to use Huet's 'zipper' structure [Hue97].

A *location* in the tree is represented by a pair of the current subtree and a *path* from the root of the tree to the root of the current subtree. A path has to contain enough information to be able to move anywhere in the tree above the current subtree. Paths are given by the datatype:

$$\begin{aligned}\mathcal{P} &= \mathsf{top}\\ &\mid \mathsf{down}\ ((e^{TA}, x^{A}), \mathcal{P})\\ &\mid \mathsf{left}\ ((v^{A_1+A_2}, x_1^{A_1}, x_2^{A_2}), \mathcal{B}, \mathcal{P})\\ &\mid \mathsf{right}\ ((v^{A_1+A_2}, x_1^{A_1}, x_2^{A_2}), \mathcal{B}, \mathcal{P})\end{aligned}$$

This is essentially the *derivative*, that is the type of one-holed contexts, of the binding tree datatype [McB01][5]. This is not quite the derivative, because the boolean flag has disappeared. The reason why the flag is no longer present is that it is now redundant. The left and right constructors encode which branch a path is in.

The type of locations is $\mathcal{B} \times \mathcal{P}$. The zipper functions allow movement around the tree, and the current subtree to be changed.

$go\_down(\mathsf{comp}\ (e,x,t),p) = \text{return}\ (t, \mathsf{down}\ ((e,x),p))$
$go\_down(\mathsf{sum}\ (v,x_1,t_1,x_2,t_2,true),p) = \text{return}\ (t_1, \mathsf{left}\ ((v,x_1,x_2),t_2,p))$
$go\_down(\mathsf{sum}\ (v,x_1,t_1,x_2,t_2,false),p) = \text{return}\ (t_2, \mathsf{right}\ ((v,x_1,x_2),t_1,p))$

$go\_up(t, \mathsf{down}\ ((e,x),p)) = \text{return}\ (\mathsf{comp}\ (e,x,t),p)$
$go\_up(t_1, \mathsf{left}\ ((v,x_1,x_2),t_2,p)) = \text{return}\ (\mathsf{sum}\ (v,x_1,t_1,x_2,t_2,true),p)$
$go\_up(t_2, \mathsf{right}\ ((v,x_1,x_2),t_1,p)) = \text{return}\ (\mathsf{sum}\ (v,x_1,t_1,x_2,t_2,false),p)$

$go\_right(t_1, \mathsf{left}\ ((v,x_1,x_2),t_2,p)) = \text{return}\ (t_1, \mathsf{right}\ ((v,x_1,x_2),t_2,p))$
$go\_left(t_2, \mathsf{right}\ ((v,x_1,x_2),t_1,p)) = \text{return}\ (t_2, \mathsf{left}\ ((v,x_1,x_2),t_1,p))$

$change((\_,p),t) = \text{return}\ (t,p)$

4.6.3 The cursor

We use a single reference cell *cursor* as a cursor to indicate the current position in the tree.

$initialiseCursor() = cursor := (\mathsf{hole}, \mathsf{top})$

$down() = cursor := go\_down(!cursor)$

[5]McBride's notion of differentiating datatypes is also related to Joyal's theory of species [Joy81, Joy87, BLL98].

```
$up() = cursor := go\_up(!cursor)$
$left() = cursor := go\_left(!cursor)$
$right() = cursor := go\_right(!cursor)$

$insert(t) = cursor := change(!cursor, t)$
```

The function *initialiseCursor* sets *cursor* to point to the root of an empty tree. The other functions simply specialise the zipper functions to the cursor. It is easy to see that each of the specialised zipper functions can be implemented in constant time, using the fact that !*cursor* is always overwritten.

We define a function *resetCursor* for resetting the cursor to the root of the tree. If its argument is true, then this also flips the branch flags on the way up, in order to indicate that on the next pass we want to move onto the next branch of the tree. It returns true if the whole tree has been explored, and false otherwise.

```
$resetCursor(\mathsf{top}, b)$ = return $b$
$resetCursor(\mathsf{left}(\_), true)$ =
    $right()$
    $up()$
    return $resetCursor(!cursor, false)$
$resetCursor(\mathsf{right}(\_), true)$ =
    $left()$
    $up()$
    return $resetCursor(!cursor, true)$
$resetCursor(\_, b)$ =
    $up()$
    return $resetCursor(!cursor, b)$
```

Sum bindings are flipped by moving left or right before moving up. It is only necessary to flip branches if the whole of the current subtree has been explored.

4.6.4 The *bind* and *collect* functions

The *bind* function creates a new binding if necessary, moves down the binding tree, and reflects the appropriate binder.

```
$bind^{TA}(e)$ =
    let $x = getCompBinder(!cursor, e)$
    $down()$
    return $\uparrow^{A} x$
```

```
$bind^{A_1+A_2}(e) =$
          let $(x_1, x_2, b) = getSumBinder(e)$
          if $inLeftBranch(b)$ then
                    $down()$
                    return $\uparrow^{A_1} x_1$
          else if $inRightBranch(b)$ then
                    $down()$
                    return $\uparrow^{A_2} x_2$
$inLeftBranch(b) = b$
$inRightBranch(b) = \text{not } b$
$getCompBinder((\mathsf{hole}, p), e) =$
          $insert(\mathsf{comp}\ (e, x, \mathsf{hole}))$,          $x$ fresh
          return $x$
$getCompBinder((\mathsf{comp}\ (\_, x, \_), \_), \_) =$
          return $x$
$getSumBinder((\mathsf{hole}, \_), v) =$
          let $b = true$
          $insert(\mathsf{sum}\ (v, x_1, \mathsf{hole}, x_2, \mathsf{hole}, b))$,          $x_1, x_2$ fresh
          return $(x_1, x_2, b)$
$getSumBinder((\mathsf{sum}\ (\_, x_1, \_, x_2, \_, b), \_), \_) =$
          return $(x_1, x_2, b)$
```

The auxiliary functions *getCompBinder* and *getSumBinder* are used to create a new binding if one does not already exist, then return the binder pointed to by the cursor.

The collect function resets the binding tree and generates a list of computation terms and a new binding tree. Then it outputs the term corresponding to the binding tree with the list of terms plugged in the holes.

```
$collect(f) =$
          let $cursor_0 = !cursor$
          $initialiseCursor()$
          let $es = splurge(f)$
          let $e = (!cursor)[es]$
          $cursor := cursor_0$
          return $e$
$splurge(f) =$
          let $e = f()$
          let $done = resetCursor(!cursor, true)$
          if not $done$ then return $e :: splurge(f)$
          else return $\langle e \rangle$
```

The auxiliary function *splurge* invokes its argument, then resets the cursor, and recurses until the binding tree has been completely explored.

4.6.5 Discussion

We have described an algorithm for performing normalisation by evaluation on λ^{+}_{ml*} using global state as a side-effect. Furthermore, by using the zipper structure we need only to use a single state cell. It is straightforward to translate our implementation to Filinski's setting of λ_c extended with sums, and given that we are using the standard semantics, the same technique can be used for TDPE. In fact we have implemented a TDPE algorithm in ML which uses this technique. We believe that our method can be adapted in order to replace the use of set / cupto with global state, in Balat et al.'s algorithm for performing TDPE on λ^{+}.

Addressing Sumii and Kobayashi's quotation from the beginning of this section, we claim that ours is a correct algorithm for "state-based case-insertion" (a generalisation of "state-based if-insertion"). We leave the question of efficiency to Chapter 6. It is true that our state-based implementation is effectively simulating the application of a continuation twice, and that it duplicates some static computation (which need not be duplicated if first-class continuations are available). It seems unlikely that it would be possible to use state and a native evaluator without duplicating some computation, but it may be possible to improve efficiency by performing some kind of memoisation.

Filinski uses a monadic counterpart of the algorithm of §4.5.2, which uses the continuation monad. In order to store the binding data for sums in the monad it is necessary that sums are treated as computations. It is most natural to work with an extension of λ_c rather than λ_{ml*}, as λ_c has a monadic semantics for all terms, as all terms are computations.

It is also possible to construct a monadic counterpart for the state-based algorithm. The relevant monad combines an accumulation monad over binding trees with the list (non-determinism) monad. The list contains the semantic values of the terms which are to be plugged in the tree. Thus, with regard to Filinski's quotation at the beginning of this section, we claim that it *is* possible to extend an accumulation-based interpretation to sums.

The non-deterministic accumulation monad is defined as follows:

$$\begin{aligned}
T\mathbf{A} &= \mathcal{B} \times \mathbf{A}\ \mathit{list} \\
\mathit{val}(s) &= (\mathsf{hole}, \langle s \rangle) \\
\mathit{let}((t, vs), f) &= \text{let } (ts', vs') = \mathit{unzip}(\mathit{map}\ f\ vs) \\
&\ \text{in } (t[ts'], vs')
\end{aligned}$$

We omit the details of the normalisation by evaluation algorithm, but note that it is straightforward to construct from the definition of the residualising monad. We expect that threading a monad through values as well as computations is likely to be less efficient than using an appropriate side-effect, but that it is also likely to be easier to reason about. However, with the non-deterministic accumulation monad there is no unnecessary duplication of computation because non-determinism allows all branches to be returned at once. Note that the branch flag in the binding tree datatype becomes redundant for the same reason.

Our use of state and the zipper structure can be seen as an instance of a more general technique. The same approach can be used to simulate a wide-range of uses of shift / reset, using state but no first-class control operators. The main idea is to partition the body of a shift operation into a component which can be computed before the captured continuation is invoked and a collection of other components which depend on the captured continuation. The first component is stored in global state (possibly using a zipper for convenience). The subsequent components are obtained by repeated calling a function (simulating the replication of a continuation). It would be interesting to investigate how far this approach can be taken, and how general it is. It might even be possible to use something similar to obtain a state-based alternative to Filinski's generic monadic reflection operations [Fil94, Fil96, Fil99a].

4.7 Polymorphism and recursive types

It is not difficult to add polymorphism to normalisation by evaluation algorithms. Essentially one just adds a second environment — a type environment. Then the reification and reflection functions defined on polymorphic types follow a similar pattern to those defined on function types. Vestergaard's unpublished manuscript [Ves] gives a syntactic account of normalisation by evaluation for System F. Altenkirch et al. [AHS96, AHS97] give a categorical treatment.

In principle recursive types are easy to add to normalisation by evaluation algorithms. However, the resulting algorithms may not terminate. Admitting η-expansion leads to non-terminating reduction sequences, and even β-reduction can lead to non-termination, if we are not careful. We discuss these issues further, and show how to implement normalisation by evaluation with recursive types in §5.5.

4.8 Embedding types in the semantics

Most presentations of normalisation by evaluation adopt a Curry-style typing discipline. For many purposes it does not make any difference whether the Church-style or the Curry-style is used. However, it turns out that the Church-style does allow for a useful modification of normalisation by evaluation algorithms, which is not so easily expressed in the Curry-style. Instead of indexing the reification function with a type it is sufficient to use an unindexed reification function and fold the types of bound variables into the semantics. We illustrate the technique with $\lambda^{\rightarrow}$.

Parameters which take abstractions as arguments have to be annotated with the types of bound variables:

$$\begin{aligned}
[\![\, x \,]\!]_\rho &= \rho(x) \\
[\![\, \mathsf{lam}(x^A, e) \,]\!]_\rho &= \mathit{lam}^A(\lambda s.[\![\, e \,]\!]_{\rho[x \mapsto s]}) \\
[\![\, \mathsf{app}(e_1, e_2) \,]\!]_\rho &= \mathit{app}([\![\, e_1 \,]\!]_\rho, [\![\, e_2 \,]\!]_\rho)
\end{aligned}$$

We can still define the semantic domains separately for each type, but later it will also be necessary to consider a universal domain:

$$
\begin{aligned}
[\![O]\!] &= \Lambda^{\to}\text{-}ne_O \\
[\![A \to B]\!] &= ([\![A]\!] \to [\![B]\!]) \times \tau^A \\
[\![\Lambda^{\to}]\!] &= \biguplus [\![A]\!], \quad \text{where } A \text{ ranges over all types}
\end{aligned}
$$

$$
\begin{aligned}
lam^A(f) &= (f, A) \\
app((f,A), s) &= f(s)
\end{aligned}
$$

τ^A is a singleton domain consisting only of the type A.

The reification function is defined by pattern matching. Semantic objects are either terms ranged over by e, or a (function, type) pair ranged over by (f, A).

$$
\begin{aligned}
&\downarrow^A : [\![\Lambda^{\to}]\!] \to \Lambda^{\to}\text{-}nf_A \\
&\uparrow^A : \Lambda^{\to}\text{-}ne_A \to [\![\Lambda^{\to}]\!]
\end{aligned}
$$

$$
\begin{aligned}
\downarrow(e) &= e \\
\downarrow(f,A) &= \mathsf{lam}(x^A, \downarrow(f(\uparrow^A x))), \quad x^A \text{ fresh}
\end{aligned}
$$

$$
\begin{aligned}
\uparrow^O e &= e \\
\uparrow^{A \to B} e &= (\lambda s.\, \uparrow^B (\mathsf{app}(e, \downarrow s)), A \to B)
\end{aligned}
$$

Because $\downarrow$ is not type-indexed, it is not necessary to know the type of a term in order to normalise it:

$$norm(e) = \downarrow(A)[\![e]\!]_{\uparrow}$$

Sheard [She97] used the idea of embedding types in the semantics for a variant of TDPE. In the next section we discuss how embedding types in the semantics can be useful in actual implementations.

4.9 Implementing normalisation by evaluation in SML

In this section we describe how we implement normalisation by evaluation algorithms in SML. First we define some auxiliary structures. Fresh variable names are generated using a global counter.

```
structure Supply =
struct
    val count = ref 0

    fun init () =
        count := 0

    fun new s =
        let val this = !count
        in
            count := this + 1;
            s ^ (Int.toString this)
        end
end
```

Many data structures are suitable for implementing environments, including functions, lists, binary trees and hash tables. In many applications the number of variables is likely to be limited such that it might well be reasonable just to use an array of fixed length. If memory is cheap, then this is likely to be quite efficient. Here we use SML/NJ's map data structure, which is implemented using red-black trees. We use strings for variable names (and hence for indexing the map).[6]

```
(* string as a key *)
structure StringKey : ORD_KEY =
struct
    type ord_key = string
    val compare = String.compare
end

(* map from string to 'a *)
structure StringMap = BinaryMapFn(StringKey)


structure Env =
struct
    type 'a env = 'a StringMap.map
```

[6]In our implementation of normalisation by evaluation for MIL we use the SML/NJ map data structure in conjunction with integer variable names.

```
        val extend = StringMap.insert
        val lookup = StringMap.find

        fun init () = StringMap.empty
    end
```

`extend(env, x, v)` extends the environment `env` with the binding `x` ↦ `v`.

`lookup(env, x)` returns the semantic value to which `x` is bound in `env`.

We now define simple types and syntax for Church-style λ-calculus terms in which bound variables have type annotations.

```
    structure Typing =
    struct
        datatype Type = B | F of Type * Type
    end
```

`B` represents the base type O.

`F(A,B)` represents a function of type `A` → `B`.

```
    structure Syntax =
    struct
        type ide = string
        datatype exp = VAR of ide
                     | LAM of (ide * Typing.Type) * exp
                     | APP of exp * exp
    end
```

`VAR`, `LAM`, and `APP` represent respectively variables, abstractions and applications.

The type annotations on bound variables allow us to perform normalisation by evaluation on closed terms without explicitly passing a type parameter to the normalisation function.

In order to stay within the limits of the ML type system we assume a single semantic domain for interpreting all terms. The types of the parameters are given by the following signature:

```
    signature SEMANTIC_PARMS =
    sig
```

```
        type sem
        val lam : ((sem -> sem) * Typing.Type) -> sem
        val app : (sem * sem) -> sem
    end
```

We define a functor for the parameterised semantics.

```
    functor Semantics (I : SEMANTIC_PARMS) :
    sig
        val eval : Syntax.exp * I.sem Env.env -> I.sem
    end =
    struct
        open Syntax I
        fun eval (VAR x, env) =
            Env.lookup(env, x)
          | eval (LAM ((x, t), m), env) =
            lam (fn s => eval (m, Env.extend (env, x, s)), t)
          | eval (APP (m, n), env) =
              let val f = eval (m, env)
                  val s = eval (n, env)
              in
                  app (f, s)
              end
    end
```

We find it convenient to instantiate the semantic parameters, and define the reification function at the same time using a *residualiser* [7] . This is particularly useful for normalisation by evaluation algorithms in which the semantic parameters call the reification function, such as in §4.4.

```
    signature RES =
    sig
        include SEMANTIC_PARMS
        val reify : sem -> Syntax.exp
    end
```

The `Norm` functor takes a residualiser and gives a normalisation function.

[7] This idea, and the following implementation of parameterised semantics using ML functors, is essentially due to Filinski [Fil02].

```
functor Norm(I : RES) :
sig
   val norm : Syntax.exp -> Syntax.exp
end =
struct
    structure ResidualisingSemantics = Semantics (I)

    fun norm e =
        (Supply.init ();
         I.reify (ResidualisingSemantics.eval (e, Env.init ())))
end
```

Now we define a residualiser for performing normalisation with respect to β-reduction and η-expansion.

```
structure Residualiser1 : RES =
struct
    open Syntax Typing

    datatype sem = REFLECT of exp
                 | FUN of ((sem -> sem) * Type)

    fun reify (REFLECT m) = m
      | reify (FUN (f, t)) =
        let val x = Supply.new "x"
        in
            LAM ((x, t), reify (f (reflect t (VAR x))))
        end

    and reflect B m = REFLECT m
      | reflect (F(t1, t2)) f =
        FUN (fn s => reflect t2 (APP (f, reify t1 s)))

    val lam = FUN
    fun app (FUN (f, _), s) = f s
      | app _ = raise Fail "Not␣a␣function"
end
```

Passing the residualiser to the `Norm1` structure.

```
structure Norm1 = Norm(Residualiser1)
```

gives rise to the normalisation by evaluation function `Norm1.norm`.

Here is an alternative residualiser in which the reflect function has been defunctionalised and the apply function inlined in order to make η-expansion explicit.

```
structure Residualiser2 : RES =
struct
    open Syntax Typing

    datatype sem = REFLECT of exp * Type
                 | FUN of ((sem -> sem) * Type)

    fun reify (REFLECT (m, B)) = m
      | reify (REFLECT (m, F(t1, t2)) =
        let val x = Supply.new "x"
        in
            LAM ((x, t),
                 reify (APP (m, reify (REFLECT (VAR x, t)))))
        end
      | reify (FUN (f, t)) =
        let val x = Supply.new "x"
        in
            LAM ((x, t), reify (f (REFLECT (VAR x, t))))
        end

    val lam = FUN
    fun app (FUN (f, _), s) = f s
      | app (REFLECT (m, F(t1, t2)), s) =
        REFLECT (APP (m, reify s), t2)
end

structure Norm2 = Norm(Residualiser2)
```

Now we remove the η-expansion to obtain a normalisation by evaluation algorithm for performing just β-reduction.

```
structure Residualiser3 : RES =
struct
    open Syntax Typing

    datatype sem = REFLECT of exp
```

```
                    | FUN of ((sem -> sem) * Type)

    fun reify (REFLECT m) = m
      | reify (FUN (f, t)) =
        let val x = Supply.new "x"
        in
            LAM ((x, t), reify (f (REFLECT (VAR x, t))))
        end

    val lam = FUN
    fun app (FUN (f, _), s) = f s
      | app (REFLECT (m, s) = REFLECT (APP (m, reify s))
end

structure Norm3 = Norm(Residualiser3)
```

Sums can be simulated in ML using datatypes.

```
datatype ('a,'b) Sum = Inj1 of 'a | Inj2 of 'b
```

The constructors `Inj1` and `Inj2` give the first and second injections into a binary sum. case s of $x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2$ is translated to:

```
case s of Inj1 x1 => s1
        | Inj2 x2 => s2
```

Shift and reset are implemented using call/cc and a single reference cell, as described by Filinski [Fil94].

An alternative approach to implementing normalisation by evaluation, proposed by Filinski and Yang [Yan99], uses a clever technique to encode types in such a way that the type includes the corresponding reify and reflect functions at that type. This approach allows the native ML evaluator to be used, and can be used for TDPE or decompilation of existing code. However, we would like to be able to change the semantics in ways which are not possible using this technique. Thus we use a universal semantic datatype and a custom evaluator instead.

Danvy et al. [DRR01] use *phantom types* to statically constrain the output of TDPE to be in normal form.

Chapter 5

Implementation

We have implemented a variety of normalisation by evaluation algorithms which incorporate features of MIL. Initially these began as standalone prototype implementations on 'toy languages'. Subsequently they were developed into full-blown normalisation by evaluation algorithms which operate on actual MIL generated by the SML.NET compiler from ML source code.

In this chapter we describe the incremental process we used for moving from a basic normalisation by evaluation algorithm for the computational metalanguage to normalisation by evaluation algorithms for MIL. Normalisation by evaluation for the computational metalanguage can be adapted naturally to include many of the features of MIL. We begin by adapting both the normalisation by evaluation algorithm for the computational metalanguage, and MIL, as little as possible in order to allow them to work together. The idea is that this provides a platform for assessing normalisation by evaluation and incrementally adding new features. After establishing a framework for studying normalisation by evaluation on MIL, we extend it.

The rest of this chapter is structured as follows. In §5.1 we discuss differences between the simplified and full versions of MIL and λMIL. In §5.2 we discuss the relationship between MIL and λMIL. In §5.3 we introduce our first normalisation by evaluation algorithm for λMIL. This algorithm is not semantics-preserving. In §5.4 we give an improved algorithm which is semantics-preserving. In §5.5 we add sums and recursive types. In §5.6 we add exceptions. In §5.7 we give a normalisation by evaluation algorithm which targets MIL rather than λMIL. Finally, in §5.8 we summarise our range of normalisation by evaluation implementations.

5.1 Full versus simplified MIL/λMIL

In the first four chapters of this thesis we have worked exclusively with simplified versions of MIL and λMIL. Mostly, the implementation details are not important, so we shall continue to use the simplified calculi. However, in this chapter, and the next, we shall indicate where the concrete implementation requires special attention.

5.1.1 Arity-raising

In concrete MIL and λMIL, terms are *arity-raised*: functions, computations, sums and products are *n*-ary rather than binary or unary. In contrast, simplified MIL and λMIL have unary functions and computations, whilst products and sums are binary, and there is also a unary product **1**. Arity-raising is not difficult to handle, though it does introduce a few subtleties, and the semantics has to be augmented with arity information in certain places.

5.1.2 Source information

For debugging purposes, all bound variables in the implementation are annotated with a string. This string is the name of the corresponding source variable, if any, and the empty string otherwise. We embed the source information in the semantics using the same technique that we used to embed the type annotations on bound variables in §4.8. Each parameter that takes an abstraction as an argument is augmented with an extra parameter for the source information.

