SPCF: Interpreter Development and Affine Program Transformation Showcase

Jay Rabjohns

Bachelor of Science in Computer Science
The University of Bath
2023/2024

SPCF: Interpreter Development and Affine Program Transformation Showcase

Submitted by: Jay Rabjohns

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

We explore the concept of function denesting in SPCF, a derivative of typed λ-calculus with non-local control. Function nesting involves sharing variables from one function's scope with another function's arguments. This transformation is based on the work of Laird who provides denotational semantics for a retraction on terms of a higher type into a lower one, and can be seen as rewriting programs to adhere to affine typing rules. We illustrate this transformation working in the real world and evaluate its benefits.

As part of the project we also build and release an interpreter for SPCF, a language that currently lacks one which is widely available. This both has the focus of illustrating the denesting action working, as well as contributing to the language's ecosystem, hopefully making it easier to experiment with the language.

Contents

List of Figures

List of Tables

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Jim Laird, for the endless source of expertise and support throughout the year. This project has been a joy from front to cover and it certainly wouldn't have been possible without your thoughtful explanations and feedback.

sI would also like to thank my family and friends for their perennial support and endless proofreading throughout the project.

Chapter 1

Introduction

It is common practice in programs to nest function calls, that is, to share variables from a function's scope as arguments of another function. It allows for incredibly terse and expressive programs at the cost of complicating their implementations. Laird (2007) shows that it is theoretically possible to transform programs to an equivalent form which eliminates this behaviour. More formally, any term in SPCF is observationally equivalent to a term of the affinely typed sub-language ASPCF. Affine typing in this context means that recursion, as well as state sharing between functions, is prohibited as both can be seen as forms of function nesting.

This transformation can be viewed as a form of refactoring, where programs are altered without changing their behaviour. Refactoring is useful and exists in many forms. One is a sort of silent refactoring or optimisation done by the compiler, for example, tail recursion optimisation, a popular optimisation where certain recursive functions may reuse a stack frame rather than generate new ones with each nested call. Another form of refactoring is explicit refactoring done by the programmer, where code is rewritten and improved in some aspect. The described denesting action could be seen as a kind of refactoring which lies in the middle, where source code is algorithmically modified but also persists between runs. There are many potential implications, in particular a possible debug tool to deconstruct and visualise a program's possible execution paths.

The goal of this project is to illustrate this working as a piece of software and evaluate its viability in the real world.

The easiest way to observe the transformation working is by evaluating terms with an interpreter. There are no widely available interpreters for SPCF so this project aims to build one to both help illustrate the denesting action working but also to contribute to SPCF's ecosystem and to allow easier experimentation with the language.

We build up to an in-depth discussion on the action in Chapter 4, covering prerequisite concepts along the way and interpreter development in Chapters 2 and 3.

Chapter 2

Literature, Technology, and Data Survey

This chapter discusses the literary and historical background that the project relies on. It starts by introducing concepts relating to abstract computation and discussing their historical significance before ultimately landing on concrete implementation goals for the project. Some of the topics include:

- The λ-calculus and its computational model
- SPCF and extensions of the λ-calculus
- Nesting removal in SPCF
- Implementation details for an SPCF interpreter

2.1 The λ-calculus

The λ-calculus is an important tool in the field of functional programming. It is an abstract model of computation introduced by Church (1936) which provides compact semantics for studying computation. It is analogous to a simple yet powerful programming language. While it was originally used to study the foundations of mathematics, specifically the 'Entscheidungsproblem' or 'Decision Problem', it has since been adapted and expanded to accommodate a wide array of domains, including being the basis for functional programming. It is computationally complete, meaning it can represent any computable function, or equivalently it can simulate any Turing machine Turing (1937).

Terms of the λ-calculus are defined by a Backus-Naur form (BNF) grammar (2.1). Each term denotes a function which may be applied to any other term. x is one of infinitely many variables, represented as a string label. $\lambda x.N$ denotes an abstraction, a function to evaluate M by binding all free occurrences of x in M. λ-abstractions may be regarded as the suspended execution of a function, enabling it to be composed and reasoned about before its evaluation. MN is the application of an argument N to a function M. Any valid term in the λ-calculus can be constructed from a combination of these terms, highlighting its conciseness and further making the fact it is Turing complete quite incredible.

$$M, N ::= x \mid \lambda x.M \mid MN \tag{2.1}$$

2.1.1 Computation in the λ-calculus

Computations in the lambda calculus are usually presented as a series of transformations $M \to M' \to M'' \to \ldots$. The basic computation step is a β-reduction (2.2), where a term $(\lambda x.M)N$ is said to reduce to $M[N/x]$, meaning that every occurrence of x in M is substituted with N. This is an example of a reducible expression or β-redex. A normal form is a term to which no further computation can be performed, they are significant because ultimately they provide a convenient way of defining relations between terms. One non-obvious takeaway from this is that terms can contain free variables, variables not bound by a surrounding abstraction. This will become especially relevant later in Section 2.3 when discussing argument sharing.

$$(\lambda x.M)N \to_\beta M[N/x] \tag{2.2}$$

Terms may have relations defined between them, $M \sim N$, meaning that M and N are related by some operation $\sim$. Equivalence relations express that terms are equivalent, modulo something related to the relation. For example, α-equivalence, $M =_\alpha N$, states that terms are equivalent if they are identical in every respect except for variable names. β-equivalence, $M =_\beta N$, holds when M and N both reduce to a common term, denoted $M \to_\beta^* N$. Each substitution is assumed to avoid variable capture, where substituted variables are renamed to not conflict with existing bounds, $M[N/x] =_\alpha M$. The most relevant equivalence relation to this project is observational equivalence, $M \simeq N$, where terms are considered to be equivalent if their outputs are indistinguishable for any given input. This will become more relevant in Section 2.3.

Many familiar high-level constructs are definable in the λ-calculus, such as booleans, if-then-else expressions, natural numbers, and numerical operations like addition. These constructs facilitate real computation in terms of the λ-calculus. Church introduced a series of encodings for these, aptly named the Church encodings, which cleverly use higher-order functions to encode these familiar constructs. Examples of these encodings for booleans and if-then-else are provided (2.3). Commonly, a series of named constants are defined as syntactic sugar along with the language grammar, making programs considerably more readable.

The encodings for numerals and associated operators have been omitted for brevity.

$$\begin{aligned} true &= \lambda x.\lambda y.x \\ false &= \lambda x.\lambda y.y \\ \text{ifthen} &= \lambda b.\lambda x.\lambda y.bxy \end{aligned} \tag{2.3}$$

An example reduction of $ifthen$ can be seen below.

$$\begin{aligned}
&\text{ifthen } true\ M\ N \to^*_\beta M \\
&\text{ifthen } true\ M\ N = (\lambda b.\lambda x.\lambda y.bxy)(\lambda x.\lambda y.x)MN \\
&\qquad\qquad \to_\beta (\lambda x.\lambda y.(\lambda x.\lambda y.x)xy)MN \\
&\qquad\qquad \to_\beta (\lambda y.(\lambda x.\lambda y.x)My)N \\
&\qquad\qquad \to_\beta (\lambda x.\lambda y.x)MN \\
&\qquad\qquad \to_\beta (\lambda y.M)N \\
&\qquad\qquad \to_\beta M
\end{aligned}$$

Oppositely, it can be said that $ifthen\ false\ M\ N \to^*_\beta N$, but the full reduction is omitted here.

A function which is *recursive* references itself in its definition. In the λ-calculus, recursion is modelled by a so-called 'fixed point' operator F such that $M =_\beta FM$ its input is equal to its output. Every term in the lambda calculus has at least one fixed point. The fixed point of a term can be found through a so-called fixed point 'combinator', which cleverly uses self-application to deduce the fixed point. Many fixed point combinators exist, one of the simplest is the Y-combinator (2.4) introduced by Curry (1930). A combinator is simply a closed term, a term with no free variables. As an aside, the constants such as $ifthen$ and $true$ defined earlier are also kinds of combinators.

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{2.4}$$

2.1.2 Typed λ-calculi

Extending the λ-calculus by editing its grammar is possible and indeed common. One such way is the addition of types, creating a typed λ-calculus. Types encode additional information about terms, including a more restrictive set of rules for application. Adding types is a trade-off, some terms are no longer expressible, but more can be proven about the remaining terms. Generality is lost in favour of specificity.

The simply typed λ-calculus (STLC) is one variant of typed λ-calculus originally introduced by Church (1940) to address what he felt was a paradox in his untyped lambda calculus, self-application. An example of self-application is the term $\lambda x.xx$, commonly denoted Ω, which is the prototypical divergent program. A grammar for the STLC is given by (2.5), where types are either a base type o or constructed from two types $\sigma \to \tau$. For example, see that the term $(\lambda x^\tau.M)N$ is only valid as a typed term if N has type τ.

$$\begin{aligned}
M, N &::= x \mid \lambda x^\tau.M \mid MN \\
\tau, \sigma &::= o \mid \sigma \to \tau
\end{aligned} \tag{2.5}$$

The significance of preventing self-application is that recursion is no longer possible. However, in return, the STLC is strongly normalising, a property indicating that every term terminates to a β-normal form (Tait, 1967). This is illustrated by the Y-combinator (2.4) being untypable, any type assigned to x will always lead to a contradiction for the type of f. This is a significant tradeoff which influenced the design of further extensions of the λ-calculus, which we discuss in the coming sections.