5.2 MIL versus λMIL

MIL is rather unlike the other λ-calculi to which we have applied normalisation by evaluation. There is a syntactic distinction between *atomic* and *non-atomic* values. The syntax ensures that all non-atomic values are named. This is a desirable property for a compiler intermediate language to have, as it exposes the control flow and enables all kinds of program transformations to be naturally expressed [App92].

Recall from Chapter 2 that MIL is just a restriction of λMIL, and the convertibility relation of MIL is correspondingly defined as the restriction of the convertibility relation of λMIL to MIL. Now, λMIL is just an extension of λ_{ml*} and, as is shown in the rest of this chapter, one can define normalisation by evaluation for λMIL as an extension of normalisation by evaluation for λ_{ml*}.

A natural question to ask is whether one can obtain a normalisation by evaluation algorithm for MIL simply by restricting the normalisation by evaluation algorithm for λMIL. Unfortunately, this does not work, as the normal forms often lie outside of MIL. For instance, $\mathsf{letval}\ x \Leftarrow \mathsf{pair}(a,b)\ \mathsf{in}\ \mathsf{app}(f,x)$ normalises to $\mathsf{app}(f,\mathsf{pair}(a,b))$ in λMIL, but this term is not a valid MIL term as $\mathsf{pair}(a,b)$ is not an atom.

It is not entirely clear what canonical normal forms should be for MIL. For instance:

- The order in which independent values are bound is not important:

$$\begin{aligned}&\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ \mathsf{letval}\ z \Leftarrow \mathsf{proj}_1(y)\ \mathsf{in}\ f z\\ &\qquad\equiv \mathsf{letval}\ z \Leftarrow \mathsf{proj}_1(y)\ \mathsf{in}\ \mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ f z\end{aligned} \tag{5.1}$$

- The same non-atomic value may be bound more than once:

$$\begin{aligned}&\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ \mathsf{letfun}\ g(x) \Leftarrow m\ \mathsf{in}\ \mathsf{val}(\mathsf{pair}(f,g))\\ &\qquad\equiv \mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ \mathsf{val}(\mathsf{pair}(f,f))\end{aligned} \tag{5.2}$$

Recall that the set of MIL terms **MIL** is a strict subset of the set of λMIL terms ΛMIL. One way of defining normal forms is to define a function *milify* from ΛMIL to **MIL** such that *milify*$(e) \equiv e$. If such a function exists then it can be composed with a normalisation function for λMIL to give a normalisation function for MIL. In fact no such function can exist, as there are λMIL terms which are not convertible to any MIL terms. For instance, the value term $\mathsf{pair}(\mathsf{pair}(a,b),c)$ cannot be converted to a value term in MIL, as the only way of naming $\mathsf{pair}(a,b)$ is inside a computation term. However, the MIL term output by the frontend is always a computation term, and it is possible to define versions of *milify* restricted to computation terms. We shall discuss how to do this in section 5.7. For now we concentrate on normalisation by evaluation for λMIL. Note that we can still use the frontend to generate λMIL terms, and obtain useful performance data, without having to consider normalisation by evaluation for MIL.

Parameterised semantics As usual we give a parameterised semantics for ΛMIL (Figure 5.1). Type annotations are attached to the parameters which interpret terms with bound variables.

Unknown terms and types λMIL has many new constructs on top of λ_{ml*} for features such as sums, recursive types, exceptions and references. There are a number of possible ways of dealing with these. We take an incremental approach, in which we begin by eliding the constructs that we do not wish to handle (essentially pretending they have no semantic content at all), then subsequently deal with them one-by-one. We call the terms we wish to elide *unknown terms* and the types we wish to elide *unknown types*.

5.3 Absorbing values for unknowns

As a first attempt, we introduce a special *absorbing* value-term constructor $\mathsf{unknown}$ to which unknown values will be normalised. Unknown computation terms will be normalised to $\mathsf{val}(\mathsf{unknown})$. Unknown values could potentially be of any type, so we add the typing rule:

$$\frac{}{\mathsf{unknown}{:}A}$$

Values of unknown type will be normalised to $\mathsf{unknown}$ and computations of unknown type to $\mathsf{val}(\mathsf{unknown})$. The idea is that new term constructors be incrementally added to the algorithm. The value term $\mathsf{unknown}$ is *absorbing* in that placing it in an elimination

$$
\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![*]\!]_\rho &= \mathit{star} \\
[\![c^A]\!]_\rho &= \mathit{constant}^A(c) \\
[\![\mathsf{lam}(x^A, m)]\!]_\rho &= \mathit{lam}^A(\lambda s.[\![m]\!]_{\rho[x\mapsto s]}) \\
[\![\mathsf{pair}(v, w)]\!]_\rho &= \mathit{pair}([\![v]\!]_\rho, [\![w]\!]_\rho) \\
[\![\mathsf{proj}_i(v)]\!]_\rho &= \mathit{proj}_i([\![v]\!]_\rho) \\
[\![\mathsf{inj}_i(v)]\!]_\rho &= \mathit{inj}_i([\![v]\!]_\rho) \\
[\![\mathsf{fold}_{\mu X.A}(v)]\!]_\rho &= \mathit{fold}_{\mu X.A}([\![v]\!]_\rho) \\
[\![\mathsf{unfold}(v)]\!]_\rho &= \mathit{unfold}([\![v]\!]_\rho) \\
\\
[\![\mathsf{app}(f, v)]\!]_\rho &= \mathit{app}([\![f]\!]_\rho, [\![v]\!]_\rho) \\
[\![\mathsf{val}(v)]\!]_\rho &= \mathit{val}([\![v]\!]_\rho) \\
[\![\mathsf{let}\ x^A \Leftarrow m\ \mathsf{in}\ n]\!]_\rho &= \mathit{let}^A([\![m]\!]_\rho, \lambda v.[\![n]\!]_{\rho[x\mapsto v]}) \\
[\![\mathsf{case}\ v\ \mathsf{of}\ (x_1^{A_1} \Rightarrow n_1 \mid x_2^{A_2} \Rightarrow n_2)]\!]_\rho &= \mathit{case}^{(A_1,A_2)}\begin{pmatrix} [\![v]\!]_\rho, \\ \lambda s.[\![n_1]\!]_{\rho[x_1\mapsto s]}, \\ \lambda s.[\![n_2]\!]_{\rho[x_2\mapsto s]} \end{pmatrix} \\
[\![\mathsf{raise}(E)]\!]_\rho &= \mathit{raise}(E) \\
[\![\mathsf{try}\ x^A \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H]\!]_\rho &= \mathit{try}^A([\![m]\!]_\rho, \lambda s.[\![n]\!]_{\rho[x\mapsto s]}, \mathcal{H}[\![H]\!]_\rho) \\
[\![\mathsf{read}(v)]\!]_\rho &= \mathit{read}([\![v]\!]_\rho) \\
[\![\mathsf{write}(v, w)]\!]_\rho &= \mathit{write}([\![v]\!]_\rho, [\![w]\!]_\rho) \\
[\![\mathsf{new}]\!]_\rho &= \mathit{new} \\
\\
\mathcal{H}[\![H]\!]_\rho &= \mathit{map}\ (\lambda(E, n).(E, \lambda().[\![n]\!]_\rho))\ H
\end{aligned}
$$

Figure 5.1: Parameterised semantics for ΛMIL

context gives another unknown. For instance, for sums and products, this property is captured by the absorbing conversion rules:

$$\begin{array}{ll}
(\to.\mathrm{ABS}) & \mathsf{app}(\mathsf{unknown}, v) \equiv \mathsf{unknown} \\
(\times.\mathrm{ABS}(i)) & \mathsf{proj}_i(\mathsf{unknown}) \equiv \mathsf{unknown}
\end{array}$$

This approach has the advantage of being easy to implement, and the resulting term containing only the parts of the source term which are relevant to the normalisation algorithm. Assuming a significant proportion of the source term is not unknown, it should provide us with an indication of the behaviour of normalisation by evaluation algorithms on realistically sized terms. It has the obvious disadvantage that the resulting code has large chunks missing, so it cannot be used to compile code. However, this is not so important for the purposes of benchmarking normalisation algorithms.

5.3.1 Normalisation

We consider extensional normalisation for λMIL restricted to the conversion rules for unit, functions, products and computations, where sums, recursive types, exceptions and references, and their associate terms are unknown. Thus, the conversion rules are: $\mathbf{1}.\eta$, $\to.\beta$, $\to.\eta$, $\times.\beta i$, $\times.\eta$, $T.\beta$, $T.\eta$, $T.T.$CC, $\to.$ABS and $\times.$ABS(i). We obtain long normal forms, corresponding to η-expansion. They are given by the grammar:

$$\begin{array}{ll}
\text{Normal values} & v, w ::= u^0 \mid * \mid \mathsf{lam}(x^A, m) \mid \mathsf{pair}(v, w) \mid \mathsf{unknown} \\
\text{Neutral values} & u^A ::= x^A \mid c^A \mid \mathsf{proj}_1(u^{A\times B}) \mid \mathsf{proj}_2(u^{B\times A}) \\
\text{Normal computations} & m ::= \mathsf{val}(v) \mid \mathsf{let}\ x^A \Leftarrow \mathsf{app}(u^B, v)\ \mathsf{in}\ m
\end{array}$$

v, w ranges over normal values, n^A over neutral values of type A, and m over normal computations. The set of normal forms ΛMIL-*nf* is the union of the set of normal values and the set of normal computations.

We shall now present a normalisation by evaluation algorithm. A residualising semantics using the continuation monad appears in Figure 5.2. The semantic domain for each value type is augmented with *unknown*, the interpretation of unknown values. Unknown types are simply interpreted as *unknown*. Note that the parameter *Comp* is

$$
\begin{aligned}
[\![\mathit{Int}]\!] &= \Lambda\text{MIL-}\mathit{nf}_{\mathit{Int}} + \mathit{unknown} \\
[\![A\ \mathit{ref}]\!] &= \mathit{unknown} \\
[\![\mathbf{1}]\!] &= \mathit{star} + \mathit{unknown} \\
[\![A \to T_{\varepsilon}(B)]\!] &= ([\![A]\!] \to [\![T_{\varepsilon}(B)]\!]) + \mathit{unknown} \\
[\![A \times B]\!] &= ([\![A]\!] \times [\![B]\!]) + \mathit{unknown} \\
[\![A + B]\!] &= \mathit{unknown} \\
[\![\mu X.A]\!] &= \mathit{unknown} \\
\\
[\![T_{\varepsilon}(A)]\!] &= \mathit{Comp}_{\varepsilon}([\![A]\!])
\end{aligned}
$$

Figure 5.2: Semantics with absorbing values for λMIL

annotated with a set of effects. In our implementations we just pass the effects through unchanged, but potentially one could do something more sophisticated with them. The semantic parameters appear in Figure 5.3. Parameters which eliminate values, namely *app* and *proj*, return *unknown* if the elimination argument is *unknown*. Unknown values are interpreted as *unknown* and unknown computations as *val*(*unknown*).

The reification and reflection functions appear in Figure 5.4. They are standard, except:

- *unknown*, at any type, is reified as unknown; and
- any value, at an unknown type, is reflected as *unknown*.

5.3.2 Implementation issues

A number of implementation issues arise:

- Concrete MIL includes some support for polymorphism. We conveniently ignore polymorphism (without just normalising every polymorphic function to unknown), by applying SML.NET's monomorphisation transformation as a pre-processing stage.

$$
\begin{aligned}
&\textit{star} \text{ is uninterpreted} \\
\mathit{constant}^A(c) &= \uparrow^A c \\
\mathit{lam}(f) &= f \\
\mathit{app}(f,s) &= f\ s \\
\mathit{app}(\mathit{unknown},s) &= \mathit{unknown} \\
\mathit{pair}(s_1,s_2) &= (s_1,s_2) \\
\mathit{proj}_i(s_1,s_2) &= s_i \\
\mathit{proj}_i(\mathit{unknown}) &= \mathit{unknown} \\
\mathit{val}(s) &= \lambda\kappa.\kappa\ s \\
\mathit{let}(t,f) &= \lambda\kappa.t(\lambda s.f\ s\ \kappa)
\end{aligned}
\qquad
\begin{aligned}
\mathit{fold}_{\mu X.A}(s) &= \mathit{unknown} \\
\mathit{unfold}(s) &= \mathit{unknown} \\
\mathit{inj}_i(s) &= \mathit{unknown} \\
\mathit{case}(p,f_1,f_2) &= \mathit{val}(\mathit{unknown}) \\
\mathit{raise}(E) &= \mathit{val}(\mathit{unknown}) \\
\mathit{try}(s,f_1,H) &= \mathit{val}(\mathit{unknown}) \\
\mathit{read}(s) &= \mathit{val}(\mathit{unknown}) \\
\mathit{write}(s,s') &= \mathit{val}(\mathit{unknown}) \\
\mathit{new} &= \mathit{val}(\mathit{unknown})
\end{aligned}
$$

$$\mathit{Comp}_\varepsilon([\![A]\!]) = ([\![A]\!] \to \Lambda\text{MIL-}\mathit{nf}_{T_\varepsilon(A)}) \to \Lambda\text{MIL-}\mathit{nf}_{T_\varepsilon(A)}$$

Figure 5.3: Semantic parameters with absorbing values for λMIL

- We do not take advantage of MIL's effect annotations, but we do preserve them.
- The implementation includes support for .NET interoperability via various additions to MIL. We just treat these as unknown terms in the normal way.

5.4 Fixed constants for unknowns

At first sight, one might ask why it is not just as straightforward simply to leave unknown terms as is, rather than normalising them to unknown (or val(unknown)). One problem is how to define the semantics. The hope is to preserve the term, so it would seem reasonable simply to interpret unknown terms as themselves.

It then becomes necessary to extend the definitions of the semantic parameters. Our discussion focuses on *app*, but the same principles apply to other elimination parameters as well. We need to define $\mathit{app}(u,s)$ for u the interpretation of an unknown value. In the absorbing semantics, if u were *unknown* then we would just return *unknown*. Now, u is a term, so analogously we would like to return another term. This should be

$$\downarrow^A : [\![A]\!] \to \Lambda\mathrm{MIL}\text{-}nf_A$$
$$\uparrow^A : \Lambda\mathrm{MIL}\text{-}ne_A \to [\![A]\!]$$

$$\begin{aligned}
\downarrow^A \mathit{unknown} &= \mathsf{unknown}\\
\downarrow^{Int} e &= e\\
\downarrow^{\mathbf{1}} \mathit{star} &= *\\
\downarrow^{A\to B} f &= \mathsf{lam}(x^A, \downarrow^B (f(\uparrow^A x))), \quad x^A \text{ fresh}\\
\downarrow^{A\times B} (s_1, s_2) &= \mathsf{pair}(\downarrow^A s_1, \downarrow^B s_2)\\
\downarrow^{T_\varepsilon(A)} t &= t(\lambda s.\mathsf{val}(\downarrow^A s))
\end{aligned}$$

$$\begin{aligned}
\uparrow^{Int} e &= e\\
\uparrow^{\mathbf{1}} e &= \mathit{star}\\
\uparrow^{A\to B} e &= \lambda s. \uparrow^B (\mathsf{app}(e, (\downarrow^A s)))\\
\uparrow^{A\times B} e &= (\uparrow^A \mathsf{proj}_1(e), \uparrow^B \mathsf{proj}_2(e))\\
\uparrow^{T_\varepsilon(A)} e &= \lambda\kappa.\mathsf{let}\ x^A \Leftarrow e\ \mathsf{in}\ \kappa(\uparrow^A x), \quad x^A \text{ fresh}\\
\uparrow^A e &= \mathit{unknown}
\end{aligned}$$

$$\mathit{norm}(e^A) = \downarrow^A [\![e]\!]_\uparrow$$

Figure 5.4: Extensional NBE with absorbing values

u applied to a term which s denotes. The obvious way to obtain a syntactic representation of s is to reify it. But, we have defined $\downarrow$ as a type-indexed function, so we need to know the type of s. One possibility is to embed a type-inference algorithm in the semantics. A simpler solution is to use the technique explained in Chapter 4 to embed the types of bound variables in the semantics. Then $\downarrow$ can be defined without a type parameter.

Note, however, that the resulting semantics is not sound with respect to η-rules. It is sound with respect to the β and CC rules, though. In fact, it is just the usual semantics for intensional normalisation by evaluation in the computational metalanguage, where unknown terms are interpreted as fixed constants. One can obtain a pseudo normalisation by evaluation algorithm from this semantics which performs some but not all η-expansion. Specifically, known terms are expanded and unknown ones are not. However, this ad hoc η-expansion seems somewhat unnatural. A more natural approach is just to take the usual semantics for extensional normalisation by evaluation where unknown terms are interpreted as fixed constants. But then type inference becomes necessary again, as in extensional normalisation by evaluation a fixed constant c^A is interpreted as:

$$\uparrow^A c$$

Of course, for intensional normalisation by evaluation, this issue does not arise because reflection is just the identity. Given that we would be unlikely to want to perform η-expansion in a compiler anyway, we shall avoid type-inference in the semantics, and implement intensional normalisation by evaluation instead.

As a further improvement over absorbing normalisation we shall normalise known subterms which appear inside unknown terms. This is expressed by treating unknown *syntax constructors*, rather than unknown terms, as fixed constants.

5.4.1 Normalisation

We consider intensional normalisation for λMIL restricted to the conversion rules for unit, functions, products and computations, where sums, recursive types, exceptions and references, and their associate terms are unknown. Thus, the conversion rules are: $\rightarrow.\beta$, $\times.\beta i$, $T.\beta$, $T.T.$CC. Normal forms are given by the grammar:

$$
\begin{aligned}
[\![\mathit{Int}]\!] &= \Lambda\mathrm{MIL}\text{-}\mathit{ne}_{\mathit{Int}} \\
[\![A\ \mathit{ref}]\!] &= \Lambda\mathrm{MIL}\text{-}\mathit{ne}_{A\ \mathit{ref}} \\
[\![\mathbf{1}]\!] &= \mathit{star} \\
[\![A \to T_{\varepsilon}(B)]\!] &= (([\![A]\!] \to [\![T_{\varepsilon}(B)]\!]) \times \tau_A) + \Lambda\mathrm{MIL}\text{-}\mathit{ne}_{A\to T_{\varepsilon}(B)} \\
[\![A \times B]\!] &= ([\![A]\!] \times [\![B]\!]) + \Lambda\mathrm{MIL}\text{-}\mathit{ne}_{A\times B} \\
[\![A + B]\!] &= \Lambda\mathrm{MIL}\text{-}\mathit{ne}_{A+B} \\
\\
[\![T_{\varepsilon}(A)]\!] &= \mathit{Comp}_{\varepsilon}([\![A]\!])
\end{aligned}
$$

Figure 5.5: Semantics with fixed constants for λMIL

Normal values	$v,w ::= u \mid \mathsf{lam}(x,m) \mid \mathsf{pair}(v,w)$
Neutral values	$u ::= x \mid * \mid c \mid \mathsf{proj}_i(u) \mid \mathsf{inj}_i(v) \mid \mathsf{fold}_{\mu X.A}(u) \mid \mathsf{unfold}(u)$
Normal computations	$m,n ::= \mathsf{val}(v) \mid p \mid \mathsf{let}\ x \Leftarrow p\ \mathsf{in}\ m$
Neutral computations	$p ::= \mathsf{app}(u,v)$
	$\mid \mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$
	$\mid \mathsf{raise}(E) \mid \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H$
	$\mid \mathsf{read}(v) \mid \mathsf{write}(v,w) \mid \mathsf{new}$

v,w ranges over normal values, u over neutral values, m,n over normal computations and p over neutral computations. The set of normal forms ΛMIL-*nf* is the union of the set of normal values and the set of normal computations. The set of neutral terms ΛMIL-*ne* is the union of the set of neutral values and the set of neutral computations.

We now present our intensional normalisation by evaluation algorithm with unknown syntax constructors interpreted as fixed constants.

The parameterised semantics of Figure 5.1 still applies, but this time the extra type annotations on the parameters are needed. A residualising semantics, using the metalanguage's internal monad and making use of the shift and reset control operators, appears in Figure 5.5. The semantics is the standard one for intensional normalisation by evaluation, except unknown types are interpreted as the set of neutral terms of the

$$
\begin{aligned}
&\mathit{star} \text{ is uninterpreted}\\
\mathit{constant}^A(c) &= c^A\\
\mathit{lam}^A(f) &= (f, A)\\
\mathit{app}((f,A),s) &= f\ s\\
\mathit{app}(e, s^A) &= \mathsf{app}(e, \downarrow s)\\
\mathit{pair}(s_1,s_2) &= (s_1,s_2)\\
\mathit{proj}_i(s_1,s_2) &= s_i\\
\mathit{proj}_i(e) &= \mathsf{proj}_i(e)\\
\mathit{inj}_i(s) &= \mathsf{inj}_i(\downarrow s)\\
\mathit{fold}_{\mu X.A}(s) &= \mathsf{fold}_{\mu X.A}(\downarrow s)\\
\mathit{unfold}(s) &= \mathsf{unfold}(\downarrow s)\\
\mathit{case}(p, f_1, f_2) &= \mathsf{case} \downarrow p \text{ of } (x_1 \Rightarrow \downarrow f_1(x_1) \mid x_2 \Rightarrow \downarrow f_2(x_2))\\
\mathit{val}(v) &\text{ is uninterpreted}\\
\mathit{let}^A(\mathit{val}(v), f) &= f\ v\\
\mathit{let}^A(e, f) &= \mathit{bindCmp}^{T_\emptyset(A)}(e, f)\\
\mathit{raise}(E) &= \mathsf{raise}(E)\\
\mathit{try}^A(s, f, H) &= \mathsf{try} \downarrow s \Leftarrow x^A \text{ in } \downarrow f(x) \text{ unless } \downarrow H\\
\mathit{read}(s) &= \mathsf{read}(\downarrow s)\\
\mathit{write}(s, s') &= \mathsf{write}(\downarrow s, \downarrow s')\\
\mathit{new} &= \mathsf{new}
\end{aligned}
$$

$$
\mathit{Comp}_\varepsilon([\![A]\!]) = \mathit{val}([\![A]\!]) + \Lambda\text{MIL-}\mathit{ne}_{T_\varepsilon(A)}
$$

$$
\begin{aligned}
\mathit{collect}(t) &= \langle t \rangle\\
\mathit{bindCmp}^{T_\emptyset(A)}(e, f) &= \mathcal{S}(\lambda\kappa.\mathsf{let}\ x^A \Leftarrow e \text{ in } \mathit{collect}(\lambda().\kappa(f\ x))), \quad x^A \text{ fresh}
\end{aligned}
$$

Figure 5.6: Semantic parameters with fixed constants for λMIL

$$\downarrow : [\![\Lambda\text{MIL}]\!] \to \Lambda\text{MIL-}\mathit{nf}$$

$$\begin{aligned}
\downarrow e &= e \\
\downarrow (f, A) &= \mathsf{lam}(x^A, \mathit{collect}(\lambda().\downarrow (f\ x))), \quad x^A \text{ fresh} \\
\downarrow \mathit{star} &= * \\
\downarrow (s_1, s_2) &= \mathsf{pair}(\downarrow^A s_1, \downarrow^B s_2) \\
\downarrow \mathit{val}(s) &= \mathsf{val}(\downarrow s)
\end{aligned}$$

$$\mathit{norm}(e) = \downarrow [\![e]\!]_{\uparrow}$$

where e ranges over terms
s ranges over all semantic objects
f ranges over functions

Figure 5.7: Intensional NBE with fixed constants

corresponding type. The semantic parameters appear in Figure 5.6. Each unknown parameter is instantiated by reifying all of its arguments.