2.2 Extensions of the simply typed λ-calculus

2.2.1 Logic of Computable Functions (LCF)

In 1969, Dana Scott proposed the logic of computable functions, a logic which serves as the basis for a typed calculus supporting fixed point recursion. It relies on the concept of a complete partial order (CPO), a relation between sets '$\leq$' that captures the idea of ordered computation. More specifically, a partial order relates elements of a set, it is complete in the sense that each sequence of partially ordered elements has a least upper bound. In the context of fixed points, Scott shows that it is possible to construct a 'least fixed point', which is the smallest solution to a fixed-point equation. Its existence implies that the recursion converges to a solution. Note that this does not imply that the function terminates. For example, a function which always returns a constant value and never terminates is considered to have converged. LCF has laid the groundwork for more complex calculi than the STLC, and indeed Scott later used it to construct his X-Calculus but that is unrelated to the project at hand. This work was originally part of an unpublished note, however, it has since been published as a memorandum by Milner (1973) to make it more accessible.

2.2.2 Programming of Computable Functions (PCF)

PCF is a sequential functional language introduced by Plotkin (1977) which is based on LCF. Whilst LCF is focused on providing a logic for proofs, PCF focuses more on practical computation and is more similar to practical programming languages. It is an extension of the simply typed λ-calculus supporting recursion as well as providing data types and functions out of the box. Its grammar is shown in (2.6), and similar to the STLC types are either of ground type or constructed between two existing types.

$$\begin{aligned} M, N &::= n \mid f \mid x_\tau \mid (\lambda x_\sigma . M_\tau)_{\sigma\to\tau} \mid (M_{\sigma\to\tau}\ N_\sigma)_\tau \\ n &::= 1, 2, 3, \ldots \\ f &::= succ_{o\to o} \mid pred_{o\to o} \mid cond_{o\to o\to o\to o} \mid Y_{(\tau\to\tau)\to\tau} \\ \tau, \sigma &::= o \mid \sigma \to \tau \end{aligned} \tag{2.6}$$

One of the major challenges relating to PCF has been to create a model which is fully abstract, a crucial property for characterising the observational equivalence of programs. Plotkin (1977) admits that for the provided model to be fully abstract, there must exist functions capable of computing more than one argument simultaneously, which contradicts PCF being a sequential language. This is due to the CPO model of PCF being defined over continuous functions despite the language only being able to represent sequential functions. Further models have been developed which provide full abstraction, notably one by Milner (1977), however, these are considered less than satisfactory. Later, Loader (1996) disproved the existence of an *effectively representable* fully abstract model of finitary PCF, which is another important property regarding the definition of observational equivalence. Another approach to the problem is to extend the language rather than to modify its model, enabling the definition of a sequential and fully abstract language. This is what will be discussed in the next section.

2.2.3 Sequential PCF (SPCF)

SPCF is an extension of PCF developed by Cartwright and Felleisen (1992) which introduces error generators and escape handlers, acting as two kinds of control operators. Error generators describe the misapplication of terms and escape handlers can be thought of as escaping from local evaluation of a phrase.

$$\begin{aligned} M, N &:= n \mid f \mid e \mid x^\tau \mid (\lambda x^\sigma.M^\tau)^{\sigma\to\tau} \mid (M^{\sigma\to\tau}\ N^\sigma)^\tau \\ n &= 1, 2, 3, \ldots \\ f &= succ^{o\to o} \mid pred^{o\to o} \mid if0^{o\to o\to o\to o} \mid Y^{(\tau\to\tau)\to\tau} \\ e &= {error_1}^o \mid {error_2}^o \mid catch^{\tau_1\to\ldots\tau_n\to o} \end{aligned} \tag{2.7}$$

$$\tau, \sigma := o \mid \sigma \to \tau \tag{2.8}$$

Observing evaluation order with errors

Functions in SPCF are error-sensitive, meaning that if an argument evaluates to an error the function also returns an error. This error propagation is analogous to the behaviour of *try...catch* statements commonly found in practical languages. Error sensitivity allows a programmer to determine the evaluation order of a function's arguments by substituting them with distinct error values and observing which one is thrown. For example, consider two possible recursive definitions of addition in SPCF. One recurses on its first parameter and the other on its second.

$$+_l = Y(\lambda +.(\lambda x.\lambda y.\ if0\ x\ y\ succ(+\ (pred\ x)\ y)))$$
$$+_r = Y(\lambda +.(\lambda x.\lambda y.\ if0\ y\ x\ succ(+\ x\ (pred\ y))))$$

Due to functions being error-sensitive, the programmer can manually apply errors in different orders and observe the changes in behaviour. $(+_l\ error_1\ error_2)$ evaluates x first and so it returns $error_1$, $(+_r\ error1\ error2)$ evaluates y first and so it returns $error_2$. This behaviour is precisely what makes SPCF *observably* sequential and what ultimately enables the construction of a fully abstract sequential language.

Observing evaluation order with catch

The catch operator is introduced as part of SPCF such that the evaluation order of a function's arguments may be determined internally as part of a program. Errors enable this to be done manually, where one can apply different combinations of errors as function arguments and remember the result.

Cartwright and Felleisen (1992) introduce a family of catch procedures based on the catch construct found in the original version of Scheme. They define it as a family of procedures $(\tau_1 \to \cdots \to \tau_n \to o)$, which is to say that if f is a function with type $\tau_1 \to \cdots \to \tau_n$, *catch* f will return a base type. *catch* f returns the index of the argument in which it is *strict*, which means the argument which is evaluated first. If f evaluates no arguments and returns a constant, then the constant is returned plus the number of arguments is returned. This catch procedure is equivelantly expressive to the downward catch defined in Scheme but this is slightly simpler to reason about.

We discuss in further detail the implications of errors and catch in Chapter 3, but for now, it is worth knowing that control operators will play a fundamental role in the removal of function nesting, where a series of jumps effectively replace nested calls.

2.3 Nesting Removal in SPCF

It is incredibly common when writing programs to nest function calls. It leads to terse and expressive code but can lead to complicated implementations. A more formal definition of a nested function call is the sharing of variables from a function's scope as arguments of another function. It is possible to refactor a program, that is to change its representation to an equivalent form without changing its behaviour. Laird (2007) outlines a method for this in SPCF by transforming programs to a sublanguage ASPCF and subsequently projecting it back to the original type. Further, he shows that for every term of SPCF M there exists an ASPCF term M' such that $M \simeq M'$.

ASPCF is SPCF with some additional typing constraints, referred to as affine typing. Affine typing restricts the use of variables and removes recursion, enforcing a form of linearity in programs. Commonly, affine typing implies variables may only be used once, but in this case it means variables may have a single reference or copy in use at any given time. Consequently, terms of an application may not share free variables, preventing functions from interfering with one another's evaluation. For example, for an affinely typed term $(A\ B)\ C$, B and C can contain no common free variables. It is for this reason that fixed point recursion is also removed, clearly the inner applications of the Y combinator (2.4) cannot generally guarantee that terms contain no common free variables.

Laird first considers a bounded version of SPCF where types are finite, for example, the booleans or a bounded subset of the natural numbers. As well as this, terms are not recursive. What follows are denotational semantics creating a pair of injection and projection relations between SPCF terms and observationally equivalent ASPCF terms. Secondly, an unbounded call-by-value SPCF is considered. Here, numerals are defined over the natural numbers and terms may have recursive definitions. To not violate affine typing rules, recursion is replaced with iteration. From this, it is shown that unbounded terms have an affinely typed observationally equivalent term, and hence the same method is also applicable to unbounded terms.

While it may be possible to refactor unbounded terms, in reality, it makes little sense to implement and run this process naively for non-terminating programs. The refactoring process would also be unbounded, both in execution time and memory usage. One potential use case is to refactor terms in parts, considering that some paths in an unbounded program may still be bounded and it would be possible to remove nesting for these parts.

The transformation is wholesale, meaning entire branches of the program would have to be considered at once. Importantly it's not possible to denest a function in isolation without also denesting its dependants. The process would have to be done to entire branches or paths of a program at once. A program path is a common conceptualisation of decisions being made in the program, for example, conditionals and function calls both fork the program into multiple paths. From this, it is theoretically possible to lazily refactor the paths of a program and over time return these to the user. This approach would of course also work for the bounded case, where it would terminate as before rather than having

one or many long-standing or potentially infinite tasks.

2.4 Implementation

An appropriate representation for SPCF programs in computer memory is needed, such as an abstract syntax tree (AST) which is commonly used in language compilers and parsers. They represent programs as a hierarchical tree of statements which follow the same syntactic ordering of the original program. This provides a convenient and efficient representation to operate on. The AST could be constructed either through tokenising an input string or by hard coding it. For this project, we provide methods of doing both.

A very natural way to model an AST is through the use of an algebraic data type (ADT), which is supported well by Haskell. ADTs provide a way to define and combine product types, such as structs and tuples, as well as sum types, such as enums. Their expressiveness and ability to model recursive types make them a natural fit to model inductive definitions such as the grammar for SPCF and recursive data structures like ASTs. Jones (2003) discusses idioms related to abstract syntax tree implementation and agrees that functional languages provide a natural basis for them because they support user-defined recursive datatypes. For this interpreter, variables, abstractions, and applications will each have a corresponding representation in the ADT, as well as functional constants such as $succ_{o \to o}$.

Once an AST is constructed, it is possible to evaluate terms by implementing small-step operational semantics set out by Laird and other texts on SPCF. Each reduction step will be implemented as a function which pattern matches on the term's ADT.

In regards to implementing the denesting action, Laird denotationally outlines the injective and projective mappings between SPCF and ASPCF terms which provide guidance on their implementation.

2.4.1 Testing and Evaluation

To test the interpreter functions as expected, nested and denested terms will be evaluated on arbitrary data and the results will be compared. These will be checked by hand at least once to ensure they are correct. For bounded terms, it is possible to map every input and output for the SPCF term and check that they are equivalent to that of the ASPCF term. This should hold since they are observationally equivalent.

Unit tests could be used for the lower level language concepts such as reduction and term construction.

2.5 Conclusion

Laird provides a valuable framework for the removal of function nesting in SPCF. By tracing the lineage of SPCF from PCF, LCF, and the λ-calculus, we contextualise function denesting with a wider scope. We plan clear goals for the project, some potential applications of unbounded function denesting, and briefly cover one of the primary findings of Cartwright and Felleisen in Section 2.2.3. Haskell is a fitting language for implementation due to its support of algebraic datatypes and lazy evaluation. With a well-defined basis, the project's implementation is a natural next step.

Chapter 3

SPCF Interpreter

This chapter introduces the prerequisite tools and foundational material needed for Chapter 4, which discusses the denesting action in detail. This action relies on a wide range of concepts, such as continuation passing style programming, fixed point recursion, and various approaches to typing. We aim to build on Section 2.4 while including notes on implementation.

To effectively showcase the denesting action, we require a suitable representation for SPCF which can be manipulated and reasoned about in computer memory. Additionally, one of the most effective approaches to test its implementation is to evaluate terms pre and post-transformation transformation, ensuring their behaviour is unchanged, which is one of the transformation's key properties. For this reason, we have developed an interpreter for SPCF, complete with a frontend for parsing programs in plaintext. At the time of writing, there are no widely available interpreters for SPCF, meaning that this is done mostly out of necessity, however, this also presents an excellent opportunity to discuss interesting features of SPCF and highlight design decisions required for implementing the denesting action in Chapter 4.

3.1 Execution model

An adequate runtime execution model is required to construct and evaluate terms in SPCF. Haskell is a popular choice for building interpreters for variants of the λ-calculus and a common approach to representing terms is with an algebraic data type (ADT). This is illustrated in Listing 3.1, each term of the grammar (2.7) is represented as a constructor. In terms of abstract syntax trees, leaf nodes represent variables and constants whereas branches are applications, abstractions, and functional constants.

Other variants of lambda calculus share many aspects with SPCF, such as variables, abstractions, applications, and common function constants such as the successor and predecessor. What is in some senses interesting about SPCF is both the typeable fixed point combinator borrowed from LCF and its inclusion of a non-local control operator 'catch'.

Listing 3.1: SPCF AST inductive definition using an ADT

```
data Term
  = Numeral Int
  | Error Error
  | Variable Label
  | Lambda Label Type Term
  | Apply Term Term
  | Succ Term
  | Pred Term
  | YComb Term
  | If0 Term Term Term
  | Catch Term
```

3.1.1 Capture avoiding substitution

Similarly to any other variant of the λ-calculus, SPCF assumes well-defined definitions for α, β, η equivalence, and of course, capture avoiding substitution. Our implementation includes these, with the notable omission of η-expansion because it is not relied upon by other operations. Capture avoiding substitution is often glossed over when discussing abstract language definitions, however seeing as our implementation is concerned with a concrete language definition, we shall briefly discuss it.

Central to capture avoiding substitution is the notion of a fresh variable. Fresh variables are valid variable names which are not already bound in a term. We use Haskell's lazy evaluation to construct an infinite list of variable names of the following form:

$$vars := \{"a", \ldots, "z"\} \cup \{\ ci \mid c \in \{"a", \ldots, "z"\}, i \in \mathbb{N}\}$$
where ci is the concatenation of c and i.

A fresh variable is then chosen by taking the first element of the set which is not already used in the current term. This method guarantees the existence of a fresh variable and minimises computational overhead as it is lazy. Then, after being sure to rename bound instances of the given variable to the fresh variable, substitution can be carried out as normal.

3.1.2 Bounded and Unbounded SPCF

We introduce a bounded variant of SPCF which will serve as a basis for the denesting action working on bounded terms later. This is achieved primarily by removing fixed point recursion and limiting numerals with an upper bound n, denoted with the type $x : \underline{n}$ where $x < n$. Some other simplifications have been made such as removing if0, succ, pred, and error constants. Additionally, it is extended with binary products, n-fold products, and the conditional operator 'case'.

$$\begin{aligned} M, N &:= B \mid F \mid x^\tau \mid (\lambda x^\sigma.M^\tau)^{\sigma\to\tau} \mid (M^{\sigma\to\tau}\ N^\sigma)^\tau \mid M \times N \\ B &= \{\underline{n} \mid n \in \mathbb{N}\} \\ F &= \mathrm{catch}_m^{\tau_1\to\ldots\tau_n\to o} \mid \mathrm{case}^{o\to o^n\to o} \end{aligned} \tag{3.1}$$

Listing 3.1: SPCF inductive definition for types using an ADT

```
data Type
  = Base            -- Base type (numerals and errors)
  | Empty           -- Return type of a non-terminating function
  | (:->) Type Type -- Function
  | Cross Type Int  -- n-fold Product
  | Pair Type Type  -- Binary product
  | Unit            -- Empty product
```

$$\tau, \sigma := o \mid \underline{0} \mid \sigma \to \tau \mid \sigma \times \tau \mid I \tag{3.2}$$

Products

This definition relies on a suitable encoding for binary as well as n-fold products along with projections $\pi_i : \tau^n \Rightarrow \tau$ and $\pi_1 : \tau \times \sigma \Rightarrow \tau$, $\pi_2 : \tau \times \sigma \Rightarrow \sigma$. n-fold products provide a convenient syntax for constructing long collections of terms of uniform type. We provide an implementation of n-fold products by abstracting over a list and making use of its built-in functions for traversal and indexing. The empty product I is simply the empty list.

3.2 Types

Types encode additional information about terms, including how they can be applied to one another. Similarly to other typed λ-calculi, types in SPCF can be encoded as either a 'base' type o or an 'arrow type' $\sigma \to \tau$. This set has been expanded to include an explicit 'Cross' type to encode n-fold, a more general 'Pair' type for binary products, as well as an 'Empty' type to typify non-terminating programs. This is necessary for programs written in continuation passing style, which the denesting action relies upon. The choice of having two types of products is an explicit design decision and is discussed further in the following sections.

Definition 3.1. A typing *context* Γ is a map of variables to types.

A typing context can be represented as a list, for example, $\Gamma = a : \tau, b : \sigma, \ldots$. It is also common to extend contexts, for example, $\Gamma, x : \tau$ reads as the context Γ extended with x having type τ.

Definition 3.2. A typing *judgement* $\Gamma \vdash M : \tau$ is a proposition, stating that given a context Γ, the term M has type τ. A judgement can be true or false, depending on the given context.

Definition 3.3. A typing *rule* determines if it is possible for a term to have a type, sometimes this is called being well-typed. Rules are written like this

$$\frac{Premise_1 \qquad Premise_2 \qquad \ldots \qquad Premise_n}{Conclusion}$$

This can be read as 'given each of the things on the top, the thing on the bottom must be true'. It is possible to have a rule with nothing on the top, which is similar to an axiom or a statement which is always true.

Typing rules for SPCF

Constants

$$\frac{}{\Gamma, n : o \vdash n : o} \tag{3.3}$$

Variables

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{3.4}$$

Abstraction

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.M : \sigma \to \tau} \tag{3.5}$$

Application

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \tag{3.6}$$

Product

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau} \tag{3.7}$$

Projection

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i M : \tau_i} i \in \{1, 2\} \tag{3.8}$$

Case

$$\frac{\Gamma \vdash N : o \qquad \Gamma \vdash P : \tau^n}{\Gamma \vdash \text{case}\langle N, P \rangle : \tau} \tag{3.9}$$

Empty Product

$$\frac{}{\Gamma \vdash \langle\rangle : I} \tag{3.10}$$

If0

$$\frac{\Gamma \vdash C : o \qquad \Gamma \vdash T : o \qquad \Gamma \vdash F : o}{\text{if0 } C \ T \ F : o} \tag{3.11}$$

Successor

$$\frac{\Gamma \vdash M : o}{\Gamma \vdash \text{succ}(M) : o} \tag{3.12}$$

Predecessor

$$\frac{\Gamma \vdash M : o}{\Gamma \vdash \text{pred}(M) : o} \tag{3.13}$$

Y combinator

$$\frac{\Gamma \vdash M : \tau \to \tau}{\Gamma \vdash Y(M) : \tau} \tag{3.14}$$

Catch

$$\frac{\Gamma \vdash M : \tau_1 \to \ldots \tau_n \to o)}{\Gamma \vdash \text{catch}(M) : o} \tag{3.15}$$

3.2.1 Implementation details

There are two primary approaches to interpreting typing rules depending on the order in which layers are read. If read top to bottom, rules describe a type verification algorithm which ensures terms adhere to type rules. If read from bottom to top, they describe a type inference algorithm capable of determining the type of a given term. Both methods traverse a term's syntax tree applying type rules at each sub-expression and both rely on the fact that every term in SCPF has a unique type.