Reification is defined in Figure 5.7. It is the usual non-type-indexed flavour of §4.8.

5.4.2 Implementation issues

This time we treat both polymorphic terms and .NET interoperability terms as fixed constants. Because we are using fixed constants merely for syntax constructors, further reductions are not blocked. We could have used SML.NET's monomorphisation phase as before, but the monomorphisation phase has the side-effect of performing further reductions. We prefer to perform those reductions through our normalisation by evaluation algorithm.

For future work we suggest adding polymorphism to the normalisation by evaluation algorithm. As indicated in §4.7 this should be reasonably straightforward to do by adding a type environment to the semantics.

5.5 Sums and recursive types

We have already seen one algorithm implementing normalisation by evaluation for λ^{+}_{ml*} using delimited continuations in §4.5.2. We now adapt this technique to the setting in which we have fixed constants for unknowns, and we do not perform η-expansion. We also add recursive types.

Recursive types are problematic for two reasons:

- First, if we admit negative recursive types then we can embed the untyped λ-calculus, giving rise to non-terminating terms (under β-reduction). For example,

  $$\begin{aligned}\mathsf{app}(\mathsf{lam}(x^A, \mathsf{app}(\mathsf{unfold}(x), x)), \mathsf{fold}_A(\mathsf{lam}(x^A, \mathsf{app}(\mathsf{unfold}(x), x))))\\ \text{where } A = \mu X.(X \to X)\end{aligned}$$

  β-reduces to itself.

- Second, it is evident that η-expansion does not terminate — even for positive recursive types. For instance, a variable of list type can be η-expanded as follows:

  $$\begin{aligned}&x{:}\mu X.(1 + (Int * X)) =\\ &\quad \mathsf{fold}_{\mu X.(1+(Int*X))}(\mathsf{case}\ \mathsf{unfold}(x)\ \mathsf{of}\ (x_1 \Rightarrow () \mid x_2 \Rightarrow \mathsf{pair}(\mathsf{proj}_1(x_2), \mathsf{proj}_2(x_2))))\end{aligned}$$

  But, $\mathsf{proj}_2(x_2)$ is another neutral term of type $\mu X.(1 + (Int * X))$, so this expansion can be applied ad infinitum.

We have already decided not to perform η-expansion, so that solves the second problem. One approach to the first problem, is simply to ignore it! Many programs do not use negative recursive types in a way which could lead to non-termination, so this is sufficient for obtaining benchmarks. We have validated this assertion empirically. However, this approach would not be very satisfactory for an actual compiler. A compiler should terminate on all inputs. Our solution is to add a side-condition to the $\mu.\beta$-rule such that it can be applied only when the recursive type has no negative occurrences of the bound variable:

$$(\mu.\beta') \quad \mathsf{unfold}(\mathsf{fold}_{\mu X.A}(v)) \equiv v, \qquad \text{if not } \mathit{isNegative}(X, A)$$

5.5.1 Normalisation

Extending the normalisation problem of 5.4.1 to include sums and recursive types, we add the rules: $+.\beta i$, $+.T.\mathrm{CC}$ and $\mu.\beta'$. Normal forms are now given by the grammar:

Normal values	$v, w ::= u \mid \mathsf{lam}(x, m) \mid \mathsf{pair}(v, w)$
	$\mid \mathsf{inj}_i(v) \mid \mathsf{fold}_{X.A}(v)$
Neutral values	$u ::= x \mid * \mid c \mid \mathsf{proj}_i(u) \mid \mathsf{unfold}(u)$
Normal computations	$m, n ::= \mathsf{val}(v) \mid p \mid \mathsf{let}\ x \Leftarrow p\ \mathsf{in}\ m$
	$\mid \mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$
Neutral computations	$p ::= \mathsf{app}(u, v)$
	$\mid \mathsf{raise}(E) \mid \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H$
	$\mid \mathsf{read}(v) \mid \mathsf{write}(v, w) \mid \mathsf{new}$

The interpretation of sums and recursive types is:

$$\begin{aligned}
[\![A_1 + A_2]\!] &= (inj_1([\![A_1]\!]) + inj_2([\![A_2]\!])) + \Lambda\mathrm{MIL}\text{-}ne_{A_1+A_2} \\
[\![\mu X.A]\!] &= fold_{\mu X.A}([\![A[X := \mu X.A]]\!]) + \Lambda\mathrm{MIL}\text{-}ne_{\mu X.A}
\end{aligned}$$

The parameters for sums and recursive types are:

$$\begin{aligned}
&inj \text{ is uninterpreted} \\
case^{\cdot}(inj_i(v), f_1, f_2) &= f_i\ v \\
case^{A_1+A_2}(e, f_1, f_2) &= bindSum^{A_1+A_2}(e, f_1, f_2) \\
&fold \text{ is uninterpreted} \\
unfold(fold_{\mu X.A}(s)) &= s, \qquad \text{if } X \text{ does not occur negatively in } A \\
unfold(fold_{\mu X.A}(s)) &= \mathsf{unfold}(\mathsf{fold}_{X.A}(\downarrow s)), \qquad \text{if } X \text{ does occur negatively in } A \\
unfold(e) &= \mathsf{unfold}(e)
\end{aligned}$$

$$bindSum^{A_1+A_2}(e, f_1, f_2) = \mathcal{S}\left(\lambda\kappa.\mathsf{case}\ e\ \begin{array}{ll} \mathsf{of} & x_1^{A_1} \Rightarrow collect(\lambda().\kappa(f_1(x_1))) \\ \mid & x_2^{A_2} \Rightarrow collect(\lambda().\kappa(f_2(x_2))) \end{array}\right),$$
$$x_1, x_2 \text{ fresh}$$

The reification function is extended as follows:

$$\downarrow \mathit{inj}_i(s) = \mathsf{inj}_i(\downarrow s)$$
$$\downarrow \mathit{fold}_{\mu X.A}(s) = \mathsf{fold}_{X.A}(\downarrow s)$$

5.5.2 Implementation issues

In concrete MIL the case statement is arity raised and also has an optional default case. The algorithm we have described is easily generalised to this setting. ML itself has a built-in boolean type, which SML.NET translates to a sum type in MIL. ML does not allow user-defined sum types, although they can be simulated using datatypes. Recursive sum types arising from ML datatypes (such as lists) are very common in MIL code. Hence, adding recursive types to our normalisation by evaluation algorithm enables significantly more conversions than just adding sum types.

There is a practical problem with performing $+.T$.CC-conversion: each time it is applied, a term is duplicated. This can easily lead to an exponential blow-up in the size of terms, and is known to be a problem in practice [BD02]. There are a number of ways of alleviating the problem. We return to this issue in Chapter 6, once we have some concrete data.

5.6 Exceptions

Exceptions do not really add anything new. Essentially they can be seen as a combination of computations and sums (the exception monad is given by a sum). Extending the normalisation problem of 5.5.1 to exceptions, we add the $T_{exn}.\beta$ rule, and generalise $T.T$.CC and $+.T$.CC to the versions of Figure 2.10. Normal forms are now given by the grammar:

$$\begin{array}{lrl}
\text{Normal values} & v,w ::= & u \mid \mathsf{lam}(x,m) \mid \mathsf{pair}(v,w) \\
& & \mid \mathsf{inj}_i(v) \mid \mathsf{fold}_{X.A}(v) \\
\text{Neutral values} & u ::= & x \mid * \mid c \mid \mathsf{proj}_i(u) \mid \mathsf{unfold}(u) \\
\text{Normal computations} & m,n ::= & \mathsf{val}(v) \mid \mathsf{raise}(E) \mid p \\
& & \mid \mathsf{try}\ x \Leftarrow p\ \mathsf{in}\ m\ \mathsf{unless}\ H \mid \mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2) \\
\text{Neutral computations} & p ::= & \mathsf{app}(u,v) \\
& & \mid \mathsf{read}(v) \mid \mathsf{write}(v,w) \mid \mathsf{new}
\end{array}$$

where $H(E)$ is normal for all exceptions E.

The interpretation of computations is extended to account for exceptions:

$$Comp_{\varepsilon}(A) = raise(\varepsilon \cap \mathbb{E}) + val([\![A]\!]) + \Lambda\text{MIL-}ne_{T_{\varepsilon}(A)}$$

Analogously to the case of sums, the shift operator is used to follow each branch of a try. First, the default branch (for the case in which no exception is raised) is taken, then each of the branches of the handler are taken.

$$\begin{aligned}
& raise \text{ is uninterpreted} \\
try^A(val(v), f, H) &= f\ v \\
try^A(raise(E), f, H) &= H(E) \\
try^A(e, f, H) &= bindExn^A(e, f, H)
\end{aligned}$$

$$\begin{aligned}
bindExn^A(e, f, H) = {} & \mathcal{S}(\lambda\kappa.\mathsf{try}\ x^A \Leftarrow e\ \mathsf{in}\ collect(\lambda().\kappa(f\ x))\ \mathsf{unless}\ H'), \\
& \text{where} \\
& \quad x^A \text{ is fresh} \\
& \quad H' = map\ (\lambda(E,t).(E, collect(\lambda().\kappa(t())))) \ H
\end{aligned}$$

Reifying the semantic representation of an exception just gives the syntactic representation of the exception:

$$\downarrow raise(E) = \mathsf{raise}(E)$$

5.7 Targeting MIL

One way of translating λMIL terms into MIL is to use a call-by-value embedding. The call-by-value embedding of λMIL into itself simply names every non-atomic value. It appears in Figure 5.8. The embedding is very similar to Moggi's call-by-value embedding of simply-typed λ-calculus into the computational metalanguage. Various authors [Dan92, HD94, SW97] have noted the similarity between the call-by-value embedding and call-by-value CPS transformations. In particular, the call-by-value embedding introduces *administrative redexes*. In other words, it introduces more names than are necessary. For instance, some atoms are renamed.

Just like for CPS transformations, one can perform the embedding and reduce the administrative redexes in one pass — this gives a one-pass monadic transformation. We shall use the same machinery to combine normalisation by evaluation with the call-by-value embedding and administrative reductions.

5.7.1 Straight to MIL

Using delimited continuations a one-pass transformation can effectively be folded into the normalisation by evaluation algorithm. We consider intensional normalisation for MIL restricted to the conversion rules for unit, functions, products and computations, where sums, recursive types, exceptions and references, and their associate terms are unknown. In other words this is the MIL version of the normalisation problem discussed in 5.4.1. The conversion rules are: $\to.\beta$, $\times.\beta i$, $T.\beta$, $T.T.$CC. Normal forms are given by the grammar:

Normal values	$v, w ::= u \mid \mathsf{lam}(x, m) \mid \mathsf{pair}(a, b)$
Atoms	$a, b ::= x \mid * \mid c$
Neutral values	$u ::= a \mid \mathsf{proj}_i(a) \mid \mathsf{inj}_i(a) \mid \mathsf{fold}_{X.A}(a) \mid \mathsf{unfold}(a)$
Normal computations	$m, n ::= \mathsf{val}(v) \mid p \mid \mathsf{let}\ x \Leftarrow p\ \mathsf{in}\ m$
Neutral computations	$p ::= \mathsf{app}(a, b)$
	$\mid \mathsf{case}\ m\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$
	$\mid \mathsf{raise}(E) \mid \mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H$
	$\mid \mathsf{read}(a) \mid \mathsf{write}(a, b) \mid \mathsf{new}$

Values

$$\mathcal{E}'_v : \Lambda\mathrm{MIL}_A \to \mathbf{MIL}_{T_\emptyset(A)}$$

$$\begin{aligned}
\mathcal{E}'_v(x) &= \mathsf{val}(x)\\
\mathcal{E}'_v(*) &= \mathsf{val}(*)\\
\mathcal{E}'_v(c) &= \mathsf{val}(c)\\
\mathcal{E}'_v(\mathsf{lam}(x,m)) &= \mathsf{val}(\mathsf{lam}(x,\mathcal{E}_v(m)))\\
\mathcal{E}'_v(\mathsf{pair}(v,w)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow \mathcal{E}'_v(w)\ \mathsf{in}\ \mathsf{val}(\mathsf{pair}(v,w))\\
\mathcal{E}'_v(\mathsf{proj}_i(v)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{val}(\mathsf{proj}_i(x))\\
\mathcal{E}'_v(\mathsf{inj}_i(v)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{val}(\mathsf{inj}_i(x))\\
\mathcal{E}'_v(\mathsf{fold}_{X.A}(v)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{val}(\mathsf{fold}_A(x))\\
\mathcal{E}'_v(\mathsf{unfold}(v)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{val}(\mathsf{unfold}(x))
\end{aligned}$$

Computations

$$\mathcal{E}_v : \Lambda\mathrm{MIL}_{T_\varepsilon(A)} \to \mathbf{MIL}_{T_\varepsilon(A)}$$

$$\begin{aligned}
\mathcal{E}_v(\mathsf{app}(v,w)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow \mathcal{E}'_v(w)\ \mathsf{in}\ \mathsf{app}(x,y)\\
\mathcal{E}_v(\mathsf{val}(v)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{val}(x)\\
\mathcal{E}_v(\mathsf{raise}(E)) &= \mathsf{raise}(E)\\
\mathcal{E}_v(\mathsf{try}\ x \Leftarrow m\ \mathsf{in}\ n\ \mathsf{unless}\ H) &= \mathsf{try}\ x \Leftarrow \mathcal{E}_v(m)\ \mathsf{in}\ \mathcal{E}_v(n)\ \mathsf{unless}\ \mathcal{E}^H_v(H)\\
\mathcal{E}_v(\mathsf{case}\ v\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)) &= \ \mathsf{let}\ y \Leftarrow \mathcal{E}'_v(v)\\
&\quad\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ (x_1 \Rightarrow \mathcal{E}_v(n_1) \mid x_2 \Rightarrow \mathcal{E}_v(n_2))\\
\mathcal{E}_v(\mathsf{new}) &= \mathsf{new}\\
\mathcal{E}_v(\mathsf{read}(v)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{read}(x)\\
\mathcal{E}_v(\mathsf{write}(v,w)) &= \mathsf{let}\ x \Leftarrow \mathcal{E}'_v(v)\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow \mathcal{E}'_v(w)\ \mathsf{in}\ \mathsf{write}(x,y)\\
\text{where} &\quad x,y \text{ are fresh}
\end{aligned}$$

Handlers

$$\mathcal{E}^H_v : \mathbb{E} \times \Lambda\mathrm{MIL}\ \mathit{list} \to \mathbb{E} \times \mathbf{MIL}\ \mathit{list}$$

$$\mathcal{E}^H_v(H) = \mathit{map}\ (\lambda(E,n).(E,\mathcal{E}_v(n)))\ H$$

Figure 5.8: Call-by-value embedding of ΛMIL into **MIL**

Analogously to *bindCmp*, we find it useful to define a parameter *bindVal* for naming values. Calling *bindVal*(v) ensures that v is bound to a variable if it is non-atomic.

$$\begin{aligned} \mathit{bindVal}(a) &= a \\ \mathit{bindVal}(v) &= \mathcal{S}(\lambda\kappa.\mathit{tail}(\mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ {<}\kappa\ x{>})) \end{aligned}$$

$$\begin{aligned} \mathit{tail} &: \mathbf{MIL} \to \mathbf{MIL} \\ \mathit{tail}(\mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ \mathsf{val}(x)) &= v \\ \mathit{tail}(m) &= m \end{aligned}$$

The function *tail* performs *tail-call elimination*. It ensures that values are not named when they appear in tail position. We write $\downarrow_n$ for $(\mathit{bindVal} \circ \downarrow)$. The semantics is the same as that of Figure 5.5 but restricted to **MIL**. It appears in Figure 5.9. The semantic parameters appear in Figure 5.10. Whenever a value v is reified, and an atom is expected, an atomicity check is performed. If v is atomic then it is not named, and if v is non-atomic then it is named. Reification is defined in Figure 5.11. Again, non-atomic values are named.

Remark The astute reader may have noticed the fact that unlike *bindCmp*, the parameter *bindVal* does not take a type argument. Technically *bindVal* should take a type, because we need to know the type of the value. In order to output a term of the form $\mathsf{letval}\ x^A \Leftarrow v\ \mathsf{in}\ n$, we need to know the type A. In general, we do not have this type. Fortunately, in the implementation we can cheat. In concrete MIL, the term $\mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ n$ is a real term — not just syntactic sugar for $\mathsf{let}\ x \Leftarrow \mathsf{val}(v)\ \mathsf{in}\ n$. Furthermore, it does not require x to have a type annotation.

5.7.2 Limitations

The normalisation by evaluation algorithm we have just described can introduce a significant amount of redundancy. One of the reasons for naming non-atomic values is so that they can be used many times without significantly increasing the size of the term. This is particularly true of functions. One might wish that any sharing of values in the source term be preserved in the normal form. But inevitably, this information must

$$\begin{aligned}
[\![Int]\!] &= \mathbf{MIL}\text{-}ne_{Int} \\
[\![A\ ref]\!] &= \mathbf{MIL}\text{-}ne_{A\ ref} \\
[\![\mathbf{1}]\!] &= star \\
[\![A \to T_\varepsilon(B)]\!] &= (([\![A]\!] \to [\![T_\varepsilon(B)]\!]) \times \tau_A) + \mathbf{MIL}\text{-}ne_{A \to T_\varepsilon(B)} \\
[\![A \times B]\!] &= ([\![A]\!] \times [\![B]\!]) + \Lambda\mathrm{MIL}\text{-}ne_{A \times B} \\
[\![A + B]\!] &= \mathbf{MIL}\text{-}ne_{A+B} \\
\\
[\![T_\varepsilon(A)]\!] &= Comp_\varepsilon([\![A]\!])
\end{aligned}$$

Figure 5.9: Semantics with fixed constants for MIL

be lost, as the semantics models all β-equivalent terms, some of which may be highly redundant.

It should be possible to rediscover some amount of sharing using some form of common-subexpression elimination. We have written a prototype implementation, in which we remove a certain amount of redundancy using a generalisation of the techniques of Altenkirch et al. [ADHS01] and Balat et al. [BCF04], for performing normalisation by evaluation with sums.

5.8 Implementations

This chapter has outlined our approach to implementing normalisation by evaluation algorithms for SML.NET. We have implemented all of the algorithms described, together with a number of variations.

- We have implemented versions of the algorithms of §5.3 and §5.4 using: the continuation monad, the accumulation monad, state and delimited continuations.

- We have implemented extensional and intensional variants of the absorbing algorithm.

- We have extended the algorithms of §5.3 and §5.4 with sums and exceptions. In

$$
\begin{aligned}
\mathit{star}\ &\text{is uninterpreted}\\
\mathit{constant}^A(c) &= c^A\\
\mathit{lam}^A(f) &= (f, A)\\
\mathit{app}((f,A),s) &= f\ s\\
\mathit{app}(e, s^A) &= \mathsf{app}(\mathit{bindVal}(e), \downarrow s)\\
\mathit{pair}(s_1, s_2) &= (s_1, s_2)\\
\mathit{proj}_i(s_1, s_2) &= s_i\\
\mathit{proj}_i(e) &= \mathsf{proj}_i(\mathit{bindVal}(e))\\
\mathit{inj}_i(s) &= \mathsf{inj}_i(\downarrow_n s)\\
\mathit{fold}_{\mu X.A}(s) &= \mathsf{fold}_{\mu X.A}(\downarrow_n s)\\
\mathit{unfold}(s) &= \mathsf{unfold}(\downarrow_n s)\\
\mathit{case}(v, f_1, f_2) &= \mathsf{case}\ \downarrow_n v\ \mathsf{of}\ (x_1 \Rightarrow \downarrow f_1(x_1) \mid x_2 \Rightarrow \downarrow f_2(x_2))\\
\mathit{val}(v)\ &\text{is uninterpreted}\\
\mathit{let}^A(\mathit{val}(v), f) &= f\ v\\
\mathit{let}^A(e, f) &= \mathit{bindCmp}^A(e, f)\\
\mathit{raise}(E) &= \mathsf{raise}(E)\\
\mathit{try}^A(s, f, H) &= \mathsf{try}\ \downarrow s \Leftarrow x^A\ \mathsf{in}\ \downarrow f(x)\ \mathsf{unless}\ \downarrow H\\
\mathit{read}(s) &= \mathsf{read}(\downarrow s)\\
\mathit{write}(s, s') &= \mathsf{write}(\downarrow_n s, \downarrow_n s')\\
\mathit{new} &= \mathsf{new}\\
\\
\mathit{Comp}_\varepsilon([\![A]\!]) &= \mathit{val}([\![A]\!]) + \mathbf{MIL}\text{-}\mathit{ne}_{T_\varepsilon(A)}\\
\\
\mathit{collect}(t) &= <t>\\
\mathit{bindCmp}^A(e, f) &= \mathcal{S}(\lambda\kappa.\mathsf{let}\ x^A \Leftarrow e\ \mathsf{in}\ \mathit{collect}(\lambda().\kappa(f\ x))), \quad x^A\ \text{fresh}
\end{aligned}
$$

Figure 5.10: Semantic parameters with fixed constants for MIL

$$\downarrow : [\![\, \Lambda\text{MIL} \,]\!] \to \Lambda\text{MIL-}nf$$

$$\begin{aligned}
\downarrow e &= e \\
\downarrow (f, A) &= \mathsf{lam}(x^A, collect(\lambda().\downarrow (f\ x))), \quad x^A \text{ fresh} \\
\downarrow star &= * \\
\downarrow (s_1, s_2) &= \mathsf{pair}(\downarrow_n s_1, \downarrow_n s_2) \\
\downarrow val(s) &= <\mathsf{val}(\downarrow s)> \\
norm(e) &= \downarrow [\![\, e \,]\!]_{\uparrow}
\end{aligned}$$

Figure 5.11: Intensional NBE for MIL

the first instance this was using shift and reset. Subsequently we have also implemented a version with fixed constants which uses state and the zipper structure as described in §4.6.

- We have implemented the straight to MIL algorithm using delimited continuations.

In the next chapter we assess the performance of these algorithms and compare them with rewriting-based normalisation algorithms.