For this project, a type inference algorithm is necessary to construct a general application of the denesting action described in Chapter 4. The action relies on the type structure of terms it is applied to and in some senses describes a family of actions, each of which acts on terms of a specific type. The tree traversal and bookkeeping of variables is done in much the same way as for term evaluation, described in Section 3.3, with the notable addition of applying the type rules specified in 3.2 at each step.

One might wonder why the implementation of the denesting action, which ultimately takes the form of a pair of terms in the language, is written in Haskell and not in SPCF. That is to say, if our implementation is fully expressive, why must we rely on Haskell to construct these terms? The answer lies once again as a typing problem. We have described the typing system of SPCF as monomorphic, where each term has a unique type and the type inference algorithm is similar to the original definition by Curry et al. (1958). To define a term across many types as would be needed here we would require a type system supporting parametric polymorphism. Without this, it is possible to exhaustively define pairs of terms for a finite subset of types, but nothing more. This would be possible and indeed type inference algorithms for these type systems exist (*e.g.*, Hindley, 1969; Milner, 1978) however they are more complex. Therefore we utilise Haskell's more advanced type system to define these terms generically without the additional implementation complexity.

3.2.2 Examples

Consider $+_\iota$ defined in Section 2.2.3, its typing judgement, as calculated by the interpreter, is illustrated in Listing 3.2. More examples of terms and their typing judgements can be found in Appendix D.

3.3 Term Evaluation

Typically when reducing expressions it is tempting only to consider closed terms, terms with no free variables. However, to define rigorous operational semantics, we must consider all scenarios. It is common to represent the rules for reducing expressions as a series of 'small-step' operational semantics, which provide rules on what to do for a given term. Each step could be applied at any point in the computation and as such there could be free variables to account for. A common way of defining these rules is through the use of evaluation contexts.

It is worth noting that not all calculi require a notion of evaluation contexts to define evaluation. In a more classical example of a simply typed λ-calculus, basic datatypes are cleverly represented through a series of higher-order functions, called Church encodings. In that example, a term's computation can be defined as its β-reduction. It is our more

Listing 3.2: Program output from the typing of $+_l$

```
Type judgement for addLeftTerm = \f:o->o->o => \x:o => \y:o => if0 x then y else (suc
[f]: o->o->o
[x]: o
[y]: o
[pred x]: o
[f (pred x)]: o->o
[f (pred x) y]: o
[succ (f (pred x) y)]: o
[if0 x then y else (succ (f (pred x) y))]: o
[\y:o => if0 x then y else (succ (f (pred x) y))]: o->o
[\x:o => \y:o => if0 x then y else (succ (f (pred x) y))]: o->o->o
[\f:o->o->o => \x:o => \y:o => if0 x then y else (succ (f (pred x) y))]: (o->o->o)->o

Type judgement for add = \x:o => \y:o => (fix addLeftTerm) x y
[x]: o
[y]: o
[fix addLeftTerm]: o->o->o
[(fix addLeftTerm) x]: o->o
[(fix addLeftTerm) x y]: o
[\y:o => (fix addLeftTerm) x y]: o->o
[\x:o => \y:o => (fix addLeftTerm) x y]: o->o->o
```

complex language definition which calls for the need for evaluation contexts when defining its evaluation rules.

Definition 3.4. A *closure* (E, t) is an environment E paired with a term t such that the environment is defined for all free variables in t.

Definition 3.5. An *environment* E maps labels to closures. E *interprets* a label x if there exists a map from x. The result to which E interprets x is written as $E[x]$.

Definition 3.6. A closure *evaluates* $(E, t) \Downarrow v$ to value v if there exists a value v such that $M \to^* v$.

Evaluation contexts are, in essence, an environment and a 'hole' $E[\_]$ which can be filled by any term. When a hole is filled, all the term's free variables take on values from the environment. This naturally describes a way of providing inputs or starting values to a program, as well as equally describing the tracking of an ongoing computation through recording the current values of variables in the environment. This idea of capturing the current state of a computation in a higher kinded type is common in functional programming and is an idiomatic use case for a monad.

3.3.1 The Eval monad

A monad represents computations which can be composed together to form new computations. In practice, they are used to track state, print program logs, or encode the possibility of computational failure. Although monads are a useful abstraction over composable computations, frustratingly they themselves are not composable with one another. To

Listing 3.1: The Eval monad used for term evaluation

```
type Environment = Map.Map Label Term
type Eval a = (ReaderT Environment (ExceptT String (WriterT [String] Identity))) a
```

encode multiple monadic effects for a single computation, one must build what is known as a monad transformer, effectively a monad over other monads.

Our implementation of term evaluation in Haskell revolves around a monad transformer we have defined called Eval. It is composed of predefined monads from the Haskell standard library. As illustrated in Listing 3.1 it uses a 'Reader' monad to keep track of the current environment, an 'Except' monad to indicate that computation could fail as a string, and a 'Writer' monad to keep track of any supplementary logs which are a useful way of recording the order of operations to look at later. An environment is just a map of labels to terms, exactly as it is defined in Definition 3.5. Using this monad, it is possible to define an 'eval' function with the signature below:

```
eval :: Term -> Eval Term
```

This reads as 'eval is a function which maps a term to the Eval of that term'.

Recall that Term (Listing 3.1) is an algebraic data type, meaning that eval must provide a case for each possibility of what that term could be. This is called exhaustive pattern matching and is commonly used alongside monad transformers to build interpreters, (*e.g.*, Liang, Hudak and Jones, 1995). Each matched pattern corresponds to an evaluation rule of the language, these can be found in Section 3.3.2.

3.3.2 Operational semantics

Evaluation rules are defined similarly to typing rules and together form small-step operational semantics.

Constants

$$\frac{}{E[n \in \mathbb{N}] \Downarrow n} \tag{3.16}$$

$$\frac{}{E[error_i] \Downarrow error_i} \quad (i \in \{1, 2\}) \tag{3.17}$$

Variables

$$\frac{E[x] \Downarrow (E', x') \qquad E'[x'] \Downarrow V}{E[x] \Downarrow V} \tag{3.18}$$

Abstraction

$$\frac{}{E[\lambda x.M] \Downarrow E[\lambda x.M]} \tag{3.19}$$

Application

$$\frac{E[M] \Downarrow (E', \lambda x.x') \qquad E[N] \Downarrow V' \qquad E' \cup \{x \mapsto V'\}[x'] \Downarrow V}{E[MN] \Downarrow V} \tag{3.20}$$

Successor

$$\frac{E[M] \Downarrow n}{E[succ\ M] \Downarrow n+1} \tag{3.21}$$

Predecessor

$$\frac{E[M] \Downarrow n+1}{E[pred\ M] \Downarrow n} \tag{3.22}$$

If0

$$\frac{E[P] \Downarrow 0 \qquad E[M] \Downarrow V}{E[\text{if0 } P \text{ then } M \text{ else } N] \Downarrow V} \tag{3.23}$$

$$\frac{E[P] \Downarrow n \qquad n > 0 \qquad E[N] \Downarrow V}{E[\text{if0 } P \text{ then } M \text{ else } N] \Downarrow V} \tag{3.24}$$

Case

$$\frac{E[N] \Downarrow i \qquad E[P] \Downarrow p^n \qquad i < n}{E[\text{case}\langle n, p^n\rangle] \Downarrow \pi_i(p^n)} \tag{3.25}$$

Y combinator

$$\frac{}{E[\text{Y}\lambda x.M] \Downarrow E[M[\text{Y}\lambda x.M/x]]} \tag{3.26}$$

Catch

$$E[\text{catch } M] \Downarrow i \text{ Where i is where M is strict. See Section 3.3.3.} \tag{3.27}$$

3.3.3 Observing sequentiality with catch

An important and helpful insight into a program's behaviour is the order in which its arguments are evaluated. It may not be immediately obvious, but armed with that knowledge, and some other structure we are yet to discuss, it is possible to completely define a procedure's behaviour.

A programmer may determine the evaluation order for a given procedure's arguments by exploiting that functions in SPCF are error-sensitive. That is, by supplying unique error values as arguments and observing which error is thrown first, a programmer may observe the sequential evaluation order of a procedure. For example, consider the left addition operator defined in Section 2.2.3, $+_l\ error1\ error2 = error1$. This is insightful but is limited to the programmer recording the evaluation order between executions.

Cartwright and Felleisen (1992) include an operator called 'catch' in SPCF which allows the observation of evaluation order within a program. It is defined as a family of procedures with types $(\tau_1 \to \ldots \tau_n \to o) \to o$ where if f is a procedure of type $\tau_1 \to \ldots \tau_n \to o$, then catch f returns $i-1$ if f is strict in its i^{th} argument and $k+n$ if f returns a constant. This operator is a form of non-local control, where local evaluation is terminated and control transferred elsewhere in the program.

Implementing non-local control

When evaluating a catch statement, a second non-stateful[1] traversal of the AST is performed from the current term until an exit condition is met. Function constants and applications simply continue the downward traversal of catch, being sure to evaluate their arguments in sequential order. To evaluate λ-abstractions, the binding variable's label is pushed to a control stack before also continuing downward. When a variable is reached, the index of the last occurring instance of the label in the stack is returned. In the case of error constants not passed as arguments, the length of the current control stack is returned. Catch was originally defined in terms of a lexically scoped control stack, which equally describes this implementation.