Chapter 6

Performance and analysis

In this chapter we assess the efficiency of normalisation by evaluation algorithms on MIL terms corresponding to actual ML programs. We do this by benchmarking against a range of rewriting-based normalisation algorithms. We consider the normalisation problems of the previous chapter: unknown terms as absorbing values or as fixed constants, with and without η-expansion, with sums, etc.

For each normalisation problem we begin with a naïve algorithm which just performs a depth-first traversal of the term structure contracting redexes recursively, using an auxiliary substitution function. Straightforward optimisations, such as using an environment and not reducing inside abstractions that are never applied, are then added. For each normalisation problem this process gives rise to a *spectrum* of normalisation algorithms ranging from the naïve algorithm up to normalisation by evaluation.

The benchmarks range from a basic quicksort program, which just sorts a list of integers, through to a full bootstrap of the SML.NET compiler. The release version of SML.NET is compiled under the SML/NJ [smlb] compiler. We have also ported SML.NET to the MLton [mlt] compiler. We compare algorithms compiled under SML/NJ against algorithms compiled under MLton. All tests were performed on a PC with an AMD Athlon 1.4Ghz CPU and 512MB of RAM.

If we take a closer look at the optimisations made to the most naïve normalisation algorithm, it becomes apparent that each optimisation moves the algorithm closer to normalisation by evaluation. In fact the normalisation by evaluation algorithm can be obtained by a series of straightforward program transformations. This provides yet another angle on normalisation by evaluation. This perspective is closely related to that of Ager et al. [ABDM03] in their work on obtaining abstract machines from evaluation functions and vice-versa. Program transformations give a way to move neatly back and forth between algorithms. This could be especially useful if a particular extension is most easily expressed in one algorithm, but also of use in another. For instance, there are various normalisation by evaluation algorithms which handle sums, but it is not entirely clear what their more naïve counterparts would look like. Program transformation provides a means to find out.

The rest of this chapter is structured as follows. In §6.1 we mention some related work. In §6.2 we introduce a spectrum of normalisation algorithms. In §6.3 we discuss our framework for obtaining results. In §6.4 we present results for normalisation using absorbing values for unknowns. In §6.5 we present results for normalisation using fixed constants for unknowns. In §6.6 we outline some of the practical difficulties we encountered in trying to obtain meaningful results. In §6.7 we summarise our main results. Finally, in §6.8 we discuss how to obtain normalisation by evaluation algorithms by program transformation.

6.1 Related work

Berger et al. [BES98] measure the speed of normalisation by evaluation for simply-typed λ-calculus using a Scheme implementation. They compare: normalisation by evaluation with the native Scheme evaluator; normalisation by evaluation with a hand-coded evaluator; and a naïve recursive normalisation algorithm. Their benchmarks are λ-encodings of iterated functions that were deliberately chosen because they take many β-reductions to normalise, but always reduce to the identity. Their normalisation by evaluation algorithms are extensional (giving long normal forms), whereas their naïve algorithm only performs β-reduction[1]. The algorithms give the same normal forms because the benchmarks do not contain any η-redexes. Their normalisation by evaluation algorithms are much faster than the naïve algorithm. Interestingly, the normalisation by evaluation algorithm which uses the native evaluator is faster than the

[1] In fact their naïve algorithm is just a Scheme implementation of the algorithm of Figure 6.1

$$\begin{aligned}
\mathit{norm\text{-}naive} &: \Lambda u \to \Lambda u\text{-}\mathit{nf} \\
\mathit{norm\text{-}naive}(x) &= x \\
\mathit{norm\text{-}naive}(\mathsf{lam}(x,m)) &= \mathsf{lam}(x, \mathit{norm\text{-}naive}(m)) \\
\mathit{norm\text{-}naive}(\mathsf{app}(m,n)) &= \mathit{app}(\mathit{norm\text{-}naive}(m), \mathit{norm\text{-}naive}(n)) \\
\\
\mathit{app}(\mathsf{lam}(x,m),n) &= \mathit{norm\text{-}naive}(m[x := n]) \\
\mathit{app}(m,n) &= \mathsf{app}(m,n)
\end{aligned}$$

Figure 6.1: A naïve normalisation algorithm

one which uses a hand-coded evaluator.

Our tests are much more comprehensive: we use non-trivial benchmarks compiled from actual ML programs; our object language, λMIL, is considerably more complex than simply-typed λ-calculus; and we cover a wide range of different normalisation algorithms.

6.2 A spectrum of normalisation algorithms

To simplify the presentation we illustrate a spectrum of algorithms, from most naïve to normalisation by evaluation, using the untyped λ-calculus with just β-conversion. The same optimisations are easily adapted to λMIL.

The starting point is the naïve applicative-order normalisation algorithm defined in Figure 6.1. The function *norm-naive* traverses the term structure depth-first. The only interesting case is an application $\mathsf{app}(m,n)$. First m and n are normalised, then if the normal form of m is a lambda, a β-reduction is performed. It is easy to see that if *norm-naive*(e) terminates then it will return the normal form of e. Of course, the untyped λ-calculus is not strongly normalising with respect to β-reduction so *norm-naive* may not terminate.

A simple optimisation is to pass an environment around instead of explicitly performing substitution (Figure 6.2). When a β-redex is encountered (the first-line of *app*),

$$\rho : \mathbf{V} \to \Lambda u\text{-}nf$$

$$\begin{aligned}
norm_\rho &: \Lambda u \to \Lambda u\text{-}nf \\
norm_\rho(x) &= \rho\ x \\
norm_\rho(\mathsf{lam}(x,m)) &= \mathsf{lam}(x, norm_\rho(m)) \\
norm_\rho(\mathsf{app}(m,n)) &= app(norm_\rho(m), norm_\rho(n), \rho)
\end{aligned}$$

$$\begin{aligned}
app(\mathsf{lam}(x,m), n, \rho) &= norm_{\rho[x\mapsto n]}(m) \\
app(m,n,\rho) &= \mathsf{app}(m,n)
\end{aligned}$$

$$\uparrow x = x$$

$$norm\text{-}env(m) = norm_{\uparrow}(m)$$

Figure 6.2: Normalisation with an environment

the argument is bound in the environment. Whenever the bound variable is encountered its value is looked up in the environment. The initial environment $\uparrow$ is simply the identity.

Note that it is unnecessary to normalise inside lambdas which are applied. The β-reduction step will perform the normalisation anyway. We observe that the normalisation function can be decomposed into two functions:

- The first function *wnf* reduces to *weak normal form* (Λu-*wnf*), reducing everywhere except inside unapplied lambdas.
- The second function *nf* reduces inside the remaining lambdas to give a normal form.

Figure 6.3 shows a version *norm-wnf0* without environments.

Reintroducing environments is a bit subtle. The problem is that we need to record the fact that variables must eventually be substituted for inside a lambda, without actually performing the substitution. The solution is to use *closures* [Lan64], which are

$$
\begin{aligned}
wnf &: \Lambda u \to \Lambda u\text{-}wnf \\
wnf(x) &= x \\
wnf(\mathsf{lam}(x,m)) &= \mathsf{lam}(x,m) \\
wnf(\mathsf{app}(m,n)) &= app(wnf(m), wnf(n)) \\
\\
app(\mathsf{lam}(x,m),n) &= wnf(m[x := n]) \\
app(m,n) &= \mathsf{app}(m,n) \\
\\
nf &: \Lambda u\text{-}wnf \to \Lambda u\text{-}nf \\
nf(x) &= x \\
nf(\mathsf{app}(m,n)) &= \mathsf{app}(nf(m), nf(n)) \\
nf(\mathsf{lam}(x,m)) &= \mathsf{lam}(x, nf(wnf\, m)) \\
\\
norm\text{-}wnf0 &= nf \circ wnf
\end{aligned}
$$

Figure 6.3: Normalisation with weak normal forms and no environment

lambda abstractions augmented with an environment:

$$
\begin{array}{ll}
(\Lambda u) & m,n ::= x \mid \mathsf{app}(m,n) \mid \mathsf{lam}(x,m) \\
(\Lambda u_{clos}) & p,q ::= x \mid \mathsf{app}(p,q) \mid \mathsf{closure}_{\rho}(x,m)
\end{array}
$$

Using closures we obtain the normalisation algorithm of Figure 6.4. Closures are produced when *wnf* is applied to a λ-abstraction. The λ-abstraction is augmented with the current environment. Closures are consumed by *nf*. The environment of the closure is used to obtain the weak normal form of the body.

We observe that *norm-wnf* looks rather like a normalisation by evaluation algorithm; *wnf* plays a similar role to the evaluation function, and nf plays a similar role to $\downarrow$. Bearing in mind that: (i) closures encode higher-order functions, and (ii) our parameterised semantics for Λu uses higher-order functions to interpret λ-abstractions, we transform the closures into higher-order functions. This transformation is the inverse of closure conversion [App92]. The resulting algorithm appears in Figure 6.5. *u*

$$\rho : \mathbf{V} \to \Lambda_{clos}$$

$$\begin{aligned}
wnf_\rho &: \Lambda u \to \Lambda u_{clos} \\
wnf_\rho(x) &= \rho\ x \\
wnf_\rho(\mathsf{lam}(x,m)) &= \mathsf{closure}_\rho(x,m) \\
wnf_\rho(\mathsf{app}(m,n)) &= app(wnf_\rho(m), wnf_\rho(n))
\end{aligned}$$

$$\begin{aligned}
app(\mathsf{closure}_\rho(x,m),p) &= wnf_{\rho[x\mapsto p]}(m) \\
app(p,q) &= \mathsf{app}(p,q)
\end{aligned}$$

$$\begin{aligned}
nf &: \Lambda u_{clos} \to \Lambda u\text{-}nf \\
nf(x) &= x \\
nf(\mathsf{app}(p,q)) &= \mathsf{app}(nf(p), nf(q)) \\
nf(\mathsf{closure}_\rho(x,m)) &= \mathsf{lam}(x, nf(wnf_\rho m))
\end{aligned}$$

$$\uparrow x = x$$

$$norm\text{-}wnf(m) = nf(wnf_\uparrow(m))$$

Figure 6.4: Normalisation with weak normal forms and closures

ranges over $[\![\Lambda u\text{-}ne]\!]$ and f ranges over functions.

We have renamed $wnf(\cdot)$ as $[\![\cdot]\!]$, and nf as $\downarrow$. Note that, unlike closures, higher order functions do not include the name of the bound variable — hence the introduction of the fresh variable x in $\downarrow$. We could embed the name in the semantics, but then it would not model α-conversion.

norm-nbe′ is indeed a normalisation by evaluation algorithm. Expressed using the parameterised semantics of Figure 2.11 it becomes:

$$
\begin{aligned}
[\![\, \Lambda u \,]\!] &\simeq [\![\, \Lambda u\text{-}ne \,]\!] + ([\![\, \Lambda u \,]\!] \to [\![\, \Lambda u \,]\!]) \\
[\![\, \Lambda u\text{-}ne \,]\!] &\simeq \mathbf{V} + ([\![\, \Lambda u\text{-}ne \,]\!] \times [\![\, \Lambda u \,]\!])
\end{aligned}
$$

$$\rho : \mathbf{V} \to [\![\, \Lambda u \,]\!]$$

$$
\begin{aligned}
[\![\, \cdot \,]\!]_\rho &: \Lambda \to [\![\, \Lambda u \,]\!] \\
[\![\, x \,]\!]_\rho &= \rho\ x \\
[\![\, \mathsf{lam}(x,m) \,]\!]_\rho &= \lambda v.[\![\, m \,]\!]_{\rho[x \mapsto v]} \\
[\![\, \mathsf{app}(m,n) \,]\!]_\rho &= app([\![\, m \,]\!]_\rho, [\![\, n \,]\!]_\rho)
\end{aligned}
$$

$$
\begin{aligned}
app(f,s) &= f\ s \\
app(u,s) &= (u,s)
\end{aligned}
$$

$$
\begin{aligned}
\downarrow &: [\![\, \Lambda u \,]\!] \to \Lambda\text{-}nf \\
\downarrow x &= x \\
\downarrow (u,s) &= \mathsf{app}(\downarrow u, \downarrow s) \\
\downarrow f &= \mathsf{lam}(x, \downarrow f\ x), \quad x \text{ fresh}
\end{aligned}
$$

$$\uparrow x = x$$

$$norm\text{-}nbe'(m) = \downarrow ([\![\, m \,]\!]_\uparrow)$$

Figure 6.5: Intensional NBE with a higher-order semantics for neutral terms

$$
\begin{aligned}
[\![\Lambda u]\!] &\simeq [\![\Lambda u\text{-}ne]\!] + ([\![\Lambda u]\!] \to [\![\Lambda u]\!]) \\
[\![\Lambda u\text{-}ne]\!] &\simeq \mathbf{V} + ([\![\Lambda u\text{-}ne]\!] \times [\![\Lambda u]\!]) \\
\\
lam(f) &= f \\
\\
app(f, s) &= f\ s \\
app(u, s) &= (u, s) \\
\\
\downarrow : [\![\Lambda u]\!] &\to \Lambda\text{-}nf \\
\downarrow x &= x \\
\downarrow (u, s) &= \mathsf{app}(\downarrow u, \downarrow s) \\
\downarrow f &= \mathsf{lam}(x, \downarrow f\ x) \quad (x \text{ fresh}) \\
\\
\uparrow x &= x \\
\\
norm\text{-}nbe'(m) &= \downarrow ([\![m]\!]_{\uparrow})
\end{aligned}
$$

where u ranges over $[\![\Lambda u\text{-}ne]\!]$ and f ranges over functions.

This is rather close to the normalisation by evaluation algorithms in the rest of this thesis. The main difference is that neutral terms have a higher-order semantics, rather than being interpreted as themselves. We now change the interpretation of neutral terms to be:

$$
[\![\Lambda u\text{-}ne]\!] \simeq \mathbf{V} + ([\![\Lambda u\text{-}ne]\!] \times \Lambda u)
$$

and move the call $\downarrow s$ inside *app*. Note that $\downarrow$ has become the identity on neutral terms. Also note that $[\![\Lambda u\text{-}ne]\!]$ is now isomorphic to $\Lambda u\text{-}ne$, so we can simply interpret neutral terms as themselves. This gives the standard normalisation by evaluation algorithm of Figure 4.3:

$$[\![\Lambda u]\!] \simeq \Lambda u\text{-}ne + ([\![\Lambda u]\!] \to [\![\Lambda u]\!])$$

$$lam(f) = f$$

$$\begin{aligned} app(f, s) &= f\ s \\ app(u, s) &= \mathsf{app}(u, \downarrow s) \end{aligned}$$

$$\begin{aligned} \downarrow &: [\![\Lambda u]\!] \to \Lambda u\text{-}nf \\ \downarrow u &= u \\ \downarrow f &= \mathsf{lam}(x, \downarrow f\ x) \quad (x \text{ fresh}) \end{aligned}$$

$$\uparrow x = x$$

$$norm\text{-}nbe(m) = \downarrow ([\![m]\!]_{\uparrow})$$

where u ranges over Λu-*ne* and f ranges over functions. We now have a spectrum of normalisation algorithms for β-reduction on Λu-terms ranging from *norm-naive* to *norm-nbe*.

6.3 Obtaining the results

6.3.1 Benchmark programs

We use several ML source programs for benchmarking. The first five benchmarks are demos distributed with SML.NET.

- `sort` simply sorts a list of integers using quicksort (~ 70 lines of ML code).
- `xq` is an interpreter for an XQuery-like language for querying XML documents (~ 1,300 lines of ML code).
- `mllex` is a port of SML/NJ's ML-Lex utility (~ 1,400 lines of ML code).
- `raytrace` is a port to SML of the winning entry from the Third Annual ICFP Programming Contest (~ 2,500 lines of ML code).

- `mlyacc` is a port of SML/NJ's ML-Yacc utility (~ 6,200 lines of ML code).

The remaining benchmarks are much larger.

- `hamlet` is Andreas Rossberg's SML interpreter (~ 20,000 lines of ML code).
- `bootstrap` is SML.NET compiling itself (~ 80,000 lines of ML code).

We shall give the times (in milliseconds) for normalising the ΛMIL term for each ML program, where the normalisation algorithms have been compiled under both SML/NJ and under MLton.

6.3.2 Interfacing with SML.NET

SML.NET has an extensible interactive environment for coordinating the compilation of programs. It includes a *make* command which compiles an ML program. We have adapted the *make* command to create a new *makemil* command. The *makemil* command uses the frontend to generate MIL code which is then normalised. The choice of which normalisation algorithms to perform and the value of various other parameters is configurable through the interactive environment. One of the parameters *makemil.factor* specifies a repeat factor for normalisation. For some of the smaller examples it is necessary to set this to a value higher than one in order for the timing to be long enough to be measured accurately. The timing is performed using SML.NET's built-in timing mechanism, which is normally used for reporting compile times.

6.3.3 Intensional versus extensional normalisation by evaluation

It would have been interesting to collect and analyse comprehensive results for both intensional and extensional variants of normalisation by evaluation. However, due to time constraints this was not possible; instead, we have chosen to focus on intensional variants of normalisation by evaluation. Some of the reasons for choosing intensional over extensional normalisation by evaluation are:

- η-expansion does not terminate for recursive types.
- It is not generally desirable to perform η-expansion in a compiler.

- If desired, η-reduction can be easily added to intensional normalisation by evaluation algorithms.

- Extensional normalisation by evaluation with fixed constants for unknowns depends on building type-inference into the semantics. This is not difficult to do in principle, but we have not implemented such an algorithm.

6.4 Absorbing values for unknowns

Following Chapter 5 we begin by recording results for normalisation with absorbing values for unknowns. The normalisation problem is an intensional version of the one described in §5.3. In order to try to reduce the amount of the term which is designated *unknown*, it is monomorphised before being normalised. We record only the time taken for normalisation. Unfortunately the compiler's monomorphisation phase also performs some reductions which could be performed by the normalisation algorithms. Because of this, and the fact that absorbing normalisation is not semantics-preserving, we shall treat the results of this section with some caution.[2]

The normalisation algorithms correspond to those of §6.2 extended to ΛMIL.

- *naive** is a basic applicative-order normalisation algorithm.

- *env** uses environments.

- *wnf** uses closures.

- *nbe** uses normalisation by evaluation with the continuation monad.

Table 6.1 and Table 6.2 show the normalisation times using absorbing values for unknowns under SML/NJ and MLton. A chart of the results appears in Figure 6.6.

These results show that simply using an environment invariably leads to an order of magnitude speed-up over the naïve algorithm. Normalising via weak normal form,

[2] The primary reason for doing the tests with absorbing values at all is that they offer a quick (and somewhat dirty) way of checking that normalisation by evaluation is competitive with other algorithms. If normalisation by evaluation were not as fast as a naïve algorithm, in this relatively simple case, then it would probably not have been worth pursuing further.

Table 6.1: Normalisation times using absorbing values under SML/NJ

(ms)	*naive**	*env**	*wnf**	*nbe**
`sort`	111	1.13	0.44	0.90
`xq`	452	2.92	1.63	3.29
`mllex`	371	2.90	1.40	2.40
`raytrace`	779	4.46	1.97	3.77
`mlyacc`	2223	10.61	5.50	11.81
`hamlet`	5056	13.92	6.81	11.71
`bootstrap`	88846	85.13	68.12	142.93

Table 6.2: Normalisation times using absorbing values under MLton

(ms)	*naive**	*env**	*wnf**	*nbe**
`sort`	54	1.17	0.30	0.33
`xq`	176	2.65	0.78	1.15
`mllex`	195	2.47	0.59	0.76
`raytrace`	310	3.44	0.69	0.98
`mlyacc`	906	6.57	2.45	4.05
`hamlet`	1792	9.71	2.10	3.30
`bootstrap`	14360	27.23	11.61	21.23

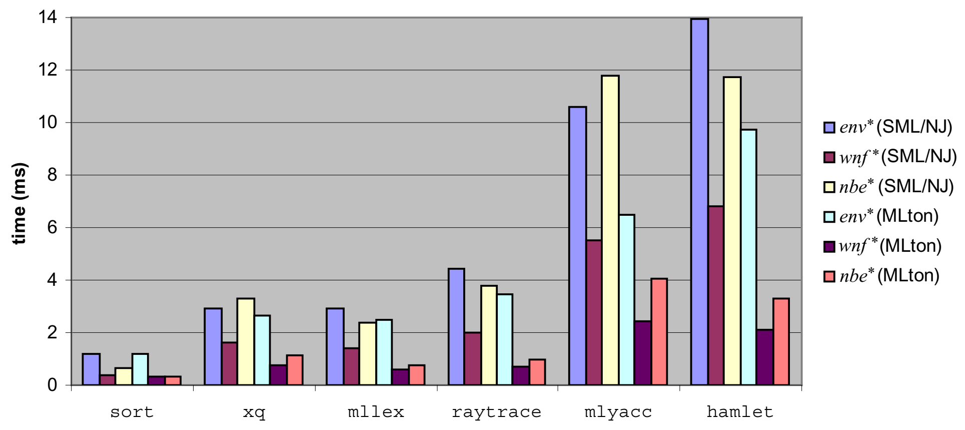

Figure 6.6: Normalisation times using absorbing values

with environments and closures, is also of some benefit, but not as much as we might expect. The *nbe** algorithm is actually slightly slower than *wnf**, and even sometimes slower than *env**. We conjecture that the results are biased towards *env** because the pre-processing phase removes a large amount of redundant code, and this is one of the problems *wnf** (and *nbe**) address.

6.5 Fixed constants for unknowns

For the rest of our tests we use fixed constants for unknowns. This preserves the semantics, and allows for many more reductions than absorbing normalisation. No preprocessing is performed, but the raw MIL term generated by the frontend is fed into the normalisation algorithm.

6.5.1 Choosing an interpretation for computations

We now compare the performance of normalisation by evaluation algorithms using different interpretations for computations. The normalisation problem is essentially that of §5.4. It is extended slightly in that $+.\beta i$-reduction is performed. This is an easy extension. However, adding $+.T.$CC-reduction as well is considerably more complex. We do this in §6.5.3. We compare the normalisation by evaluation algorithms with a normalisation algorithm *wnf* which uses closures.

- nbe_c uses delimited continuations.
- nbe_s uses a state cell to store a list of bindings.
- nbe_f uses a state cell to store a functional representation of a list of bindings.
- nbe_{mc} uses the continuation monad with answer type computation terms.
- nbe_{ms} uses an accumulation monad over a list of bindings.
- nbe_{mf} uses an accumulation monad over a functional representation of a list of bindings.

Table 6.3: Normalisation times for different interpretations of computations

(ms)	SML/NJ	MLton
wnf	2993	350
nbe_c	2913	143186
nbe_s	2980	391
nbe_f	2749	370
nbe_{mc}	2980	370
nbe_{ms}	2809	400
nbe_{mf}	2950	391
nbe_r	2783	390

- nbe_r uses higher-order rewriting to perform $T.T.$CC-reduction. This is the algorithm one obtains by 'refunctionalising' *wnf*.

For this test we use the `hamlet` benchmark. The results appear in Table 6.3. With one exception, the results are strikingly similar. Excluding delimited continuations under MLton, the choice of interpretation for computations does not significantly affect the normalisation time. Furthermore, the normalisation by evaluation algorithms take about the same time as *wnf*.

It is not particularly surprising that delimited continuations are so slow under MLton, as it was not designed with first-class continuations in mind. In contrast, the design of SML/NJ was strongly influenced by work on continuations [App92], and call/cc was a natural extension.

Grobauer and Yang [GY99] describe a slight modification of TDPE in which one can remove some calls to reset. They use this modification to obtain monomorphically-typed instances of shift and reset, which enables them to perform the Second Futamura transformation on their TDPE algorithm. We tried applying their technique to nbe_c, but it had no discernible effect on the performance.

6.5.2 Comparing normalisation by evaluation against other algorithms

Now we compare the performance of normalisation by evaluation with other normalisation algorithms across all of our benchmarks. We chose nbe_f, as it performs well under both SML/NJ and MLton. We could just as well have chosen any of the other normalisation by evaluation algorithms apart from nbe_c. In addition to *wnf* we have applicative-order normalisation algorithms: *naive*, which does not use environments; and *env*, which does.

The algorithm *naive* is so slow that we omit results for it. Even on the `sort` benchmark it did not terminate after our threshold of 10^6 milliseconds. The results for SML/NJ appear in Table 6.4 and those for MLton appear in Table 6.5. The `bootstrap` benchmark takes longer than 10^6 milliseconds, even with the more sophisticated normalisation algorithms. Figure 6.7 shows a chart of the results.

The algorithm *env* is typically several times slower than *wnf* and nbe_f. The algorithms *wnf* and nbe_f are roughly the same speed on all the benchmarks. This is encouraging because it indicates that normalisation by evaluation is indeed fast compared with other normalisation algorithms. It performs roughly the same as an optimised normalisation algorithm. It also indicates that both the compilers are doing a good job of compiling higher-order representations.

We believe that the long normalisation times for `bootstrap` are due to the blow-up in code size caused by unrestricted inlining. Even the smaller `hamlet` example gives an order of magnitude increase in the size of the target MIL as compared with the source. This might be partially alleviated by removing some redundancy as suggested in §5.7.2. Alternatively, the blow-up can be eliminated by using shrinking reductions as described in Chapter 7, but this gives a very different flavour of algorithm to the normalisation by evaluation algorithms discussed so far.

Remark Because SML.NET is a whole program compiler, it always includes the basis library in the source MIL term; so even small ML programs translate to a relatively large MIL term. The vast majority of our simplest benchmark, `sort`, is just the basis library. However, most of the basis is not used. The hope would be that most of the

Table 6.4: Normalisation times using fixed constants under SML/NJ

(ms)	*env*	*wnf*	nbe_f
`sort`	131	95	115
`xq`	932	206	197
`mllex`	8212	861	860
`raytrace`	874	497	463
`mlyacc`	4137	802	991
`hamlet`	17335	2993	2749
`bootstrap`	$> 10^6$	$> 10^6$	$> 10^6$

Table 6.5: Normalisation times using fixed constants under MLton

(ms)	*env*	*wnf*	nbe_f
`sort`	39	15	19
`xq`	133	44	44
`mllex`	805	121	125
`raytrace`	101	118	121
`mlyacc`	501	135	133
`hamlet`	2453	350	370
`bootstrap`	$> 10^6$	$> 10^6$	$> 10^6$

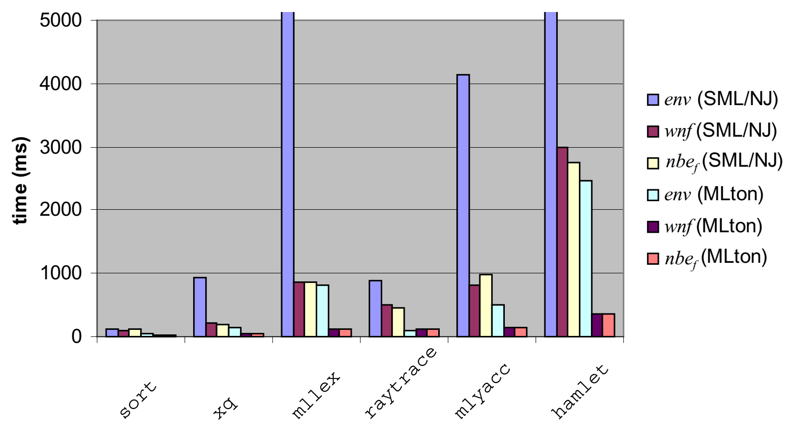

Figure 6.7: Normalisation times using fixed constants

Table 6.6: Normalisation times for sums

(ms)	wnf^+	nbe_c^+	nbe_s^+	nbe_r^+
`sort` (SML/NJ)	779	725	1953	648
`xq` (SML/NJ)	1292	1152	4938	1051
`sort` (MLton)	143	3055	781	104
`xq` (MLton)	230	5398	1953	167

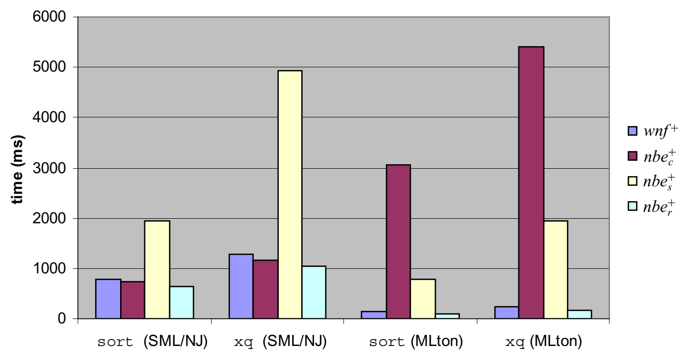

Figure 6.8: Normalisation times for sums

source term is immediately discarded (there is no point in optimising dead code). Indeed, not reducing inside lambdas (as in the closure-based normalisation algorithms, and the normalisation by evaluation algorithms) achieves this aim. This is reflected in the benchmarks that use fixed constants.

6.5.3 Sums

We have extended some of the algorithms of the previous section to perform $+.T$.CC-reduction as described in §4.5 and §5.5.

- wnf^+ is an extension of *wnf* which performs $+.T$.CC-reduction.
- nbe_c^+ uses delimited continuations as described in §4.5.2.
- nbe_s^+ uses a single reference cell and a zipper as described in §4.6.

- nbe_r^+ uses higher-order rewriting to perform $T.T$.CC- and $+.T$.CC-reduction. It is a 'refunctionalised' version of wnf^+.

As remarked in §5.5, $+.T$.CC-reduction does, in practice, lead to exponential growth in the size of terms. Thus, we were only able to obtain results for two of the benchmarks: `sort` and `xq`. The results appear in Table 6.6. Figure 6.8 shows a chart of the results.

Under SML/NJ the timings for wnf^+, nbe_c^+ and nbe_r^+ are roughly the same. Under MLton the same is true of wnf^+ and nbe_r^+. Under both SML/NJ and MLton nbe_s^+ is slower than the fastest algorithms. This is not surprising given that it does duplicate some computation. Under MLton nbe_s^+ is actually faster than nbe_c^+ due to MLton's slow implementation of call/cc.

Remark Under both compilers the nbe_r^+ function appears to be slightly faster than wnf^+. This is rather curious given that wnf^+ is a defunctionalised variant of nbe_r^+. However, there is not really enough data, and the difference is not large enough, to be sure.

6.5.4 Size of terms

In theory unrestricted β-reduction could lead to a non-elementary blow-up in the size of terms. Adding sums and $+.T$.CC-reduction makes matters even worse, and seems to be a bigger problem in practice. SML.NET includes a metric for the size of MIL terms.[3] We use this metric to compare the size of terms before and after normalisation. Roughly, the metric gives one for the size of syntax constructors, one for constants and zero for variables.

The sizes are shown in Table 6.7 where:

- source is the size of the source term.

- source* is the size of the source term after monomorphisation, where unknown terms are given size 0.

[3]This is used internally to decide when to perform certain optimisations such as inlining.

Table 6.7: Size of terms before and after normalisation

(ms)	source	source*	norm	norm*	norm+
`sort`	9885	1007	8407	3	243069
`xq`	13967	2820	17822	55	290007
`mllex`	14494	3720	125519	18	?
`raytrace`	18145	6627	35583	26	?
`mlyacc`	24239	6397	96123	194	?
`hamlet`	56321	28757	846266	143	?
`bootstrap`	152853	140329	?	866	?

- norm is the size of the normal form where fixed constants are used for unknowns.
- norm* is the size of the normal form where absorbing values are used for unknowns.
- norm+ is the size of the normal form in the presence of $+.T$.CC.

The entries marked ? indicate where a result was not obtained because the algorithm did not terminate within the 10^6 millisecond threshold.

The blow-up due to unrestricted β-reduction can be significant. In the case of `hamlet` the normalised term is over 15 times larger than the source. As expected, $+.T$.CC-reduction has an even more dramatic effect. In the case of `sort` the normalised term is almost 25 times larger than the source term.

As well as showing large increases in size for normalisation with fixed constants for unknowns, Table 6.7 also illustrates how much information is lost by using absorbing values for unknowns. In the case of `sort`, for instance, the normal form is over 3,000 times smaller than original source term.

6.6 Obstacles

We encountered a number of difficulties in trying to collect accurate results. The first problem was that the numbers were being recorded under Windows XP. It is not possible to guarantee that the operating system will not interrupt a process under Windows. By assigning a high priority to the ML processes, and minimising the number of other programs running in the background we were able to reduce interruptions significantly. We do not believe that, after having taken these precautions, the operating system had a significant impact on the accuracy of our results.

The second problem was with memory management and, in particular, garbage collection. It is difficult to predict or influence when garbage collection takes place, and it can have a significant effect on running time. Both SML/NJ and MLton include a function to force garbage collection to take place. By forcing garbage collection immediately before timing each of our algorithms we were able to improve the consistency of the timings.

However, we did observe some unusual patterns. To begin with we configured SML.NET to perform each normalisation algorithm in turn automatically, without requiring any user interaction. Under SML/NJ running each normalisation by evaluation algorithm in turn leads to each successive algorithm being slightly slower than the previous one. When the order is changed the same thing happened. Usually MLton seems to be more predictable, and the times less affected by the order in which the algorithms are run, or how many times they are run. We did observe one anomaly, though. We noticed that nbe_f was taking about half the time of the other normalisation by evaluation algorithms. Further investigation revealed that this was only the case if nbe_f was run immediately after nbe_s. Another factor which can make benchmarking difficult is the complexity of modern hardware. Long pipelines, caches, high memory latencies and concurrency all affect the predictability of performance. In order to get more meaningful results from both SML/NJ and MLton we restarted SML.NET for each timing.

6.7 Summary

We summarise our key results:

- *naive* typically performs orders of magnitude slower than algorithms which use an environment.

- *env* is typically several times slower than wnf.

- For normalisation by evaluation, the choice of interpretation for computations has little effect on performance — with one notable exception:
- Under MLton nbe_c is orders of magnitude slower than the other normalisation by evaluation algorithms.
- Apart from nbe_c under MLton, the normalisation by evaluation algorithms are about the same speed as *wnf*.
- Under both SML/NJ and MLton, nbe_r^+ is an order of magnitude faster than nbe_s^+.
- Under SML/NJ nbe_c^+ is about the same speed as nbe_r^+.
- Under MLton nbe_c^+ is several times slower than nbe_s^+ (presumably because of MLton's inefficient implementation of call/cc).

Our main conclusion is that normalisation by evaluation is indeed efficient — comparing favourably to optimised rewriting-based normalisation algorithms.

6.8 Normalisation by evaluation by program transformation

The development of §6.2 suggests that we might derive normalisation by evaluation algorithms by program transformation. We have actually found this to be a rather useful tool. The idea applies to all of the normalisation problems we have benchmarked in this chapter. The general pattern is to start with an existing algorithm, which is typically operationally inspired, and then to apply a few simple program transformations to obtain a normalisation by evaluation algorithm, with a denotational component.

The key steps, in the case of the λ-calculus and extensions, are: splitting the normalisation function into two stages, introducing environments, and "refunctionalisation". As our results show, the final stage is not necessarily an optimisation, so may not always be of practical interest. However, normalisation by evaluation does provide a rather powerful framework for reasoning about normalisation and semantics, so the final stage is interesting in its own right. As an example, one might start with an existing normalisation algorithm which is known to be correct, and then derive a corresponding normalisation by evaluation algorithm, which is correct by correctness of the program transformations, from which other semantic (e.g. completeness) and syntactic (e.g. confluence) properties can then be extracted.

Chapter 7

Shrinking reductions

The β-normalisation algorithms we have considered so far have reduced all β-redexes in a term. Performing β-reduction corresponds to inlining. It is well-known that unrestricted inlining can lead to a large blow-up in the size of terms. Our testing has shown that this is an issue in practice. For instance the output term from the second `hamlet` benchmark of the previous chapter is more than fifteen times the size of the input. What is more, the worst case time complexity of normalisation in simply-typed λ-calculus is known to be bad (see §2.8). The failure to obtain results at all for the second `bootstrap` benchmark can be attributed to this.

To solve the problem of normalised terms getting too big, functional language compilers such as SML/NJ and SML.NET perform a restricted form of β-reduction in which terms are guaranteed to decrease in size. The *simplify* transformation performs such *shrinking reductions*. In particular, $\rightarrow.\beta$-redexes are reduced only if the bound variable has at most one use.

Appel and Jim [AJ97] describe three algorithms for shrinking reductions. The first 'naïve' and second 'improved' algorithms both have quadratic worst-case time complexity, and the third 'imperative' algorithm is linear, but requires a mutable representation of terms. Appel and Jim did not implement the third algorithm, which does not integrate easily in a mainly-functional compiler. Both SML/NJ and SML.NET use the 'improved' algorithm, which is reasonably efficient in practice. Nevertheless, SML.NET spends a significant amount of time performing shrinking reductions. We have now implemented a variant of the imperative algorithm in SML.NET, and achieved significant speedups.

The rest of the chapter is structured as follows. In §7.1 we formalise shrinking reductions for a variant of MIL, and give some normalisation results. In §7.2 we outline Appel and Jim's shrinking reduction algorithms. In §7.3 we introduce a graph-based representation of MIL terms. This supports movement around the term, both through traversing the underlying tree structure and via connections between variable occurrences. In §7.4 we describe in detail our one-pass imperative algorithm for performing shrinking reductions on MIL terms. Following Appel and Jim this uses an abstract variant of our graph-based representation. In §7.5 we state and informally justify some correctness properties for our algorithm. In §7.6 we compare our one-pass algorithm with the existing shrinking reductions algorithm used in SML.NET. Finally, in §7.7 we summarise.

7.1 Normalisation for shrinking reductions

For the rest of this chapter we use a slightly modified version of simplified MIL. The syntax is as follows:

$$\begin{array}{lrl}
\text{Atoms} & a,b ::= & x \mid * \mid c \\
\text{Values} & v,w ::= & a \mid \mathsf{pair}(a,b) \mid \mathsf{proj}_1(a) \mid \mathsf{proj}_2(a) \mid \mathsf{inj}_1(a) \mid \mathsf{inj}_2(a) \\
\text{Computations} & m,n,p ::= & \mathsf{app}(a,b) \mid \mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n \\
 & & \mid \mathsf{val}(v) \mid \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n \mid \mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)
\end{array}$$

where variables are ranged over by f, g, x, y, z, and constants are ranged over by c. The only significant change change is that the letfun construct is now more than just syntactic sugar for binding a lambda. This allows us to bind recursive as well as non-recursive functions. Note that exceptions, references and recursive types have been removed, but the analysis and implementation extends straightforwardly to include them. Indeed, we have implemented the algorithms described in this chapter in the actualy SML.NET compiler for the full version of MIL.

We define the *size* of MIL terms $|\cdot|$ as:

$$
\begin{array}{cc}
|a| = 1 & |\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n| = |m| + |n| + 1 \\
|\mathsf{proj}_i(a)| = |\mathsf{inj}_i(a)| = 2 & |\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n| = |m| + |n| + 1 \\
|\mathsf{app}(a,b)| = |\mathsf{pair}(a,b)| = 3 & |\mathsf{val}(v)| = |v| + 1 \\
\multicolumn{2}{c}{|\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)| = |n_1| + |n_2| + 2}
\end{array}
$$

We say that a reduction is a shrinking reduction if it always reduces the size of terms. The most important reductions are given by the shrinking β-rules:

$$
\begin{array}{lll}
(\to.\beta_0) & \mathsf{letfun}\ f(x) \Leftarrow n\ \mathsf{in}\ m \longrightarrow m, & f \notin fv(m) \\
(\to.\beta_1) & \mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ C[\mathsf{app}(f,a)] \longrightarrow C[m[x := a]], & f \notin fv(C[\cdot], m, a) \\
(T.\beta_0) & \mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ m \longrightarrow m, & x \notin fv(m) \\
(T.\beta_a) & \mathsf{letval}\ x \Leftarrow a\ \mathsf{in}\ m \longrightarrow m[x := a] & \\
(\times.\beta) & \mathsf{letval}\ y \Leftarrow \mathsf{pair}(a_1,a_2)\ \mathsf{in}\ C[\mathsf{proj}_i(y)] & \\
 & \quad \longrightarrow \mathsf{letval}\ y \Leftarrow \mathsf{pair}(a_1,a_2)\ \mathsf{in}\ C[a_i] & \\
(+.\beta) & \mathsf{letval}\ y \Leftarrow \mathsf{inj}_i(a) & \\
 & \mathsf{in}\ C[\mathsf{case}\ y\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)] & \\
 & \quad \longrightarrow \mathsf{letval}\ y \Leftarrow \mathsf{inj}_i(a)\ \mathsf{in}\ C[n_i[x_i := a]] &
\end{array}
$$

We write R_β for the one-step reduction relation defined by the β-rules. The *simplify* transformation also performs commuting conversions. These ensure that bindings are explicitly sequenced, which enables further rewriting.

$$
\begin{array}{ll}
(T.CC) & \mathsf{let}\ y \Leftarrow (\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n)\ \mathsf{in}\ p \\
 & \quad \longrightarrow \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p \\
(\to.CC) & \mathsf{let}\ y \Leftarrow (\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n)\ \mathsf{in}\ p \\
 & \quad \longrightarrow \mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p \\
(+.CC) & \mathsf{let}\ y \Leftarrow (\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_1))\ \mathsf{in}\ m \\
 & \quad \longrightarrow \mathsf{letfun}\ f(y) \Leftarrow m\ \mathsf{in}\ \mathsf{case}\ a\ \ \mathsf{of}\ \ x_1 \Rightarrow \mathsf{let}\ y_1 \Leftarrow n_1\ \mathsf{in}\ \mathsf{app}(f,y_1) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \ x_2 \Rightarrow \mathsf{let}\ y_2 \Leftarrow n_2\ \mathsf{in}\ \mathsf{app}(f,y_2)
\end{array}
$$

We write R_{CC} for the reduction relation defined by the CC-rules, and R for $R_\beta \cup R_{CC}$. Unlike the β rules, the commuting conversions are not actually shrinking reductions. However, $T.CC$ and $\to.CC$ do not change the size, whilst $+.CC$ gives only a constant increase in the size.

An alternative to the $+.CC$ rule is:

$$\begin{array}{ll} (+.CC') & \mathsf{let}\ y \Leftarrow \mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)\ \mathsf{in}\ m \\ & \quad \longrightarrow\ \mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{let}\ y_1 \Leftarrow n_1\ \mathsf{in}\ m_1 \mid x_2 \Rightarrow \mathsf{let}\ y_2 \Leftarrow n_2\ \mathsf{in}\ m_2) \end{array}$$

where y_1, y_2 are fresh, $m_i = m[y{:=}y_i]$. This rule duplicates the term m and can exponentially increase the term's size. The $+.CC$ rule instead creates a single new abstraction, shared across both branches of the case, though this inhibits some further rewriting. We write R'_{CC} for the reduction relation defined by the CC-rules where $(+.CC)$ is replaced by $(+.CC')$, and R' for $R_\beta \cup R'_{CC}$.

Proposition 7.1. *R' is strongly-normalising.*

Proof. First, note that R_β is strongly-normalising as R_β-reduction strictly decreases the size of terms. We define the measures $|\cdot|_\beta$ for β-reduction:

$$\begin{array}{cc} |a|_\beta = 1 & |\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n|_\beta = |m|_\beta + |n|_\beta + 1 \\ |\mathsf{proj}_i(a)|_\beta = |\mathsf{inj}_i(a)|_\beta = 2 & |\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n|_\beta = |m|_\beta + |n|_\beta + 1 \\ |\mathsf{app}(a,b)|_\beta = |\mathsf{pair}(a,b)|_\beta = 3 & |\mathsf{val}(v)|_\beta = |v|_\beta + 1 \\ \multicolumn{2}{c}{|\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)|_\beta = max(|n_1|_\beta, |n_2|_\beta) + 2} \end{array}$$

and $|\cdot|_{CC}$ for CC-reduction:

$$\begin{array}{cc} |a|_{CC} = 1 & |\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n|_{CC} = |m|_{CC} + |n|_{CC} + 1 \\ |\mathsf{proj}_i(a)|_{CC} = |\mathsf{inj}_i(a)|_{CC} = 2 & |\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n|_{CC} = |m|_{CC}^2 + |n|_{CC} + 1 \\ |\mathsf{app}(a,b)|_{CC} = |\mathsf{pair}(a,b)|_{CC} = 3 & |\mathsf{val}(v)|_{CC} = |v|_{CC} + 1 \\ \multicolumn{2}{c}{|\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)|_{CC} = max(|n_1|_{CC}, |n_2|_{CC}) + 2} \end{array}$$

The lexicographic ordering $(|\cdot|_\beta, |\cdot|_{CC})$ is a measure for R'-reduction. Each shrinking β-reduction decreases $|\cdot|_\beta$, whilst each CC-reduction decreases $|\cdot|_{CC}$ and leaves $|\cdot|_\beta$ unchanged. □

Proposition 7.2. *R is strongly-normalising.*

The proof uses R'-reduction to simulate R-reduction. The full details are omitted, but the idea is that for any R-reduction a corresponding non-empty sequence of R'-reductions can be performed. Thus, given that all R'-reduction sequences are finite, all R-reduction sequences must also be finite. The proof is slightly complicated by the fact that no non-empty sequence of R'-reductions corresponds with the β-reduction of a function introduced by the $+.CC$ rule. A simple way of dealing with this is to count a $+.CC'$-reduction as two reductions.