We discuss further the implications of the existence of catch in Chapter 4, along with how it is used to construct the affinely definable denesting transformation.

This method assumes that terms must be closed because catch does not keep track of the state of other parts of the program. It would be theoretically possible to implement catch without this requirement by having a global control stack which kept track of all binding variables and then offsetting the index returned by catch with the length of the global stack. However, this would have non-obvious implications on the index of unbound variables which would be negative.

3.4 Parser

As part of the project, we have provided a lexer and parser for SPCF to make experimentation with the language easier and to bring it closer to practical programming languages. A lexer and parser make up what is known as a compiler frontend and together they enable writing programs as plain text with some syntax sugar, making programs significantly easier to read and write. The execution pipeline looks like this:

Lexical analysis -> Parse tokens -> Object code -> Execute object code.

3.4.1 Lexical analysis

Lexical analysis, also called tokenisation, describes the decomposition of source code to a stream of tokens. Using an incredibly simple example, the code ‘x = 5;‘ could be decomposed into the tokens ["x", "=", "5", ";"]. We are effectively classifying substrings according to what they do. Patterns for these tokens are defined ahead of time along with the language definition. It is this list of tokens which is used by the parser to construct an AST. The action of *lexing* is conceptually similar to applying a series of nested regular expressions.

Tokenisation is a well-defined problem and general tools exist to make the job easier. Alex[2] is a code generation tool used to build lexers in Haskell. It is configured with a set of constant strings which have a one-to-one correspondence with a token, and regular expressions for more complicated matches. Tokens are defined as entries in an algebraic datatype which can be referenced from other parts of the compilation pipeline. Once

[1] It is non-stateful in the sense that it does not consider the state from the evaluation up to this point.
[2] `https://haskell-alex.readthedocs.io/en/latest/about.html`

Listing 3.1: Minimal Alex configuration to lex assignment operations.

```
-- The top half configures regular expressions to be used when matching tokens
$digit = 0-9
$alpha = [A-Za-z]
tokens :-
  $digit+                       { lex (TokenNat . read) }
  \=                            { lex' TokenEquals }
  \;                            { lex' TokenSemicolon }
  $alpha [$alpha $digit \_ \']* { lex (TokenId) }
{
-- The bottom half configures the ADT tokens will inhabit
-- A token is a pair consisting of the Alex State monad and the TokeClass ADT.
-- This is to improve error messages.
data TokenClass = TokenId String | TokenSemicolon | TokenNat Int | TokenEquals
data Token = Token AlexPosn TokenClass
}
```

configured, Alex will generate Haskell code for a lexer defined on those expressions and bundle it with the interpreter binary. It works similarly to a large macro, abstracting many mundane details which would otherwise have to be considered if handwriting a lexer, while highlighting the important ones such as which tokens exist.

One benefit of Alex being built with Haskell is that it has extraordinary support for functional idioms, such as using a State Monad for tracking token information during tokenisation. This monad keeps track of metadata such as the file name and current character position, which is used to make errors more informative.

Using the previous example along with the minimal Alex configuration in Listing 3.1, it is possible to tokenise the expression x=5; as the list of tokens:

```
[(TokenId "x"), TokenEquals, (TokenNat 5), TokenSemicolon]
```

If the expression was malformed such as x@ = 5;, then the error will include information from the state monad

```
spcf: user error (programs/program.spcf:1:2: lexical error at character '@')
```

3.4.2 Parsing

A parser uses a set of recursive rules to construct the program AST from the stream of tokens generated during lexical analysis. Similarly to tokenisation, parsing is a well-defined problem and tools exist to abstract laborious details. Happy[3] is a code generation tool used to build parsers in Haskell, it is based on the famous parser-generator Yacc, which notably has a syntax reminiscent of BNF grammars. Similarly to Alex, Happy configuration can be viewed as a macro, only slightly more complex. It can be read top-down as a series of rules, where each rule may be defined inductively in terms of other rules.

Using the same example from before and the minimal Happy configuration in Listing 3.2 the expression x=5; could become encoded in the intermediate AST representation as:

[3] https://haskell-happy.readthedocs.io/en/latest/

Listing 3.2: Minimal Happy configuration to parse assignment expressions

```
-- Atomic terms have a one-to-one correspondence with a term in the AST.
-- Map TokenId to AST.Variable
-- Map TokenNat to AST.Numeral
ATerm :
  | natVal { case $1 of Token info (TokenNat n) -> SPCF.AST.Numeral info n }
  | id { case $1 of Token info (TokenId id) -> SPCF.AST.Variable info id }

-- In this case, statements may only bind values to names.
-- Map TokenId to a declaration using what is defined in `Binder`.
Statement : id Binder {
    case $1 of Token info (TokenId id) -> SPCF.Interpreter.Declare info id $2 }

-- Determine what can be on the RHS of declarations
Binder : '=' ATerm { $2 }
```

Listing 3.3: Addition defined in SPCF

```
addLeftTerm = \f:Nat->Nat->Nat =>
              \x:Nat =>
              \y:Nat => if x then y else (succ (f (pred x) y));
add = \x:Nat => \y:Nat => (fix addLeftTerm) x y;
eval (add error1 error2);


Declare "x" (Numeral 5)
```

Similarly to Alex, a State monad is used to capture information from parsing to improve error messages. Notice that the pattern matching cases in Listing 3.2 are not exhaustive, if no suitable rule is matched then a parse error is thrown and the programmer is alerted to where the error is. For example the program x = ; throws the error

```
spcf: user error (programs/program.spcf:1:5: parse error at token ';')
```

As a testament to how powerful Happy and Alex are when used together, both the lexer and parser for the Glasgow Haskell Compiler (GHC) are entirely defined using them [4].

3.5 Correctness Testing

The interpreter is complex and relies on many typing and evaluation rules, creating many opportunities for bugs to shroud themselves in non-obvious ways. This is exasperated by the prevalence of continuations and callbacks in the program logic of the denesting action implemented in Chapter 4, leading to cryptic and misleading errors being thrown far away from where the bug is. This creates the perfect conditions for a kind of 'second-order' logical error, where it is hard to discern whether issues are related to the internals of the interpreter or the program logic built on top of it.

In lieu of a proper debugger for SPCF programs, the interpreter can output logs of intermediate evaluation and type inference steps, approximating a kind of call stack for

[4] https://github.com/ghc/ghc/blob/master/compiler/GHC/Parser/Lexer.x

term manipulations. These logs may be observed and compared against solutions written by hand.

Unit tests help verify that individual components are working as expected, narrowing the potential scope for bugs, or at least explicitly stating erroneous logic as intentional at the time. The project heavily relies on a combination of unit tests and hand evaluation to verify that its behaviour is correct. One of the primary benefits of units tests is that they may be algorithmically regressed upon as more features are added, which is enforced by a GitHub action acting as a simple continuous integration system [5].

A looser guarantee of correctness is also provided by Haskell, which has a strong type system and strict compiler, both of which perform many checks at compile time. For example, integers are guaranteed to be integers and nullary cases are guaranteed to be handled.

3.6 Conclusion

The goal of implementing the SPCF denesting action relies on a large amount of background material and prerequisite tools working together. We develop an interpreter for SPCF to overcome none existing before the project.

Using appropriate techniques, we supply a suitable runtime representation for SPCF and a method for evaluating terms. Building upon this, we use code generation tools to create an interpreter frontend for the language, allowing for easier experimentation and bringing it closer to practical programming languages. We discuss a suitable approach to typing, emphasising how the denesting action requires the preference of type inference over type verification.

Particular focus is given to the catch term as it represents a conceptual problem not commonly present in other λ-calculi, requiring some nuance in its implementation. An overview of the approach of correctness testing is provided with discussions of the kinds of errors in programs of this type. At each stage, points are justified with definitions and examples where necessary, including the choice of programming language, Haskell.

In the next Chapter, we discuss in more detail the denesting action and its implementation.

[5] `https://github.com/jayrabjohns/dissertation-refactoring-spcf/blob/main/.github/workflows/haskell.yml`

Chapter 4

Affinely Definable Denesting Transformation

Up to this point, we have explored SPCF and its predecessors in Section 2.2.3, discussed concrete language definitions in Section 3.1, how they materialise in the form of an interpreter in Section 3.3, and alluded to a refactoring transformation for programs written in SPCF to an equivalent affine form in Section 2.3.

Laird originally split the problem into two cases, the transformation of bounded and unbounded terms. Bounded terms are restricted to numerals smaller than some upper bound n as well as not being recursive. Conversely, unbounded terms may use anything in the language definition given that iteration is used in favour of recursion.

In this chapter, we elaborate on the refactoring action, how it uses SPCF-specific features, and comment on its implementation details. We regularly comment on specific details relevant to bounded and unbounded terms individually. Remember, this transformation aims to construct observationally equivalent terms that adhere to affine typing rules. The two primary components to this are an injection to encode functions as an intermediate 'tuple representation' consisting of $\top$ and $\bot$ elements, and a projection to recover the original type structure.

4.1 Continuation passing style

CPS is a style of programming where all function calls are tail calls. It is possible to have an entry point to the program return the empty type $\underline{0}$ and all computation be through the calling of continuations.

Definition 4.1. A *continuation* is a higher-order function representing the current state of a computation.