Note that R-reductions are not confluent. The failure of confluence is due to the $(+.CC)$ rule. Replacing $(+.CC)$ with $(+.CC')$ does give a confluent system. Confluence can make reasoning about reductions easier, but we do not regard failure of confluence as a problem. In our case, preventing exponential growth in the size of terms is far more important.

7.2 Previous Work

Appel and Jim [AJ97] considered a calculus which is essentially a sub-calculus of our simplified MIL [1]. In our setting the reductions that their algorithms perform are equivalent to: $\rightarrow .\beta_1$-, $\times.\beta$-, $T.\beta_0$-, and a restriction of $\rightarrow .\beta_0$-reduction. Appel and Jim show that their calculus is confluent in the presence of these reductions, and other 'δ-rules' satisfying certain criteria.

The reductions rely on knowing the number of occurrences of a particular variable. The quadratic algorithms store this information in a table *Count* mapping variable names to their number of occurrences. Appel and Jim's naïve algorithm repeatedly (i) zeros the usage counts, (ii) performs a *census* pass over the whole term to update the usage counts and then (iii) traverses the term performing reductions on the basis of the information in *Count*, until there are no redexes remaining.

The improved algorithm, used in SML/NJ and SML.NET, dynamically updates the usage counts as reductions are performed. This allows more reductions to be performed on each pass, and requires a full census to be performed only once. The improved algorithm is better in practice, but both algorithms have worst-case time complexity $\Theta(n^2)$ where n is the size of the input term.

Appel and Jim's restricted version of $\rightarrow .\beta_0$-reduction is given by the two dead-function elimination rules:

[1] In fact their calculus is n-ary, like the full version of MIL.

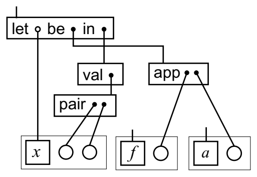

Figure 7.1: Pictorial representation of $\mathsf{let}\ x \Leftarrow \mathsf{app}(f,a)\ \mathsf{in}\ \mathsf{val}(\mathsf{pair}(x,x))$

$$\begin{array}{ll}\mathsf{letfun}\ f(x) \Leftarrow n\ \mathsf{in}\ m \longrightarrow m, & f \notin fv(m,n)\\ \mathsf{letfun}\ f(x) \Leftarrow C[\mathsf{app}(f,a)]\ \mathsf{in}\ m \longrightarrow m, & f \notin fv(C[\cdot],m)\end{array}$$

The fact that reductions are triggered only when the *total* number of occurrences of a variable is 0 or 1 explains the rather strange form of the dead-function elimination rules. In SML.NET, separate names are used for recursive and non-recursive occurrences of functions, so the unrestricted $\rightarrow .\beta_0$-rule is used.

7.3 A Graph-based Representation

Our imperative algorithm works with a mutable graph representation comprising a doubly-linked expression tree and a list of pairs of circular doubly-linked lists collecting all the recursive (respectively non-recursive) uses of each variable. Such graphs can naturally be presented pictorially as shown by the example in Fig. 7.1.

Figure 7.2 shows the β-reductions for functions in this pictorial form. We find the pictorial representation intuitively very useful, but awkward to reason with or use in presenting algorithms. Hence, like Appel and Jim, we will work with a more abstract structure comprising an expression tree and a collection of maps which capture the additional graphical structure between nodes of the tree.

The structure of expression trees is determined by the abstract syntax of simplified MIL. In order to capture mutability we use ML-style references. Each node of the expression tree is a reference cell. We call the entities which reference cells contain

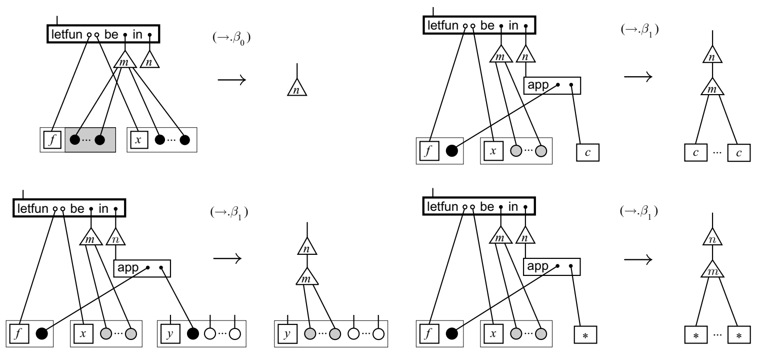

Figure 7.2: Graph reductions

objects. Given a reference cell l, we write $!l$ to denote the contents of l, and $l := u$ to denote the assignment of the object u to l.

Atoms	$!a, !b ::= r \mid * \mid c$
Values	$!v, !w ::= a \mid \mathsf{pair}(a,b) \mid \mathsf{proj}_1(a) \mid \mathsf{proj}_2(a) \mid \mathsf{inj}_1(a) \mid \mathsf{inj}_2(a)$
Computations	$!m, !n, !p ::= \mathsf{app}(a,b) \mid \mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n$
	$\mid \mathsf{val}(v) \mid \mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n \mid \mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$
	$e ::= v \mid m \qquad d ::= e \mid x \mid r$

where f, g, x, y, z range over defining occurrences, and r, s, t over uses. For the parent of the node e, we write *parent*(e). A distinguished sentinel node, *root*, marks the top of the expression tree. The object dead (omitted from the grammar) is used to indicate a *dead* node. If a node is dead then it has no parent. The *root* node is the parent of the proper expression tree and is always dead. We define *children*(e) of an expression node to be the set of nodes appearing in $!e$.

Initially both *parent* and *children* are entirely determined by the expression tree. However, in our algorithm we take advantage of the *parent* map in order to classify expression nodes as active or inactive. We ensure that the following invariant is maintained: for all expression nodes e, either

- e is active: *parent*(d) = e, for all $d \in$ *children*(e);

- e is inactive: $!(\mathit{parent}(d)) = \textsf{dead}$ for all $d \in \mathit{children}(e)$; or
- e is dead: $!e = \textsf{dead}$.

We define *splicing* as the operation which takes one subtree m and substitutes it in place of another subtree n. The subtree m is removed from the expression tree and then reintroduced in place of n. The parent map is adjusted accordingly for the children of m. We define *splicing a copy* as the corresponding operation which leaves the original copy of m in place. The operation $\lceil q \rceil$ returns a new node containing q, with parent *root*. When embedded in an enclosing node $e[\lceil q \rceil]$, the parent of $\lceil q \rceil$ is e. In patterns, $\lceil \cdot \rceil$ matches against the contents of a node.

The *def-use* maps abstract the structures used for representing occurrences:

- $\mathit{def}(r)$ gives the defining occurrence of the use r.
- $\mathit{non\text{-}rec\text{-}uses}(x)$ is the set of non-recursive uses of the defining occurrence x.
- $\mathit{rec\text{-}uses}(x)$ is the set of recursive uses of the defining occurrence x.

In the real implementation occurrences are held in a pair of doubly-linked circular lists, such that each pair of lists intersects at a defining occurrence. We find it convenient to overload the maps to be defined over all occurrences and also define some additional maps:

$$\mathit{non\text{-}rec\text{-}uses}(r) = \mathit{non\text{-}rec\text{-}uses}(\mathit{def}(r))$$
$$\mathit{rec\text{-}uses}(r) = \mathit{rec\text{-}uses}(\mathit{def}(r)) \qquad \mathit{def}(x) = x$$
$$\mathit{occurrences}(r) = \mathit{uses}(r) \cup \{\mathit{def}(r)\} \quad \mathit{uses}(r) = \mathit{non\text{-}rec\text{-}uses}(r) \cup \mathit{rec\text{-}uses}(r)$$

None of these additional definitions affects the implementation.

The graph structure allows constant time movement up and down the expression tree in the normal way, but also allows constant time non-local movement via the occurrence lists. For example, consider the dead-function eliminations:

$$\begin{array}{ll} & \textsf{letfun } f(x) \Leftarrow m \textsf{ in } C[\textsf{letfun } g(y) \Leftarrow \textsf{app}(f,y) \textsf{ in } n] \\ \longrightarrow_{(\to.\beta_0)} & \textsf{letfun } f(x) \Leftarrow m \textsf{ in } C[n] \quad \longrightarrow_{(\to.\beta_0)} \quad C[n] \end{array}$$

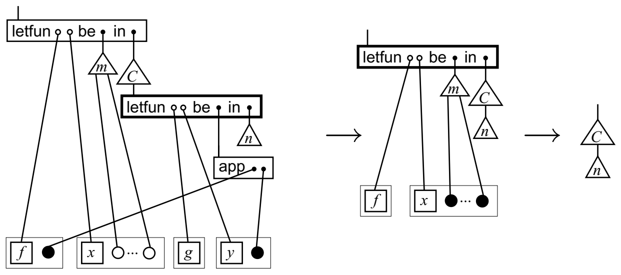

Figure 7.3: Triggering non-local reductions

where $f, g \notin fv(C, n)$, illustrated in Fig. 7.3. After one reduction, g is dead, so its definition can be deleted, removing the use of f. But this use was connected to its defining occurrence, and f is now dead. The defining occurrence is connected to its parent, so the new dead-function redex can be reduced.

7.4 A One-pass Algorithm

In contrast to Appel and Jim's imperative algorithm, the algorithm we have implemented operates in one-pass. Essentially, the one-pass algorithm performs a depth-first traversal of the expression tree, reducing redexes on the way back up the tree. Of course, these reductions may trigger further reductions elsewhere in the tree. By carefully deactivating parts of the tree, we are able to control the reduction order and limit the testing required for new redexes. Here is an outline of our one-pass imperative algorithm:

```
$contract(e)$ =   $reduceCCs(e)$
                  $deactivate(e)$
                  apply $contract$ to children of $e$
                  $reactivate(e)$
                  $reduce(e)$
  $reduce(e)$ =   if $e$ is a redex then
                          reduce $e$ in place
                          perform further reductions triggered by reducing $e$
```

The operation *reduceCCs*(*e*) performs commuting conversions on the way down the tree. The order of commuting conversions can have a significant effect on code quality, a poor choice leading to many jumps to jumps. We have found that the approach of doing them on the way down works well in practice (although the contract algorithm would still be valid without the call to *reduceCCs*).

```
$reduceCCs(e)$ = case $!e$ of
    (let $y \Leftarrow e'$ in $p$) $\Rightarrow$
        if $reduceCC(e,t,e',p) \neq \emptyset$ then $reduceCCs(e)$ else skip
    (_) $\Rightarrow$ skip
$reduceCC(e,y,e',p)$ = case $e'$ of
    (letfun $f(x) \Leftarrow m$ in $n$) $\Rightarrow$
      splice $\lceil$let $y \Leftarrow n$ in $p\rceil$ in place of $e'$
      splice $\lceil$letfun $f(x) \Leftarrow m$ in $e'\rceil$ in place of $e$
      return $\{e'\}$
    (let $x \Leftarrow m$ in $n$) $\Rightarrow$
      splice $\lceil$let $y \Leftarrow n$ in $p\rceil$ in place of $e'$
      splice $\lceil$let $x \Leftarrow m$ in $e'\rceil$ in place of $e$
      return $\{e'\}$
    (case $a$ of $(x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)$) $\Rightarrow$
      splice $\lceil$let $y_1 \Leftarrow n_1$ in $\lceil$app$(f,y_1)\rceil\rceil$ in place of $n_1$
      splice $\lceil$let $y_2 \Leftarrow n_2$ in $\lceil$app$(f,y_2)\rceil\rceil$ in place of $n_2$
      splice $\lceil$letfun $f(y) \Leftarrow p$ in $\lceil$case $a$ of $(x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)\rceil\rceil$
      in place of $e$
      return $\{n_1, n_2\}$
    (_) $\Rightarrow$ return $\emptyset$
```

Note that commuting conversions can also be triggered by other reductions. The return value for *reduceCC* will be used in the definition of *reduce* in order to catch reductions which are triggered by applying commuting conversions.

deactivate(e) deactivates e: *parent*(d) is set to dead for every $d \in$ *children*(e).

reactivate(e) reactivates e: *parent*(d) is set to e for every $d \in$ *children*(e).

Deactivating nodes on the way down prevents reductions from being triggered above the current node in the tree. On the way back up the nodes are reactivated, allowing any new redexes to be reduced. Because subterms are known to be normalised, fewer tests are needed for new redexes. Consider, for example:

$$\text{let } y \Leftarrow (\text{let } x \Leftarrow m \text{ in } n) \text{ in } p \longrightarrow_{T.CC} \text{let } x \Leftarrow m \text{ in let } y \Leftarrow n \text{ in } p$$

Because we know that $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n$ is in normal form, m cannot be of the form $\mathsf{let}(\dots), \mathsf{letfun}(\dots), \mathsf{case}(\dots)$ or $\mathsf{val}(\dots)$. Hence, it is not necessary to check whether $\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p$ is a redex. (Of course, $\mathsf{let}\ y \Leftarrow n\ \mathsf{in}\ p$ may still be a redex, and indeed exposing such redexes is one of the main purposes of performing CC-reduction.)

Rather than placing them in a redex set, as in Appel and Jim's imperative algorithm, *reduce*(e) reduces any new redexes created inside e (but none that are created above e in the expression tree). If *reduce*(e) is invoked on an expression node which is not a redex, then no action is performed. The *reduce* function also returns a boolean to indicate whether a reduction took place. As we shall see, this is necessary in order to detect the triggering of new reductions. We now expand the definition of *reduce*.

```
$reduce(e)$ = case $!e$ of
    $(\mathsf{letfun}\ f(x) \Leftarrow m\ \mathsf{in}\ n) \Rightarrow$
        if $non\text{-}rec\text{-}uses(f) = \emptyset$ then
            splice $n$ in place of $e$
            $reduceOccs(cleanExp(m))$
            return true
        else if $rec\text{-}uses(f) = \emptyset$ and $non\text{-}rec\text{-}uses(f) = \{f'\}$ then
            let $focus = parent(parent(f'))$
            case $!focus$ of
            $(\mathsf{app}(f', a) \Rightarrow$
                splice $n$ in place of $e$
                splice $m$ in place of $focus$
                let $(occs, redexes) = substAtom(x, a)$
                $reduceOccs(occs \cup cleanExp(a))$
                $reduceRedexes(redexes)$
                return true
            $(\_) \Rightarrow$ return false
        else return false
    $(\mathsf{let}\ x \Leftarrow \lceil \mathsf{val}(v) \rceil\ \mathsf{in}\ n) \Rightarrow$
        if $uses(x) = \emptyset$ then
            splice $n$ in place of $e$
            $reduceOccs(cleanExp(parent(v)))$
            return true
        else if $v$ is an atom $a$ then
            splice $n$ in place of $e$
            let $(occs, redexes) = substAtom(x, a)$
            $reduceOccs(occs \cup cleanExp(parent(a)))$
            $reduceRedexes(redexes)$
            return true
        else case $!v$ of
        $(\mathsf{pair}(a,b)) \Rightarrow$
            if $e$ is fresh then
                let $redexes = reduceProjections(e,x,a,b,uses(x))$
                if $redexes = \emptyset$ then return false
                else
                    $reduceRedexes(redexes)$
                    $reduce(e)$
                    return true
            else return false
        $(\mathsf{inj}_i(a)) \Rightarrow$
            if $e$ is fresh then
                let $(occs,redexes) = reduceCases(e,x,i,a,uses(x))$
                if $redexes = \emptyset$ then return false
                else
                    $reduceOccs(occs)$
                    $reduceRedexes(redexes)$
                    $reduce(e)$
                    return true
            else return false
        $(\_) \Rightarrow$ return false
    $(\mathsf{let}\, y \Leftarrow e' \;\mathsf{in}\; p) \Rightarrow$
        let $redexes = reduceCC(e,y,e',p)$
        for $e'' \in redexes$ do $reduce(e'')$
        return true
    $(\_) \Rightarrow$ return false
```

The first case covers β-reductions on functions, with two sub-cases:

- $(\rightarrow .\beta_0)$ If the function is dead, its definition is removed, the continuation spliced in place of e, and any uses within the dead body deleted, possibly triggering new reductions.

- $(\rightarrow .\beta_1)$ If the function has one occurrence, which is non-recursive, it is inlined. The continuation of e is spliced in place of e, the function body is inlined with the argument substituted for the parameter, and the argument deleted. Substitution may trigger further reductions.

The second case covers β-reductions on computations as well as some instances of β-reduction on products and sums. It is divided into four sub-cases.

- ($T.\beta_0$) If a value is dead, then its definition can be removed. The continuation is spliced in place of e. Then the uses inside the dead function body are deleted, possibly triggering new reductions.

- ($T.\beta_a$) If a value is atomic, then it can be inlined. First the continuation of e is spliced in place of e. Then the atom is substituted for the bound variable. Finally the atom is deleted.

- ($\times.\beta$) If a pair is bound to a variable x, and this is the first time e has been visited, then any projections of x are reduced. If this is the first time e has been visited, then we say that e is *fresh*. In practice freshness is indicated by setting a global flag. For efficiency, new projections will subsequently be reduced as and when they are created.

- ($+.\beta$) This follows exactly the same pattern as $\times.\beta$-reduction. The only difference is that the reduction itself is more complex, so can trigger new reductions in different ways.

The third case deals with commuting conversions.

The algorithm ensures that the current reduction is complete before any new reductions are triggered. Potential new redexes created by the current reduction are encoded and executed after the current reduction has completed.
reduceUp(e) reduces above e as far as possible:

```
$reduceUp(e)$ = if $reduce(e)$ then $reduceUp(parent(e))$ else skip
```

reduceRedexes reduces a set of expression redexes, whilst *reduceOccs* reduces a set of occurrence redexes:

```
$reduceRedexes(redexes)$ = for each $e \in redexes$ do $reduceUp(e)$
$reduceOccs(xs)$ = for each $r \in xs$ do
    if $isSmall(r)$ then $reduceUp(parent(def(r)))$ else skip
$isSmall(r) = r \notin rec\text{-}uses(r)$ and $|non\text{-}rec\text{-}uses(r)| \leq 1$
```

cleanExp(e) removes all occurrences and subexpressions inside e and returns a set of occurrence redexes.

```
$cleanExp(e)$ = case $!e$ of
    (dead) $\Rightarrow$ return $\emptyset$
    $(c) \Rightarrow$
        $e :=$ dead
        return $\emptyset$
    $(*) \Rightarrow$
        $e :=$ dead
        return $\emptyset$
    $(r) \Rightarrow$
        $e :=$ dead
        return $deleteUse(r)$
    (letfun $f(x) \Leftarrow m$ in $n$) $\Rightarrow$
        $e, f, x :=$ dead
        return $cleanExp(m) \cup cleanExp(n)$
    (app$(a, b)$) $\Rightarrow$
        $e :=$ dead
        return $cleanExp(a) \cup cleanExp(b)$
    ... (similar for the other MIL constructs)
```

Remark Marking nodes as dead ensures that unnecessary work is not done on dead redexes. A crucial difference between the imperative algorithms and the improved quadratic one is that reduction in the former immediately detects new redexes, whereas the improved quadratic algorithm only detects new (non-local) redexes on a subsequent traversal.

$deleteUse(r)$ removes r and returns a set of 0 or 1 occurrence redexes:

```
$deleteUse(r)$ =
    if $r$ is already dead then return $\emptyset$
    let $s = nextOcc(r)$
    $uses(s) := uses(s) - \{r\}$
    return $\{s\}$

$nextOcc(r)$ =
    let $x = def(r)$
    if $r$ is non-recursive then return $s \in (non\text{-}rec\text{-}uses(x) \cup \{x\}) - \{r\}$
    else if $r$ is recursive then return $s \in (rec\text{-}uses(x) \cup \{x\}) - \{r\}$
```

$reduceProjections(e, x, a_1, a_2, xs)$ reduces projections indexed by xs. e is an expression node of the form letval $x \Leftarrow$ pair(a_1, a_2) in m, and xs is a subset of the uses of x.

```
$reduceProjections(e, x, a_1, a_2, xs) =$
    let $redexes := \emptyset$
    for each $s \in xs$ do
        let $focus = parent(parent(s))$
        case $!focus$ of
        $(\mathsf{proj}_i(s)) \Rightarrow$
            splice a copy of $a_i$ in place of $focus$
            $redexes := redexes \cup \{parent(focus)\}$
        $(\_) \Rightarrow$ skip
    return $redexes$
```

All the projections in which a member of xs participates are reduced, and a set of expression redexes is constructed. Each projection can trigger the creation of a new $T.\beta_a$-redex. For instance, consider:

$$\begin{array}{rl} & \mathsf{letval}\ x \Leftarrow \mathsf{pair}(a,b)\ \mathsf{in}\ \mathsf{letval}\ y \Leftarrow \mathsf{proj}_1(x)\ \mathsf{in}\ m \\ \longrightarrow_{\times.\beta} & \mathsf{letval}\ x \Leftarrow \mathsf{pair}(a,b)\ \mathsf{in}\ \mathsf{letval}\ y \Leftarrow a\ \mathsf{in}\ m \\ \longrightarrow_{T.\beta_a} & \mathsf{letval}\ x \Leftarrow \mathsf{pair}(a,b)\ \mathsf{in}\ m \end{array}$$

$reduceCases(e, x, i, a, xs)$ reduces case-splits indexed by xs. e is an expression node of the form $\mathsf{letval}\ x \Leftarrow \mathsf{inj}_i(a)\ \mathsf{in}\ m$, and xs is a subset of the uses of x.

```
$reduceCases(e, x, i, a, xs) =$
    let $occs := \emptyset$
    let $redexes := \emptyset$
    for each $s \in xs$ do
        let $focus = parent(parent(s))$
        case $!focus$ of
        $(\mathsf{case}\ s\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2)) \Rightarrow$
            $occs := occs \cup cleanExp(n_{3-i})$
            $deleteUse(s)$
            splice $n_i$ in place of $focus$
            let $(occs', redexes') = substAtom(x_i, a)$
            $occs := occs \cup occs'$
            $redexes := redexes \cup redexes' \cup \{parent(focus)\}$
            $x_1, x_2 := \mathsf{dead}$
        $(\_) \Rightarrow$ skip
    return $(occs, redexes)$
```

The structure of *reduceCases* is similar to that of *reduceProjections*. However, it is slightly more complex because a single $+.\beta$-reduction inlines multiple atoms, splices one branch of a case and discards the other. Discarding the branch which is not taken gives a set of occurrence redexes as well as the expression redexes.

substAtom(x,a) substitutes the atom a for all the uses of the defining occurrence x. It returns a pair of a set of occurrence redexes and a set of expression redexes.

```
$substAtom(x,a)$ = case $(!a)$ of
    $(r) \Rightarrow substUse(x,r)$
    $(\_) \Rightarrow$
        for each $r \in uses(x)$ do
            splice a copy of $a$ in place of $r$
            $x :=$ dead
        return $(\emptyset,\emptyset)$
```

This is straightforward for non-variable atoms, as it cannot generate new redexes. In contrast, substituting a variable can trigger $\times.\beta$- and $+.\beta$-reductions.

substUse(x,r) substitutes r for all the uses of the defining occurrence x.

```
$substUse(x,r) =$
    let $xs = uses(x)$
    if $r \in rec\text{-}uses(r)$ then
        $rec\text{-}uses(r) := rec\text{-}uses(r) \cup xs$
    else if $r \in non\text{-}rec\text{-}uses(r)$
        $non\text{-}rec\text{-}uses(r) := non\text{-}rec\text{-}uses(r) \cup xs$
    $x :=$ dead
    let $e = parent(def(r))$
    case $!e$ of
    (letval $y \Leftarrow \lceil$pair$(a_1,a_2)\rceil$ in $m$) $\Rightarrow$
        for each $s \in xs$ do $def(s) := def(r)$
        let $redexes = reduceProjections(e,y,a_1,a_2,xs)$
        return $(\emptyset, redexes)$
    (letval $y \Leftarrow \lceil$inj$_i(a_i)\rceil$ in $m$) $\Rightarrow$
        for each $s \in xs$ do $def(s) := def(r)$
        let $(occs, redexes) = reduceCases(e,y,i,a_i,xs)$
        return $(occs, redexes)$
    $(\_) \Rightarrow$ return $(\emptyset,\emptyset)$
```

Substitution is implemented by merging two sets together. Concretely, this amounts to the constant-time operation of inserting one doubly-linked circular list inside another. In addition, if x is bound to a pair, then projections are reduced, or if x is bound to an injection, then case-splits are reduced.