Classically, continuations have one argument, the value of the computation so far, and return the final result of the computation when the rest of the program has also finished. They are analogous to callback functions or partial function applications.

In some senses, CPS programming is the natural opposite of classical functional programming. With the latter, all computations are effectively the return values of functions,

whereas with CPS, computations are function side effects captured by running continuations. The transformation relies on the notion of continuation passing as its conceptual base case. It is possible to rewrite any function to an equivalent curried function written in CPS, then this transformation may be applied.

A valid question is what curried functions written in CPS look like, consider an example term with the type $bool \times bool \Rightarrow \underline{0}$

$$\begin{aligned}\lambda x^{bool \times bool}.&\text{if } \pi_0(x)\\ &\text{then (if } \pi_1(x) \text{ then } \bot \text{ else } \bot)\\ &\text{else } \bot\end{aligned}$$

No information can be garnered from the term's return value, it is always the same, but clearly, it may have different runtime behaviours depending on its inputs. For instance, if the first element of x is ff, the second element is never evaluated. This captures the essence of CPS programming.

4.2 Procedures as decision trees

Graphs often encapsulate certain aspects of a program or function's behaviour. These can be at any level of abstraction, such as a high-level data flow visualisation of neural networks, or the low-level control flow of a function's possible execution paths. The inclusion of non-local control and error sensitivity in SPCF means that its graph model for functions has more structure than generic function graphs. Intuitively, it makes sense that the graph model for a sequential language should have to encode some essence of sequentiality, specifically the evaluation order of function arguments. This is exactly the case and one of the key findings of Cartwright and Felleisen (1992), who outline a denotational model for continuous functions in SPCF constructed from decision trees.

Consider the term:

$$\lambda xyz.\ \text{if0 } x \text{ then } (succ\ y) \text{ else } (pred\ z)$$

It is strict in its first argument, and the value of x determines the strictness of its second and third arguments. When x is 0, y is evaluated, and otherwise, z is evaluated. It is possible to entirely represent this function in terms of the index at which it is strict, along with a mapping of all possible values of the strict argument to the return value of the function; this is written as the pair $\langle i, f \rangle$. Functions in SPCF are error sensitive and as such error1 and error2 will always map to error1 and error2 respectively. Since functions are continuous, $\bot$ always maps to $\bot$. Thus the function above can be visualised as the

decision tree in (4.1).

$$\langle 1, \begin{cases} \bot \mapsto \bot \\ \text{error}_1 \mapsto \text{error}_1 \\ \text{error}_2 \mapsto \text{error}_2 \\ 0 \mapsto \langle 2 \begin{cases} \bot & \mapsto \bot \\ \text{error}_1 & \mapsto \text{error}_1 \\ \text{error}_2 & \mapsto \text{error}_2 \\ 0 & \mapsto 1 \\ 1 & \mapsto 2 \\ & \dots \end{cases} \rangle \\ 1 \mapsto \langle 3 \begin{cases} \bot & \mapsto \bot \\ \text{error}_1 & \mapsto \text{error}_1 \\ \text{error}_2 & \mapsto \text{error}_2 \\ 0 & \mapsto \bot \\ 1 & \mapsto 0 \\ 2 & \mapsto 1 \\ & \dots \end{cases} \rangle \\ \dots \end{cases} \rangle \tag{4.1}$$

Vertices of the tree are the set of pairings $\langle i, f \rangle$ and the edges leading from each vertex are the set of possible values for argument i.

Other models for SPCF exist, for example, Kanneganti, Cartwright and Felleisen (1993) provide an alternate model, reconstructed from this one, using the idea of a prime basis (Winskel, 1980) in an attempt to highlight other aspects of SPCF. We focus on this original decision tree model because the mental model of functions being equivalent to some branching structure is exceedingly helpful when reasoning about the denesting action discussed in the next section.

4.3 Procedures as tuples

The basis of the transformation is a pair of functions 'inj' and 'proj'. 'inj' maps a term to a tuple of the index at which it is strict and a collection of continuation functions for each possible applicable value.

Definition 4.2. The insertion of a value x at the position i into a given tuple $t = \langle t_0, \dots, t_{n-1} \rangle$ where $i \leq n$ is denoted $t\lfloor x \rfloor_i$. For example $t\lfloor x \rfloor_i = \langle t_0, \dots, t_{i-1}, x, t_i, \dots, t_{n-1} \rangle$

Definition 4.3. The removal of a value x at the position i from a given tuple $t = \langle t_0, \dots, t_{n-1} \rangle$ where $i \leq n$ is denoted $\lceil t \rceil_i$. For example $\lceil t \rceil_i = \langle t_0, \dots, t_{i-1}, t_{i+1}, \dots, t_{n-1} \rangle$

Injection to tuple form

Definition 4.4. The function *inj*, which maps a term from the bounded SPCF to a so-called 'tuple form', is definable as

inj: $[\![\underline{n}^{m+1} \Rightarrow \underline{0}]\!] \to [\![\underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n]\!]$

$$\text{inj}(f) = \begin{cases} f & \text{if } f \in \{\top, \bot\} \\ \langle i, \langle f'_j \mid j < n \rangle \rangle & \text{if } i = \text{catch } f, \text{ where } f'_i(x) = f(x\lfloor j \rfloor_i) \end{cases}$$

Here $\top$ is interpreted as an unrecoverable error, taking the place of error constants $error_1$ & $error_2$. This is definable as the affinely typeable term:

$$\lambda f.\text{case}\langle \text{catch } f, \langle \langle i, \langle \lambda x.f(\langle \pi_0(x), \ldots, \pi_{m-1}(x) \rangle \lfloor j \rfloor_i) \mid j < n \rangle \rangle \mid i \leq m \rangle \rangle$$

To understand this term it helps to consider its type one component at a time and correspond that to the term definition. f is the only parameter of inj and has the type $\underline{n}^{m+1} \Rightarrow \underline{0}$, meaning f is a function mapping $\underline{n}^{m+1}$, an $m+1$ fold product of numerals $\leq$ n, to the empty type $\underline{0}$. The second half of inj's type is a pair with the first element being $\underline{m+1}$, precisely the type provided by 'catch f' and a tuple containing the same number of candidate continuation functions as there are numerals in the language. Each continuation has the type $\underline{n}^m \Rightarrow \underline{0}$ which can be seen as the same type as f if it took one fewer argument. Looking at the term, it is clear that this is because an argument is applied in the strict position as part of the continuation.

This action effectively represents bounded terms as a series of nested tuples consisting of the index at which a function is strict and a tuple of continuation functions for all the possible values at its strict argument.

This concrete definition is a slight amendment from Laird's original definition. Originally, the roles of i and j are swapped, a subtle change which changes the direction in which the tuples are nested. This appears to be inconsistent with the type structure and descriptions of the term elsewhere in the paper and thus it is assumed to be a typographical error[1].

Projection from tuple form

Definition 4.5. The function *proj*, which transforms a term in tuple form back to its original type, is definable as
proj: $[\![\underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n]\!] \to [\![\underline{n}^{m+1} \Rightarrow \underline{0}]\!]$

$$\text{proj}(\langle i, c \rangle)(x) = \begin{cases} i & \text{if } i \in \{\top, \bot\} \\ \top & \text{if } \pi_i(x) = \top, \text{ or } \pi_i(x) = \top \\ \top & \text{if } f'_i(\lceil x \rceil_i) = \top, \text{ where } f'_i = \pi_j(c) \\ \bot & \text{otherwise} \end{cases}$$

This is definable as the term

$$\lambda tx.\text{case } \langle \pi_0(t), \langle \text{case } \langle \pi_i(x), \langle \pi_j(\pi_1(t)) \lceil \langle \pi_0(x), \ldots, \pi_m(x) \rangle \rceil_j \mid j < n \rangle \rangle \mid i \leq m \rangle \rangle$$

To understand this term, it helps to bear in mind the structure of f when in tuple form. The parameter t is of the form $\langle i, c \rangle$ where i is the index at which f is strict and c is a tuple of continuation functions for each possible value supplied at the strict position. The parameter x is the tuple of f's parameters. In that sense the term defined here is actually a curried function of the type $[\![\underline{m+1} \times (\underline{n}^m \Rightarrow \underline{0})^n \times \underline{n}^{m+1}]\!] \to [\![\underline{0}]\!]$.

[1]This was confirmed through personal communication, May 2024

4.3.1 Implementation details

The definitions of inj and proj are quite dense and require many prerequisite features for their implementation. In Chapter 3.1 we discuss most of these, however, it is worth noting some subtleties which may not be obvious:

1. Care is needed when defining whether function arguments and tuples are 0-indexed or not. Both terms rely on the definition of catch to index a tuple, so their indexing must be consistent. This is another justification for using a stateless catch operation as mentioned in Section 3.3.3.
2. Extra care is required when choosing a representation for tuples. There are two cases in the definitions of inj and proj, the n-fold product and the binary product.
3. The construction of each term depends on the type of their input, thus a suitable method of type inference is required. This influenced the decision to rely on type inference for type checking discussed in Chapter 3.2.

To expand on point (2), recall that the n-fold product is a product with length n where each element is of the same type and that the binary product has no such restriction on typing. This should be clear by the fact that inj returns a product of two elements with different types, a numeral and another product. It follows that it is not possible to define this pairing as an n-fold product and have the program correctly typed.