7.5 Analysis

There are two obvious operations mapping terms from the functional to the imperative representations, which we call *mutify* and *demutify*, respectively. We have a semi-formal argument for the following:

Proposition 7.3. *Let e be a term and $e' = (\mathit{demutify} \circ \mathit{contract} \circ \mathit{mutify})(e)$. Then e' is a normal form for e.*

The argument uses the invariants of §7.3, plus the invariant that the children of the current node are in normal form. When new redexes are created, this invariant is modified such that subterms may contain redexes, but only those stored in appropriate expression redex sets or occurrence redex sets. It is reasonably straightforward to verify that the operations which update the graph structure do in fact correspond to MIL reductions. When *contract* terminates, all the redex sets are empty and the term is in normal form.

7.5.1 Complexity without Commuting Conversions

Although our approach of performing CCs on the way down the tree works well in practice, the worst case time complexity is still quadratic in the size of the term. We define a version of our algorithm $\mathit{contract}_\beta$ which does not perform commuting conversions. This is obtained simply by removing the call to *reduceCCs* from *contract*, and the test for commuting conversions from *reduce*.

Proposition 7.4. *$\mathit{contract}_\beta(e)$ is linear in the size of e.*

The argument is very similar to that of Appel and Jim [AJ97] for their imperative algorithm. Most of the operations take constant time and shrink the size of the term; the exception is substitution. In the case where a non-variable is substituted for a variable x, the operation is linear in the number of uses of x. But it is only possible to subsitute a non-variable for a variable once, therefore the total time spent substituting atoms is linear. In the case where a variable y is substituted for a variable x, the operation is constant, providing y is not bound to a pair or an injection. If y is bound to a pair or an injection, then the operation is linear in the number of uses of x. Again, once bound to a pair or an injection, a variable cannot be rebound, so the time remains linear.

Crucially, this argument relies on the fact that back pointers from uses back to defining occurrences are maintained only for pairs and injections. In our SML.NET implementation we found that maintaining back pointers from *all* uses back to defining occurrences does not incur any significant cost in practice. Even when bootstrapping the compiler ($\sim$ 80,000 lines of code) there was no discernible difference in compile time. Maintaining back pointers also allows us to perform various other rewrites including η-reductions. In the presence of all backpointers, optimising the union operation to always add the smaller list to the larger one guarantees $O(n\ log\ n)$ behaviour. Using an efficient *union-find* [CLR90, GI91] algorithm would restore essentially linear complexity.

7.5.2 Complexity with Commuting Conversions

Naïvely reducing commuting conversions can give quadratic behaviour. For instance, consider the following (innermost first) reductions:

$$\begin{array}{l}
\mathsf{let}\ x_k \Leftarrow (\mathsf{let}\ x_{k-1} \Leftarrow \ldots \mathsf{let}\ x_1 \Leftarrow m_1\ \mathsf{in}\ m_2\ \mathsf{in}\ \ldots m_k)\ \mathsf{in}\ n \\
\longrightarrow^* \ (S(k-1)\ T.CC\text{-reductions}) \\
\mathsf{let}\ x_k \Leftarrow (\mathsf{let}\ x_1 \Leftarrow m_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_{k-1} \Leftarrow m_{k-1}\ \mathsf{in}\ m_k)\ \mathsf{in}\ n \\
\longrightarrow^* \ (k-1\ T.CC\text{-reductions}) \\
\mathsf{let}\ x_1 \Leftarrow m_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_k \Leftarrow m_k\ \mathsf{in}\ n
\end{array}$$

The total number of reductions is given by the recurrence: $S(1) = 0, S(k) = S(k-1) + k - 1$. This has solution $S(k) = k(k-1)/2$. Assuming each of the m_is and n have constant size, then k is linear in the size of the term. Hence the number of reductions is quadratic in the size of the term. If the *contract* function directly performed these reductions, then it would also be quadratic.

Another problem is that $+.CC$-reductions can introduce ‘useless functions’:

$$
\begin{array}{ll}
 & \mathsf{let}\ z \Leftarrow (\mathsf{let}\ y \Leftarrow (\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))\ \mathsf{in}\ m)\ \mathsf{in}\ p \\
\longrightarrow^* & \mathsf{letfun}\ f(y) \Leftarrow m \\
 & \mathsf{in}\ \ \mathsf{let}\ z \Leftarrow \mathsf{case}\ a\ \ \mathsf{of}\ \ x_1 \Rightarrow \mathsf{let}\ y_1 \Leftarrow n_1\ \mathsf{in}\ \mathsf{app}(f,y_1) \\
 & \qquad\qquad\qquad\quad \mid\ \ x_2 \Rightarrow \mathsf{let}\ y_2 \Leftarrow n_2\ \mathsf{in}\ \mathsf{app}(f,y_2) \\
 & \qquad \mathsf{in}\ p \\
\longrightarrow^* & \mathsf{letfun}\ f(y) \Leftarrow m \\
 & \mathsf{in}\ \ \mathsf{letfun}\ g(z) \Leftarrow p \\
 & \qquad \mathsf{in}\ \mathsf{case}\ a\ \ \mathsf{of}\ \ x_1 \Rightarrow \mathsf{let}\ y_1 \Leftarrow n_1\ \mathsf{in}\ \mathsf{let}\ z_1 \Leftarrow \mathsf{app}(f,y_1)\ \mathsf{in}\ \mathsf{app}(g,z_1) \\
 & \qquad\qquad\qquad\quad \mid\ \ x_2 \Rightarrow \mathsf{let}\ y_2 \Leftarrow n_2\ \mathsf{in}\ \mathsf{let}\ z_2 \Leftarrow \mathsf{app}(f,y_2)\ \mathsf{in}\ \mathsf{app}(g,z_2)
\end{array}
$$

The function g is useless in the sense that it is always applied to the result of applying f to an argument. One might hope that g be composed with f. If we change the reduction order, such that the commuting conversions are performed outermost first, then it is:

$$
\begin{array}{ll}
 & \mathsf{let}\ z \Leftarrow (\mathsf{let}\ y \Leftarrow (\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))\ \mathsf{in}\ m)\ \mathsf{in}\ p \\
\longrightarrow^* & \mathsf{let}\ y \Leftarrow (\mathsf{case}\ a\ \mathsf{of}\ (x_1 \Rightarrow n_1 \mid x_2 \Rightarrow n_2))\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow m\ \mathsf{in}\ n \\
\longrightarrow^* & \mathsf{letfun}\ f(y) \Leftarrow \mathsf{let}\ z \Leftarrow m\ \mathsf{in}\ p \\
 & \mathsf{in}\ \mathsf{case}\ a\ \ \mathsf{of}\ \ x_1 \Rightarrow \mathsf{let}\ y_1 \Leftarrow n_1\ \mathsf{in}\ \mathsf{app}(f,y_1) \\
 & \qquad\qquad\quad \mid\ \ x_2 \Rightarrow \mathsf{let}\ y_2 \Leftarrow n_2\ \mathsf{in}\ \mathsf{app}(f,y_2)
\end{array}
$$

Fortunately, given the limited ways in which commuting conversions can trigger other reductions, the full imperative algorithm can get away with performing commuting conversions outermost first, with an initial call to *reduceCCs*(e) before recursively contracting e's children. The operation *reduceCCs*(e) repeatedly checks e to see if it is a CC-redex. If it is, then it performs the commuting conversion, and iterates. If not, then it returns.

The previous example of quadratic behaviour due to commuting conversions becomes linear with this reduction strategy. However, quadratic behaviour can still arise through inlining functions that trigger further commuting conversions:

$$
\begin{array}{lrcl}
\mathsf{letfun} & f_k(x_k) & \Leftarrow & \mathsf{let}\ y_k \Leftarrow \mathsf{app}(g,x_k)\ \mathsf{in}\ \mathsf{app}(g,y_k) \\
 & f_{k-1}(x_{k-1}) & \Leftarrow & \mathsf{let}\ y_{k-1} \Leftarrow \mathsf{app}(f_k,x_{k-1})\ \mathsf{in}\ \mathsf{app}(g,y_{k-1}) \\
 & \vdots & & \\
 & f_1(x_1) & \Leftarrow & \mathsf{let}\ y_1 \Leftarrow \mathsf{app}(f_2,x_1)\ \mathsf{in}\ \mathsf{app}(g,y_1) \\
\mathsf{in} & \mathsf{app}(f_1,a) & &
\end{array}
$$

contract takes quadratic time to reduce this term. In order to get a linear number of reductions, one would have to inline all the functions first, before performing any commuting conversions.

It is unfortunate that CCs and inlining conspire to produce quadratic complexity. Sabry and Wadler's study of CPS translations offers an interesting insight [SW97]. In their variant of Moggi's computational lambda calculus λ_{c**}, terms are in CC-normal form by definition, and β-reduction of an application is combined with CC-normalisation of its enclosing let-expression: adopting this more refined notion of re-dex may allow us to achieve linear complexity.

7.5.3 Shrinking reductions as normalisation by evaluation

We can view our imperative algorithm as an instance of normalisation by evaluation. If, as in Proposition 7.3, we state the problem as performing normalisation on the functional representation, then we obtain the normalisation function *norm*:

$$\mathit{norm} = \mathit{demutify} \circ \mathit{contract} \circ \mathit{mutify}$$

It turns out that *contract* and *mutify* can be quite naturally composed. Then we can define:

$$\begin{aligned}
[\![\cdot]\!] &= \mathit{contract} \circ \mathit{mutify} \\
\downarrow &= \mathit{demutify}
\end{aligned}$$

to give the usual characterisation of normalisation by evaluation. The object language is the functional representation and the residualising semantics is given by the normalised imperative representation. This is not entirely satisfactory as an instance of normalisation by evaluation, as the semantics does not seem very natural — it is rather close to being a term model. However, it seems unlikely that there is a more natural denotational semantics for shrinking reductions. The process of normalisation in such a system is inherently rather syntactic in nature.

Table 7.1: Total compile time

(seconds)	SML/NJ		MLton	
	simplify	*mcd*	*simplify*	*mcd*
`sort`	2.11	3.47	0.46	0.52
`xq`	13.13	14.36	2.46	1.76
`mllex`	11.64	15.97	2.39	2.03
`raytrace`	18.14	23.95	4.30	3.03
`mlyacc`	57.25	43.77	10.05	6.04
`hamlet`	218.71	155.82	43.70	26.22
`bootstrap`	1311.01	1189.60	289.24	221.18

7.6 Performance

We have extended our one-pass imperative algorithm *contract* to the whole of MIL and compared its performance with the current implementation of *simplify*. Replacing *simplify* with *contract* is not entirely straightforward, as all the other phases in the pipeline are written to work on a straightforward immutable tree datatype for terms, which is incompatible with the representation used in *contract*. We therefore make use of *mutify* and *demutify* to change representation before and after *contract*. Since both *mutify* and *demutify* completely rebuild the term, they are very expensive – calling *mutify* and *demutify* generally takes longer than *contract* itself. Ideally, of course, all the phases would use the same representation. However, using two representations allowed us to compare the running times of *simplify* and *contract* on real programs.

We use the benchmark programs of Chapter 6. Table 7.1 compares the total compile times of the benchmark programs for the existing compiler, using *simplify*, and for the modified one, using *demutify* ∘ *contract* ∘ *mutify*. Each benchmark was run under two different versions of SML.NET. One was compiled under SML/NJ and the other under MLton. Benchmarks were run on a 1.4Ghz AMD Athlon PC equipped with 512MB of RAM and Microsoft Windows XP SP1.

On small benchmarks, the current compiler is faster. But for medium and large benchmarks, we were surprised to discover that *demutify* ∘ *contract* ∘ *mutify* is faster than *simplify*, even though much of the time is spent in useless representation changes.

Table 7.2 and Table 7.3 give the total time spent performing shrinking-reductions

Table 7.2: Shrinking reduction time under SML/NJ

(seconds)	Total		Breakdown of *mcd*		
	simplify	*mcd*	*contract*	*mutify*	*demutify*
`sort`	1.00	2.00	0.70	0.87	0.43
`xq`	5.86	5.98	3.61	1.90	0.47
`mllex`	6.09	7.49	3.16	3.31	1.02
`raytrace`	9.32	11.76	5.44	5.16	1.17
`mlyacc`	33.16	19.96	8.60	9.42	1.94
`hamlet`	84.49	56.36	21.53	26.24	8.59
`bootstrap`	439.16	282.38	100.11	129.60	52.67

Table 7.3: Shrinking reduction time under MLton

(seconds)	Total		Breakdown of *mcd*		
	simplify	*mcd*	*contract*	*mutify*	*demutify*
`sort`	0.22	0.11	0.07	0.02	0.02
`xq`	1.46	0.54	0.13	0.35	0.06
`mllex`	1.21	0.57	0.23	0.27	0.07
`raytrace`	2.13	0.65	0.19	0.37	0.09
`mlyacc`	5.63	1.26	0.37	0.68	0.21
`hamlet`	23.27	5.54	2.77	1.85	0.92
`bootstrap`	107.17	36.60	18.41	11.82	6.38

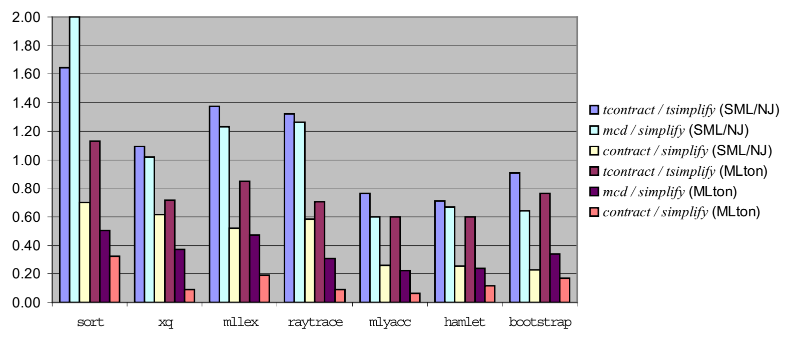

Figure 7.4: Comparing *contract* with *simplify*

in the current compiler (*simplify*), in the modified compiler (*mcd*), and a breakdown of *mcd* into the total time spent in *mutify*, *contract* and *demutify*. Figure 7.4 gives a graphical comparison. *tcontract* is the total compile time using the modified compiler, and *tsimplify* is the total compile time using the existing compiler. Under SML/NJ, a decrease of nearly 30% in the total compile time is seen in some cases. Under MLton, there is a decrease of up to 40% in total compile time. This is a significant improvement, given that in the existing compiler only around 50% of compile time is spent performing shrinking reductions. Comparing the actual shrinking reduction time, *contract* is up to four times faster than *simplify* under SML/NJ, and up to 15 times faster under MLton. The level of improvement under MLton is striking. Our results suggest that MLton is considerably better than SML/NJ at compiling ML code which makes heavy use of references.

As an exercise, one of the other transformations *deunit*, which removes redundant units was translated to use the new representation. The *contract* function is called before and after *deunit*, so this enabled us to eliminate one call to *demutify* and one call to *mutify*. This translation was easy to do and did not change the performance of *deunit*. We believe that it should be reasonably straightforward, if somewhat tedious, to translate the rest of the transformations to work directly with the mutable representation.

7.7 Summary

We have implemented and extended Appel and Jim's imperative algorithm for shrinking reductions and shown that it can yield significant reductions in compile times relative to the algorithm currently used in SML/NJ and SML.NET. The improvements are such that, for large programs, it is even worth completely changing representations before and after the *simplify* phase, but this is clearly suboptimal. The results of this experiment indicate that it would be worth the effort of rewriting some of the other phases of the compiler to use the graph-based representation.

Chapter 8

Conclusion

8.1 Summary

We have successfully applied normalisation by evaluation for normalising non-trivial terms in a non-trivial language MIL. The benchmarks confirm that normalisation by evaluation is faster than naïve normalisation algorithms, and is competitive with carefully optimised algorithms.

We have implemented a spectrum of normalisation algorithms for MIL ranging from a simple recursion over the structure of terms to normalisation by evaluation. These are related by straightforward program transformations. New features can be added to one algorithm and propagated to others by taking advantage of program transformations.

The computational metalanguage is at the core of MIL. We have described a general technique based on Girard-Tait reducibility for proving strong normalisation for the computational metalanguage and related calculi. Introducing frame stacks gives a uniform and effective way to handle normalisation with commuting conversions, as from computation types, sum types, and similar.

We have explored the space of normalisation by evaluation algorithms for a range of λ-calculus variants. In doing so we have exposed normalisation by evaluation as a practical implementation tool for normalisation and, in particular, compiler optimisation. We suggest that even if an implementation does not directly use normalisation by evaluation, it can be a useful tool in the design and analysis of normalisation algorithms. Normalisation by evaluation provides a direct connection between normalisation and denotational semantics.

Following a different path, we have implemented a much improved shrinking reductions algorithm in SML.NET. This significantly reduces compilation time, and takes advantage of a graph-based representation for terms. As far as we are aware this is the first implementation of a graph-based shrinking reductions algorithm in a production compiler.

8.2 Conclusions

Our main conclusions are as follows:

- Normalisation by evaluation is competitive with optimised rewriting-based normalisation algorithms, and is a practical tool for performing normalisation.

- Even though the worst case time-complexity of β-decidability, and hence normalisation, is given by a non-elementary function, unrestricted β-reduction is surprisingly tractable on medium sized examples such as `hamlet`.

- Normalisation by evaluation is not necessarily ideal for use in a compiler, because it is best suited to performing unrestricted β-reduction, whereas compilers usually restrict β-reduction to some degree.

- Despite some doubts expressed in the literature, either a single state cell, or a non-deterministic accumulation monad can be used to implement normalisation by evaluation for sums, without the need for continuations or first-class control operators.

- The graph-based shrinking reductions algorithm is both feasible and desirable for implementation in a full compiler. It offers a significant speed-up and a more direct representation than census-based algorithms.

8.3 General observations

We now make some more general observations:

- Derivation by program transformation is a useful technique. Recent work by Danvy and his students has promoted the idea of relating operational and denotational worlds by way of a series of elementary program transformations such as defunctionalisation and CPS transformations. Our use of program transformations, in order to derive the spectrum of normalisation algorithms in Chapter 6, follows a similar pattern. We suspect that this approach might offer the most perspicuous explanation of normalisation by evaluation to those unfamiliar with the area.

- Delimited continuations are useful. It seems that many of our normalisation by evaluation programs are most naturally expressed using the shift and reset control operators — particularly when sums are involved. We have shown that it is also possible to use state instead, though then the code becomes harder to understand and harder to analyse. The number of existing applications which use delimited continuations is somewhat limited, but they seem to fit well with normalisation by evaluation (and TDPE). Balat et al. [BCF04] even make use of a generalisation of shift and reset in the context of TDPE for sums.

- Typically the MLton compiler produces significantly faster code than SML/NJ. There is likely to be a number of factors involved. The most obvious difference between the compilers is that MLton is a whole-program compiler, whilst SML/NJ is a separate compiler. We believe that this difference is an important factor. It is interesting to note that the *contract* algorithm performs particularly well when compiled under MLton — often an order of magnitude faster than when compiled under SML/NJ. This suggests that MLton is better than SML/NJ at optimising code which uses ML references.

- Sometimes imperative techniques are preferable to declarative approaches. For instance, the use of mutable graph-based data structures allowed us to obtain a significant speed-up in the SML.NET compiler. We believe that graph-based data structures can also be used to open up new optimisation opportunities in SML.NET.

8.4 Future work

The three main strands of this thesis: $\top\top$-lifting, normalisation by evaluation and shrinking reductions; offer many avenues for future research. Here we outline a few.

8.4.1 Strong normalisation, confluence and sums

Pitts [Pit00] used frame stacks and $\top\top$-closure for reasoning about termination in a polymorphic language. It would be interesting to investigate the feasibility of extending our strong normalisation proofs to include System F style polymorphism.

As has been illustrated throughout this thesis, the addition of sums to λ-calculi makes their analysis considerably more involved. Altenkirch et al. [ADHS01] and Balat et al. [BCF04] have defined equational normal forms for λ^{+1}. However, the usual presentation of simply-typed λ-calculus with sums, as a reduction calculus, is not confluent. We conjecture that by defining appropriate reduction rules in addition to the usual ones, it is possible to obtain a strongly normalising and confluent reduction calculus which agrees with the equational calculus. It would be interesting to try to adapt frame stack reducibility to this setting in order to prove strong normalisation. Confluence would then follow from correctness of normalisation by evaluation.

In the equational setting it is possible to define alternative normal forms which are slightly smaller than those of Altenkirch et al. and Balat et al. First, it would seem desirable sometimes to apply the $+.\eta$ rule as a reduction. For instance, one might hope that the normal form of $\mathsf{lam}(x^{0+0}, x)$ be itself rather than:

$$\mathsf{lam}(x^{0+0}, \mathsf{case}\ x\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{inj}_1(x_1) \mid x_2 \Rightarrow \mathsf{inj}_2(x_2)))$$

It seems feasible to adapt normalisation by evaluation to do this, but it requires further work. Second, the scope of case splits can sometimes be pushed further into a term. The normal forms described so far lift case splits up as far as possible. For example:

$$\mathsf{lam}(x, \mathsf{case}\ x\ \mathsf{of}\ (x_1 \Rightarrow \mathsf{lam}(y, m_1) \mid x_2 \Rightarrow \mathsf{lam}(y, m_2)))$$

can be rewritten as:

$$\mathsf{lam}(x, \mathsf{lam}(y, \mathsf{case}\ x\ \mathsf{of}\ (x_1 \Rightarrow m_1 \mid x_2 \Rightarrow m_2)))$$

The reason why the case split is lifted up as far as possible is to ensure that case splits on the same guard are unified into a single case split. However, this case split can often be pushed back down whilst still remaining unified, as in the example above.

The algorithms of Altenkirch et al. [ADHS01] and Balat et al. [BCF04] are not directly applicable to λMIL, although they can be adapted. We have a prototype implementation in SML.NET. In fact the algorithm for λMIL (or λ_{ml*}) is rather simpler than the one for λ^+. Roughly, this is because of the fact that functions always return computations. Thus, if a function returns a sum, it must be a sum computation, which must be bound to a variable before being used. Unfortunately, the naïve approach renames each such application, even it has no side-effects. This immediately removes one of the main benefits of the normalisation by evaluation algorithms for λ^+, that is, in removing redundancy. It would be interesting to try to recover the elimination of redundancy. One, rather radical, approach would be to extend λMIL to allow pure functions.

8.4.2 Revisiting Normalisation by evaluation

Normalisation by evaluation by program transformation We have seen that normalisation by evaluation algorithms can be obtained by a series of program transformations. We believe that this approach offers a particularly intuitive insight into the nature of normalisation by evaluation. We claim that it is possible to use program transformations to derive each of our normalisation by evaluation algorithms. Formalising this process would give an alternative method for proving correctness of normalisation by evaluation.

Normalisation by evaluation from normal forms There is a striking correspondence between the structure of normal forms and the structure of normalisation by evaluation algorithms. If one already knows the structure of normal forms then this makes the task of finding a suitable normalisation by evaluation algorithm considerably easier. If we restrict ourselves to a compositional semantics, expressed as a parameterised semantics, and take advantage of the convertibility relation, then this constrains the possible semantics. We suspect that in general it should be possible to generate a suitable model and normalisation by evaluation algorithm from normal forms in this way.

The semantics can be seen as an encoding of normal forms. For instance, in the standard normalisation by evaluation algorithm for simply-typed λ-calculus with long normal forms, neutral terms are simply encoded as themselves, and lambda abstractions are encoded as functions between semantic objects.

Formal proofs The main focus of this thesis has been practical applications of normalisation by evaluation, but it should be possible to prove formally that our normalisation by evaluation algorithms are correct. Perhaps the most important omissions are a formal treatment of fresh names and correctness proofs for the ML implementations. Fresh names can be handled using either FM-set theory [GP01], term families [BES98] or a name generation monad [Fil01b]. As far as correctness of the ML implementations goes, the first problem is that although the method of §2.7.2 shows that a normalisation by evaluation algorithm is correct, it does not show that following a call-by-value evaluation strategy will lead to termination. The second problem is that if one wants to be completely formal then it is necessary to show that the semantics of ML actually corresponds with that of the metalanguage. Filinski and Dybjer [DF02] and Filinski and Rohde [FR04] do this using realisability interpretations.

Partial normalisation by evaluation Although a few of our calculi are partial, in the sense that not all terms have normal forms, we have mainly focused on total calculi — which makes sense for a compiler. It would be interesting to apply the techniques of Filinski and Rohde [FR04] in order formally to analyse normalisation by evaluation with recursive types and the alternative untyped normalisation by evaluation algorithm.

8.4.3 Extending normalisation by evaluation further

Type isomorphisms Type isomorphisms give rise to a number of useful optimisations in functional compilers. For instance, SML.NET has an arity-raising transformation which takes advantage of the isomorphism:

$$A_1 \to \cdots \to A_n \to B \simeq (A_1 \to \cdots \to A_n) \to B$$

If a function $f{:}A_1 \to \cdots \to A_n \to B$ occurs only fully applied, then it can be transformed into a function $f'{:}(A_1 \to \cdots \to A_n) \to B$. Correspondingly an application $f a_1 \ldots a_n$ can be transformed into $f'(a_1, \ldots, a_n)$.

Another example is the deunit transformation, which takes advantage of type isomorphisms involving the unit type, such as:

$$A \times \mathbf{1} \simeq A$$

Curien and Di Cosmo [CD91] show how a second order typed λ-calculus with products and unit can be made confluent in the presence of η-contraction on products and functions, and the $\mathbf{1}.\eta$-expansion rule:

$$m^{\mathbf{1}} \longrightarrow *$$

by adding a family of reductions derived from unit type isomorphisms.

It should be possible to implement a normalisation by evaluation algorithm for such a system. Essentially isomorphic types would become identified in the semantics. In this context the $\mathbf{1}.\eta$-expansion rule should be viewed as one of the family of type isomorphism rules, rather than as an η-expansion. The semantics of the absorbing value term unknown appears to be strikingly similar to that of $*$ in this setting.

Incorporating the unit type isomorphisms in the normalisation by evaluation algorithm for MIL would allow us to merge the deunit transformation into the simplify transformation. As well as cutting out a stage of compilation this would also have the advantage of ensuring that terms were normal with respect to unit isomorphism rules after each *simplify* stage.

Normalisation by evaluation for algebraic structures The free monoid is a simple structure. Correspondingly it gives rise to a simple normalisation by evaluation algorithm. In collaboration with Danvy, we have implemented similar normalisation by evaluation algorithms for a variety of more complex algebraic structures, such as: a hierarchy of free monoids each distributing over those lower in the hierarchy, groups, rings and boolean algebras.

Perhaps the most complex algebraic structures we have looked at arose from the equational theory given by Tarksi's High School Maths problem [BY01]. The equational theory is interesting because it agrees with the theory of type isomorphisms in a categorical model of simply-typed λ-calculus with sums [FCB02]. What makes the problem more interesting is that the equational theory is not complete either for the standard model of arithmetic or the categorical model. Furthermore, it has been shown that there is no finite axiomatisation of either model. However, equality of terms in both models is decidable. We have implemented a normalisation by evaluation algorithm which performs normalisation by evaluation with respect to the equational theory. It would be interesting to try to define a notion of semantic normal form, and a corresponding normalisation by evaluation algorithm for each of the models.

Effects One of the primary strengths of MIL is its effect-typing system. This makes it easy to express a variety of effect-based optimisations. One example, which appears quite straightforward to incorporate in normalisation by evaluation, is the use of effect information to simplify exception handlers. We believe it should be possible to incorporate other effect-based optimisations as well. An obvious one which would be likely to be useful is dead-code elimination for effect-free computations:

$$\mathsf{let}\ x \Leftarrow m\ \mathsf{in}\ n \equiv n$$

where m has no side-effects and $x \notin fv(n)$.

Common value elimination Unrestricted β-reduction tends to produce very large normal forms. This is nicely illustrated by the order of magnitude increase in the size of the term output by the `hamlet` benchmark, against the size of the input term. One solution is to perform restricted β-reductions such as shrinking reductions. But this changes the residualising semantics, and does not seem to fit so well with the normalisation by evaluation framework.

Balat and Danvy [BD02] describe how to perform some common value elimination, using memoisation, in an ML implementation of TDPE. This covers conversions such as:

$$\mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ C[\mathsf{letval}\ y \Leftarrow v\ \mathsf{in}\ m] \equiv \mathsf{letval}\ x \Leftarrow v\ \mathsf{in}\ C[m[y := x]]$$

However, their approach does not produce canonical normal forms with respect to the semantics, and does not detect the same value appearing in separate branches of a term. Essentially the problem is the same as the one of extensional normalisation for simply-typed λ-calculus with sums [ADHS01, BCF04]. We believe that a similar solution can be adopted. In particular, it should be possible to make use of the set/cupto control operators in order to implement a normalisation by evaluation algorithm which performs common value elimination.

8.4.4 Shrinking reductions

Graph-based representations Making more extensive use of the graph-based representation would allow many transformations to be written in a different style, for example replacing explicit environments with extra information on binding nodes, though this does not interact well with the hash-consing currently used for types [SLM98].

More speculatively, we would like to investigate more principled mutable graph-based intermediate representations. There has been much theoretical work on graph-based representations of proofs and programs, yet these do not seem to have been exploited in compilers for higher-order languages (though of course, compilers for imperative languages have used a mutable flow-graph representations for decades). With a careful choice of representation, some of our transformations (such as the commuting conversion for let) could simply be isomorphisms, and we believe that a better treatment of shared continuations in the other commuting conversions would also be possible.

Types It should be possible to use a graphical representation for types, and def-use structures for type variables. However, we have not done this with MIL as SML.NET uses a special hash-consed representation for types [SLM98]. This allows for a great deal of sharing, which is necessary because types can get very big. Unfortunately this sharing is incompatible with the in-place updates one would like to do on the graphical representation. The tension between sharing and mutability seems to be a critical issue in the choice of intermediate representations. An alternative approach which might work is to add let bindings at the level of types as in the TILT compiler [PCHS00]. This allows one to make sharing explicit.

Keeping the immutable representation It has been suggested [Gon04] that it may in fact be possible to implement the linear algorithm using an immutable term representation with def-use information on the side. This would be particularly useful for incorporating into existing compilers which use an immutable representation. It is not clear how feasible this would be, or how well it would scale, but it would be interesting to investigate.

Eliminating the immutable representation Using the graphical representation all the way through would allow us to get rid of the expensive calls to *mutify* and *demutify*. This should give a large improvement in compilation time — potentially speeding up contraction by a factor of two to three. The experience of translating *deunit* suggests that the speed of other transformations should not be adversely affected by using the graphical representation.

Bibliography

[AA00] Andreas Abel and Thorsten Altenkirch. A predicative strong normalisation proof for a λ-calculus with interleaving inductive types. In *Types for Proof and Programs, International Workshop, TYPES '99, Selected Papers*, volume 1956 of *Lecture Notes in Computer Science*, 2000.

[Aba00] Martín Abadi. $\top\top$-closed relations and admissibility. *Mathematical Structures in Computer Science*, 10(3):313–320, June 2000.

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19. ACM Press, 2003.

[ADHS01] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, Boston, Massachusetts, June 2001.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199, 1995.

[AHS96] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 98–106, 1996.

[AHS97] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for system F. Available from `http://www.cs.nott.ac.uk/~txa/publ/f97.pdf`, 1997.

[AJ97] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

[AJ04] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14(4):587–611, August 2004.

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

[AU04] Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for $\lambda^{\to 2}$. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 260–275, 2004.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Number 103 in Studies in Logics and the Foundations of Mathmatics. North Holland, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume II, pages 118–309. OUP, January 1992.

[BBdP98] P. N. Benton, Gavin Bierman, and Valeria de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, 1998.

[BCF04] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *31st Symposium on Principles of Programming Languages (POPL 2004)*, pages 64–76. ACM Press, January 2004.

[BD95] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *Proceedings of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 47–61, 1995.

[BD02] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, Pittsburgh, USA, January 2002.

[Ben04] Nick Benton. Embedded interpreters. To appear, 2004.

[Ber93] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Proceedings of TLCA '93. International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 91–106. Springer-Verlag, Utrecht, The Netherlands, March 1993.

[BES98] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In *Prospects for Hardware Foundations (NADA)*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137, 1998.

[BES03] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Term rewriting for normalization by evaluation. In *International Workshop on Implicit Computational Complexity (ICC'99)*, volume 183(1) of *Information and Computation*, pages 19–42. Academic Press, May 2003.

[BHM02] P Nicholas Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics; Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer-Verlag, 2002.

[BHR01] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, pages 420–447. Springer-Verlag, 2001.

[BHT97] Gilles Barthe, John Hatcliff, and Peter Thiemann. Monadic type systems: Pure type systems for impure settings. In *HOOTS II: Proc. 2nd Workshop on Higher-Order Operational Techniques in Semantics*, ENTCS 10. Elsevier, 1997.

[BK99] Nick Benton and Andrew Kennedy. Interlanguage working without tears: blending sml with java. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 126–137. ACM Press, 1999.

[BK00] Nick Benton and Andrew Kennedy. Monads, effects and transformations. In Andrew Gordon and Andrew Pitts, editors, *Electronic Notes in Theoretical Computer Science*, volume 26. Elsevier, 2000.

[BK01] P Nicholas Benton and Andrew J Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, July 2001.

[BKLR05] Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. Shrinking reductions in SML.NET. In *Proceedings of IFL '04*, Lecture Notes in Computer Science, 2005. A preliminary version appeared in [ifl04].

[BKR] Nick Benton, Andrew Kennedy, and Claudio Russo. The SML.NET 1.1 user guide. `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/smlnet.pdf`.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the third ACM SIGPLAN international conference on functional programming*, pages 129–140. ACM Press, 1998.

[BKR04] Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: The SML.NET experience. In *6th International Conference on Principles and Practice of Declarative Programming (PPDP 2004)*. ACM Press, 2004.

[BLL98] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial Species and Tree-like Structures*. Cambridge University Press, 1998.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Los Alamitos, 1991. IEEE Computer Society Press.

[BY01] Stanley Burris and Karen Yeats. The saga of the high-school identities. Preprint, July 2001.

[CD91] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction for the λ-calculus with surjective pairing and terminal object. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 291–302. Springer-Verlag New York, Inc., 1991.

[CD97] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

[CDS98] Djordje Cubric, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8(2):153–192, 1998.

[CFW86] W Clinger, D P Friedman, and M Wand. A scheme for a higher-level semantic algebra. pages 237–250, 1986.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[CJW00] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 56–71. Springer-Verlag, 2000.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 22, pages 440–464. The MIT Press, 1990.

[Coq90] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In *Logic and Computer Science*, pages 91–122. Academic Press, 1990.

[cS04] Chung chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the ACM SIGPLAN 2004 Scheme Workshop*, pages 99–107, September 2004.

[Dan92] Olivier Danvy. Back to direct style. In *Symposium proceedings on 4th European symposium on programming*, pages 130–150. Springer-Verlag, 1992.

[Dan96] O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg, Florida, January 1996.

[Dan98] Olivier Danvy. Type-directed partial evaluation. Technical Report LS-98-3, BRICS, December 1998.

[dB70] N. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, number 125 in Lecture Notes in Mathematics, pages 29–61. Springer-Verlag, 1970.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Math.*, 34:381–392, 1972.

[DD98] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,* (Gothenburg, Sweden, May 8–9, 1998), number NS-98-8 in Notes

Series, Department of Computer Science, University of Aarhus, December 1998. BRICS.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM Press, 1990.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science,* 2(4):361–391, 1992.

[DF02] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In *Applied Semantics: Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[dG02] Philippe de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Inf. & Comp.*, 178(2):441–464, 2002.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM Press, 1991.

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming,* pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[DRR01] Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose. Functional pearl: Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, November 2001.

[FCB02] Marcelo P. Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In

Proceedings of the 17th IEEE Symposium22-25 July 2002 on Logic in Computer Science, page 147. IEEE Computer Society, 2002.

[Fel88] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190. ACM Press, 1988.

[Fil94] Andrzej Filinski. Representing monads. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'94, Portland, OR, USA, 17–21 Jan. 1994*, pages 446–457. ACM Press, New York, 1994.

[Fil96] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996.

[Fil99a] Andrzej Filinski. Representing layered monads. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 175–188, New York, NY, 1999.

[Fil99b] Andrzej Filinski. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming*, pages 378–395, 1999.

[Fil01a] Andrzej Filinski. An extensional CPS transform (preliminary report). Technical Report 545, Computer Science Department, Indiana University, 2001.

[Fil01b] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *Typed Lambda Calculi and Applications : 5th International Conference, TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, Krakow, Poland, May 2001.

[Fil02] Andrzej Filinski. Personal communication, May 2002.

[Fio02] Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *4th International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*. ACM Press, 2002.

[FR04] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference*, volume 2987 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, March 2004.

[FT04] Carsten Führmann and Hayo Thielecke. On the call-by-value CPS transform and its semantics. *Information and Computation*, 188(2):241–283, 2004.

[FWFD88] Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 52–62. ACM Press, 1988.

[Gal90] Jean Gallier. On girard's "candidats de reductibilité". pages 123–203. Academic Press, 1990.

[Gha95] N Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, November 1995.

[GI91] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination de coupures dans l'arithméticque d'ordre supérieur*. Thèse de doctorat d'état, U. Paris VII, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Gon04] George Gonthier, January 2004. Personal communication.

[GP01] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[GR04] Emden R. Gansner and John H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, 2004. Online version available at: `http://www.standardml.org/Basis/`.

[GRR95] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, June 1995.

[GRR98] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. Return types for functional continuations. A preliminary version appeared as [GRR95], 1998.

[GS02] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 271–282. ACM Press, 2002.

[GY99] Bernd Grobauer and Zhe Yang. The second futamura projection for type-directed partial evaluation. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 22–32. ACM Press, 1999.

[Han94] Chris Hankin. *Lambda calculi - A guide for computer scientists*, volume 3 of *Graduate texts in computer science*. Oxford University Press, 1994.

[HD94] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Conference Record POPL '94*, pages 458–471. ACM Press, 1994.

[HD97] John Hatcliff and Olivier Danvy. Thunks and the lambda-calculus. *Journal of Functional Programing*, 7(3):303–319, 1997.

[Hof99] Martin Hofmann. Nbe with a logical relation. Unpublished, October 1999.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.R. Seldin and R.J. Hindley, editors, *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 479–490. Academic Press, London, New York, 1980.

[Hue76] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse d'état, Université Paris 7, Paris, France, 1976.

[Hue97] Gerard Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[Hug86] R J M Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[ifl04] 16th international workshop on implementationa and application of functional languages, (IFL'04). Technical Report 0408, Institute of Computer Science and Applied Mathematics, University of Kiel, September 2004.

[JG89] P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 218–226. ACM Press, 1989.

[JG95] C.B. Jay and N Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, 1995.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[JM97] Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language. In *ACM SIGPLAN Workshop on Types in Compilation*, 1997.

[Joy81] A. Joyal. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42:1–82, 1981.

[Joy87] A. Joyal. Foncteurs analytiques et espèces de structures. In *Combinatoire énumerative*, volume 1234 of *Lecture Notes in Mathematics*, pages 126–159. Springer-Verlag, 1987.

[Kam04a] Yukiyoshi Kameyama. Axiomatizing higher level delimited continuation. In *Proceedings of the Fourth ACM-SIGPLAN Continuation Workshop (CW'04)*, pages 49–53, January 2004.

[Kam04b] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'04)*, pages 442–457, September 2004.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[KH03] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 177–188. ACM Press, 2003.

[Lan64] P.J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[LD94] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 227–238. ACM Press, 1994.

[Lil99] Mark Lillibridge. Unchecked exceptions can be strictly more powerful than call/cc. *Higher-Order & Symbolic Computation*, 12(1):75–104, 1999.

[LS05] Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *Proceedings of TLCA '05*, Lecture Notes in Computer Science, April 2005.

[McB01] Connor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.

[ML75] P. Martin-Löf. An intuitionisitc theory of types, predicative part. In *Logic Colloquium 1973*, pages 73–118, 1975.

[mlj] MLj compiler:
`http://www.dcs.ed.ac.uk/home/mlj/`.

[mlt] MLton SML benchmarks:
`http://www.mlton.org/performance.html`.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Mog99] Torben Æ. Mogensen. Gödelisation in the untyped lambda calculus. In Olivier Danvy, editor, *Proceedings of PEPM*, San Antonio, Texas, January 1999.

[Mor95] G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.

[Nec00] George C. Necula. Proof-carrying code (abstract): design, implementation and applications. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–177. ACM Press, 2000.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[Par97] Michel Parigot. Proofs of strong normalisation for second order classical natural deduction. *The Journal of Symbolic Logic*, 62(4):1461–1479, December 1997.

[PCHS00] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, School of Computer Science, Carnegie Mellon University, December 2000.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208. ACM Press, 1988.

[Pit00] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. CUP, April 2003.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Pra71] Dag Prawitz. Ideas and results in proof theory. In *Proceedings of the 2nd Scandinavian Logic Symposium*, number 63 in Studies in Logics and the Foundations of Mathmatics, pages 235–307. North Holland, 1971.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM Press, 1993.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

[Rey98] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[Ros] Kristoffer Høgsbro Rose. Type-directed partial evaluation in Haskell. Presented at [DD98].

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 116–129. ACM Press, 1995.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):287–358, 1993.

[Sha97a] Zhong Shao. An overview of the FLINT/ML compiler. In *ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[Sha97b] Zhong Shao. Typed common intermediate format. In *1997 USENIX Conference on Domain-Specific Languages*, pages 89–102, Santa Barbara, CA, Oct 1997.

[She97] Tim Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 22–35. ACM Press, 1997.

[SK01] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2-3):101–142, 2001.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 313–323. ACM Press, 1998.

[smla] SML.NET compiler:
`http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.

[smlb] Standard ML of New Jersey (SML/NJ) compiler:
`http://cm.bell-labs.com/cm/cs/what/smlnj/`.

[SPG03] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

[SW97] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):916–941, November 1997.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[TBE$^+$01] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, October 2001.

[Thi98] Hayo Thielecke. An introduction to Landin's "a generalization of jumps and labels". *Higher Order and Symbolic Computation*, 11(2):117–123, 1998.

[Thi03] Hayo Thielecke. From control effects to typed continuation passing. In *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 139–149. ACM, 2003.

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[Tol98] Andrew P. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Proceedings of the Second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 97–115. Springer-Verlag, 1998.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[Ves] René Vestergaard. The polymorphic type theory of normalisation by evaluation. (Preliminary version).

[Ves01] René Vestergaard. The simple type theory of normalisation by evaluation. *Electronic Notes in Theoretical Computer Science*, 57, 2001. Preliminary proceedings are available as technical report number 2001.2359, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.

[VM04a] Jérôme Vouillon and Paul-André Melliès. Recursive polymorphic types and parametricity in an operational framework. Submitted for publication, 2004.

[VM04b] Jérôme Vouillon and Paul-André Melliès. Semantic types: a fresh look at the ideal model for types. In *31st Symposium on Principles of Pro-*

gramming Languages (POPL 2004), pages 52–63. ACM Press, January 2004.

[Vor97] Sergei Vorobyov. The “hardest” natural decidable theory. In *Proceedings of the 12th Symposium on Logic in Computer Science (LICS ’97)*, page 294. IEEE Computer Society, 1997.

[Vor04] Sergei Vorobyov. The most nonelementary theory. *Information and Computation*, 190(2):196–219, 2004.

[Vou04] Jérôme Vouillon. Subtyping union types. In *Proceedings of CSL ’04*, number 3210 in Lecture Notes in Computer Science, pages 415–429. Springer-Verlag, 2004.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM Press, 1990.

[Wad94] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, 1994.

[Wad99] Philip Wadler. The marriage of effects and monads. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP ’98)*, volume 34(1), pages 63–74, 1999.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

[WT03] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003. An earlier version appeared as [Wad99].

[Yan99] Zhe Yang. Encoding types in ML-like languages. *ACM SIGPLAN Notices*, 34(1):289–300, 1999.