One solution is to construct all n-fold products from the repeated application of binary products, however, having a representation for n-fold products relying on an underlying list is very convenient and provides many operations out of the box. The alternative is to include definitions for both kinds of product including π_1 and π_2 terms for the binary case. This approach retains the convenient representation for n-fold products and makes it possible to statically determine the types of constituent terms in a binary product.

4.3.2 Affinely definable retraction on types

Definition 4.6. A *retraction* $X \trianglelefteq Y$ is a function $f : Y \to X$ where there exists a function $g : X \to Y$ such that $g \circ f = id_x$.

In other words, elements from X may be recovered from Y after an initial mapping without information loss. It is in some senses a weaker form of bijection, where there is no implication of f and g being invertible.

The pair of functions inj and proj form an affinely typeable retraction (4.2). A full proof can be found in the original literature, but informally there is quite explicitly a retraction from each type to one of a lower order given by Definitions (4.4) and (4.5). By induction on m it is possible to derive (4.3).

$$\underline{n}^{\underline{m}} \Rightarrow \underline{0} \trianglelefteq \underline{m+1} \times (\underline{n}^{\underline{m}} \Rightarrow \underline{0})^{\underline{n}} \quad (\forall n > 0) \tag{4.2}$$

$$\underline{n}^{\underline{m}} \Rightarrow \underline{0} \trianglelefteq \underline{m}^{\underline{n^m \cdot (m+1)}} \quad (\forall n > 0) \tag{4.3}$$

Since inj reduces the order of a type by one, this step must be repeated $m + 1$ times to

reach a base case.

$$\begin{aligned} n^{m+1} \Rightarrow \underline{0} &\to \underline{m+1} \times (n^m \Rightarrow \underline{0})^n && \text{(Once)} \\ &\to \underline{m+1} \times (\underline{m} \times (n^{m-1} \Rightarrow \underline{0})^n)^n && \text{(Twice)} \\ &\to \underline{m+1} \times (\underline{m} \times (\dots \underline{1} \times (n^0 \Rightarrow \underline{0})^n \dots)^n)^n && \text{(m+1 times)} \end{aligned}$$

Regard that the innermost product of continuations in the $m+1$ case has elements of the type $n^0 \Rightarrow \underline{0} \equiv I \Rightarrow \underline{0}$ which is isomorphic to $\underline{0}$ (4.7). Thus a function can be entirely encoded as a product of the form $\underline{m}^{n^m \cdot (m+1)}$.

The original type structure is recoverable from this form with proj.

$$\begin{aligned} \underline{m+1} \times (\underline{m} \times (\dots (n^0 \Rightarrow \underline{0})^n \dots)^n)^n &\to \underline{m+1} \times (m \times (n^{m-1} \Rightarrow \underline{0})^n)^n && \text{(m-1 times)} \\ &\to \underline{m+1} \times (n^m \Rightarrow \underline{0})^n && \text{(m times)} \\ &\to n^{m+1} \Rightarrow \underline{0} && \text{(m+1 times)} \end{aligned}$$

The result is an affinely typable term contextually equivalent to f, with no function nesting. In other words, $(\text{proj} \circ \text{inj})\ f = f$. Seeing as Haskell has an operator for function composition, this is rather satisfyingly representable in Haskell as `(proj . inj) f`.

Implementation notes

The repeated application of these terms requires some nuance. Clearly, their output types for a single step do not match the corresponding input types. The term must first be pattern-matched on to selectively apply the relevant step to the correct subterm.

For inj, recognise that we must match on a subterm of the type $[\![n^m \Rightarrow \underline{0}]\!]$. Considering the concrete definition for inj (4.4) the relevant subterm to reapply to is each element of the n-fold product:

$$\langle \lambda x.f(\langle \pi_0(x), \dots, \pi_{m-1}(x) \rangle \lfloor j \rfloor_i) \mid j < n \rangle$$

This can be acquired by pattern matching on the preceding terms

$$\lambda f.\text{case} \langle \text{catch } f, \langle \langle i, \_\_\_\_ \mid j < n \rangle \rangle \mid i \leq m \rangle \rangle \tag{4.4}$$

Similarly for proj (4.5), we must find subterms with the type $[\![\underline{m+1} \times (n^m \Rightarrow \underline{0})^n]\!]$. In this case, the term must first be traversed to reach the base case for inj (4.5), then traversed again backwards applying proj at each step. The correct subterm to reapply proj to can be found by using the same pattern as for inj (4.4).

$$\underline{1} \times (n^0 \Rightarrow \underline{0})^n \tag{4.5}$$

Generalisation to other types

Definition 4.7. The isomorphism $\underline{0}^m \Rightarrow \underline{n} \cong \underline{m+n}$ is definable as the pair of terms:

$$\lambda f.\text{catch } \lambda k.(\text{case } \langle f \langle \pi_i(k) \mid i < m \rangle, \langle \pi_{m+j}(k) \mid j < n \rangle \rangle \tag{4.6}$$

$$\lambda x.\lambda y.\text{catch } \lambda k.\text{case } \langle x, \langle \pi_0(y), \dots, \pi_{m-1}(y), \pi_0(k), \dots, \pi_{n-1}(k) \rangle \rangle \tag{4.7}$$

Using (4.7) and the fact that if $\tau_1 \trianglelefteq \tau_2$ and $\sigma_1 \trianglelefteq \sigma_2$, then $\tau_1 \times \sigma_1 \trianglelefteq \tau_2 \times \sigma_2$ and $\tau_1 \Rightarrow \sigma_1 \trianglelefteq \tau_2 \Rightarrow \sigma_2$, we may note that for all types τ there exists integers n, m such that $\tau \trianglelefteq \underline{n}^{\underline{m}}$. Hence we may show that for any closed term $M : \tau$, $\exists n, m(\tau \trianglelefteq \underline{n}^{\underline{m}})$ where $[\![\text{inj}(M)]\!]$ is affinely definable as a term $N : \underline{n}^{\underline{m}}$ and $[\![\text{proj}N]\!] = [\![\text{proj}(\text{inj}(M))]\!] = [\![M]\!]$. Thus $M \simeq \text{proj}(N)$.

4.4 Transforming unbounded terms

The unbounded case can be viewed as similar to the bounded case. The idea is to construct a retract $(o \to o) \to \underline{0} \trianglelefteq o \to o$ into a lower order type, and inductively apply it to 'reduce' a term's type until it is a universal type $o \to o$. It can be thought of as an extension of the process described in Section 4.3, mainly differing by its use of natural numbers rather than finite tuples. This is because to adapt the terms from Section 4.3 directly for unbounded terms would require fixed point recursion, which is not affinely typeable.

An alternative point of view is to view the unbounded case as a two-player game. Specifically, seeing as a function may be represented as a decision tree, its traversal may be represented as a game *strategy* $\sigma : (\mathbf{N} \times \mathbf{N})^* \to \mathbf{N}_\perp^\top$, dictating the responses of a player with a finite list of pairs of moves. Put differently, it can represent the application of a function f to its argument $g \in \mathbf{N} \to \mathbf{N}_\perp^\top$ by sequentially choosing a number n and evaluating the second player's response $g(n)$ until the game is stopped by either player choosing $\top$ or $\bot$.

A function $f : (\mathbf{N} \to \mathbf{N}_\perp^\top)$ strategy is definable as $\mathrm{strat}(f)(s) = \mathrm{catch}(\lambda e.f(e[s]))$.

Definition 4.8. *label* is a non-local control operator similar to catch. However, it returns the value of a function's strict argument rather than its index. It is directly definable in terms of catch and is in some senses a weaker version of the *call/cc* function defined by Cartwright and Felleisen. An informal example: label $\lambda k.(kN) = k$.

Definition 4.9. Sequential *composition* $(M^o; N^{o\to\tau}) : \tau$ is identical to term application but with the guarantee to always be affinely typable.

Definition 4.10. *Iteration* is represented by a constant *it*: $(o \to o) \to o \to \underline{0}$ and is representable as the term $\lambda f.Y\lambda g.\lambda x.(f; x)g$.

Definition 4.11. An surjective partial pairing constant *pair*: $o \to o \to o$ is definable as $\mathrm{pair(m)(n)} = 2^m(2n+1)$ with accessors definable such that $fst(2^m(2n+1)) = m$ and $snd(2^m(2n+1)) = n$.

The term $N; (pairM)$ is written as $M * N$ and finite lists of k natural numbers may be represented as the numeral $n_1 * (n_2 * \cdots * (n_k * 0))$.

The surjective pairing is central to this transformation, it quite remarkably uses the fundamental theorem of arithmetic to represent pairs of numbers as another unique number. This, along with *label* and favouring iteration over recursion, is what enables the modelling of all partial recursive functions in ASPCF. This is important because the argument is made that ASPCF can model precisely the same functions described by Turning machines. It is used to replace the explicit product type used in Section 4.3.

Definition 4.12.

$$\begin{aligned}
&\mathrm{seq}_i(k,e) \in ((\mathbf{N}\times\mathbf{N})*)\\
&\mathrm{seq}_0(k,e) = \epsilon\\
&\mathrm{seq}_{n+1}(k,e) = \begin{cases}
\top \text{ if } \mathrm{seq}_n(k,e) = \top\\
\top \text{ if } \mathrm{seq}_n(k,e) = s \text{ and } k(s) = \top\\
\top \text{ if } \mathrm{seq}_n(k,e) = s \text{ and } k(s) = i \text{ and } e(i) = \top\\
s(i,j) \text{ if } \mathrm{seq}_n(k,e) = s \text{ and } k(s) = i \text{ and } e(i) = j\\
\bot \text{ otherwise}
\end{cases}
\end{aligned}$$

Definition 4.13.

$$\mathrm{fun}(k)(e) = \begin{cases}
\top \text{ if } seq_n(k,e) = \top \quad (\exists n)\\
\bot \text{ otherwise}
\end{cases}$$

Where $\mathrm{fun}(\mathrm{strat}(f)) = f$.

Definition 4.14. inj is definable as the term

$$\lambda f.\lambda x.\ \mathrm{label}\lambda k.k((\mathrm{strat}(\lambda y.k((fy)*1))x)*0)$$

Definition 4.15. proj is definable as the term

$$\lambda g.\lambda h.\ \mathrm{label}\lambda k.\mathrm{fun}(\lambda x.\mathrm{If0}(\mathrm{snd}(x))\ \mathrm{then}\ (k\ \mathrm{fst}(x))\ \mathrm{else}\ g\ \mathrm{fst}(x))$$

The internal conditional statement can be seen as an encoding for the same case conditional used in Section 4.3 for this new representation of lists as numerals. Rather than jumping straight to an element of the product with 'case, it partially retrieves elements from the list checking them with an equality operation.

Definition 4.16. Equality $(M = N) : o$ returns 0 if M and N evaluate to the same numeral and 1 otherwise.

Definition 4.17. strat $= \lambda f.\lambda y.$ label $\lambda g.f(\lambda x$ label $\lambda k(\mathrm{sub}^* y))$ where $\mathrm{sub}(g,k,x) = \lambda v.$ If0 v then $k(g\ x)$ else (If0 $(\mathrm{fst}(v) = x)$ then k fst(snd v) else snd v)

4.4.1 Implementation notes

We do not provide a working implementation of the affine transformation for unbounded terms, however, we have some observations which may be of interest:

1. The surjective pairing is sound but rather awkward to implement. It is tempting to use an intermediate representation, for example, two terms which work together to represent a partially applied pair and a fully applied pair. This however raises further oddities during type inference, making a naive implementation more appealing. Rather ironically, it could be easier to construct this term with a low-level language like C, where handling numbers as concatenations of other numbers is more idiomatic than in a strict functional language like Haskell.

2. While *label* is conceptually similar to catch, it is easiest to implement as a separate construct. Operationally speaking, it is easier to consider it as another form of tree traversal than encode it in terms of catch as suggested originally by Felleisen.

3. An interesting point of exploration is the attempt to naively implement the transformation for bounded terms using Haskell's lazy lists, which allow the ergonomic creation and mutation of infinite sequences. This would ignore the point of making the transformation affinely typeable but has the potential to achieve the same results if that property was not required.

4.5 Conclusion

We thoroughly discuss the denesting action, its roots in CPS programming, and various insights from the implementation process. Being the first time this transformation has been concretely implemented, there are many details highlighted which are not obvious from the theoretical nature of its inception, namely, an amendment to the original concrete term definition. Particular focus is given to the repeated application of 'inj' and 'proj' and the order in which subterms must be pattern-matched because this is precisely the kind of detail which is often omitted in high-level discussions while being of vital importance to those concerning low-level implementation.

Less focus is given to the transformation of unbounded terms and indeed is an area for continued exploration. An original goal of the project is stated as providing real-world implementations for the denesting of both bounded and unbounded terms, however, the project scope changed during development. Notes from the partial implementation of unbounded denesting are discussed, including a potential approach. While we do not implement unbounded denesting, we still provide an interpreter capable of evaluating unbounded terms, including constructs such as 'label'.

Chapter 5

Results

Here we briefly discuss the denesting action applied to some terms.

There is a flip-flop correspondence between the size of the term and the size of the type. It is suggested that this transformation may be useful when correctness and simplicity are favoured over program size. We find that this is most likely not the case in its current form because the program size grows to the point where it becomes feasibly impractical for a human to read. For instance, consider the function f (5.1) with 4 arguments ($m = 3$) and only boolean values ($n = 2$). To write out f as source code after applying the transformation, it would require roughly **60 thousand pages**[1] with no linebreaks.

In addition to space concerns, the transformation also requires significant computational resources. The entries marked ? in Tables 5.1, B.1 indicate that the machine running the transformation ran out of available memory after using ∼16GB with 2GB swap. Table 5.1 is a breakdown of term sizes after the affine transformation concerning the number of arguments (m + 1) and the maximum value of the language n. The double exponential increase in term size and execution time is further illustrated in Appendix B. All testing is performed on a simple term of the form (5.1).

$$f = \lambda x^{\underline{n}^{m+1}}.\text{case } \langle \pi_0(x), \bot^n \rangle : [\![\underline{n}^{m+1} \Rightarrow \underline{0}]\!] \tag{5.1}$$

In conclusion, the transformation is unfeasible in terms of improvements to readability but it is yet to be seen whether it has other applications.

Table 5.1: Term size after affine transformation, calculated as the length of the resulting terms string representation. ? indicates no data due to memory constraints.

n	**m = 0**	**m = 1**	**m = 2**	**m = 3**	**m = 4**
1	126	1233	20220	758237	69836016
2	290	7814	507387	83474616	?
3	558	30839	4532418	?	?
4	948	89184	23125371	?	?
5	1478	211085	84408096	?	?

[1] Page count estimated from the character count using `https://charactercounter.com/characters-to-words`

Chapter 6

Conclusions

In this project, we cover the implementation of an interpreter for an extension of the λ-calculus with non-local control and use this to illustrate a transformation which removes function nesting in programs, making them adhere to affine typing rules.

Chapter 2 explores related literature, introduces SPCF, discusses objectives for the project, and plans a suitable approach. There is a noted small volume of literature, making it difficult to critically compare and evaluate sources.

In Chapter 3 we build an interpreter for SPCF, using appropriate techniques, to address the issue of there being none widely available at the time. This enables tangible exploration of the language as well as forms a basis for the denesting action in Chapter 4. Conceptual problems and their solutions are discussed, such as the preference for a type inference algorithm. We apply similar approaches to established solutions where appropriate, namely the code generation tools used for the interpreter frontend. Particular attention is paid to the catch function as it is not commonly found in λ-calculi.

Chapter 4 goes into depth on the denesting action, routinely discussing implementation details and shedding light on aspects of the transformation which are not obvious from its original theoretical discussion. Focus was given to the repeated application of the inj and proj functions. The unbounded case of the transformation is discussed but with a smaller focus. Originally, one of the project's aims was to implement the unbounded case of the transformation via selective application on subterms in parallel, enabling a programmer to refactor parts of their code base in isolation. The scope of the project has since changed and this has been marked as an area for future study. Outlines are made for potential approaches and possible compromises.

Chapter 5 briefly discusses results from the application of the function denesting action on an example term and highlights its feasibility in the form that it is currently in. However, it is unclear if a variant of this transformation could be more applicable to other areas of interest, such as efficient IL code for stack-based machines.

Word Count

The number of words until this point, excluding front matter: 9825.

Bibliography

Cartwright, R. and Felleisen, M., 1992. Observable sequentiality and full abstraction. *Proceedings of the 19th acm sigplan-sigact symposium on principles of programming languages* [Online]. New York, NY, USA: Association for Computing Machinery, POPL '92, p.328–342. Available from: https://doi.org/10.1145/143165.143232.

Church, A., 1936. An unsolvable problem of elementary number theory. *American journal of mathematics* [Online], 58(2), pp.345–363. Available from: http://www.jstor.org/stable/2371045.

Church, A., 1940. A formulation of the simple theory of types. *The journal of symbolic logic* [Online], 5(2), p.56–68. Available from: https://doi.org/10.2307/2266170.

Curry, H., Feys, R., Craig, W., Hindley, J. and Seldin, J., 1958. *Combinatory logic* [Online], Combinatory logic v. 65, pt. 2. North-Holland Publishing Company. Available from: https://books.google.co.uk/books?id=BdoozgEACAAJ.

Curry, H.B., 1930. Grundlagen der kombinatorischen logik. *American journal of mathematics* [Online], 52(3), pp.509–536. Available from: http://www.jstor.org/stable/2370619.

Hindley, R., 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* [Online], 146, pp.29–60. Available from: http://www.jstor.org/stable/1995158.

Jones, J., 2003. Abstract syntax tree implementation idioms. *Pattern languages of program design* [Online]. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003). Available from: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

Kanneganti, R., Cartwright, R. and Felleisen, M., 1993. Spcf: its model, calculus, and computational power. In: J.W. de Bakker, W.P. de Roever and G. Rozenberg, eds. *Semantics: Foundations and applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.318–347.

Laird, J., 2007. On the expressiveness of affine programs with non-local control: the elimination of nesting in spcf. *Fundamenta informaticae*, 77(4), pp.511–531.

Liang, S., Hudak, P. and Jones, M.P., 1995. Monad transformers and modular interpreters. *Acm-sigact symposium on principles of programming languages* [Online]. Available from: https://api.semanticscholar.org/CorpusID:1424753.

Loader, R., 1996. Finitary pcf is not decidable. *Theoretical computer science* [Online],

266(1), pp.341–364. Available from: https://doi.org/https://doi.org/10.1016/S0304-3975(00)00194-8.

Milner, R., 1973. *Models of lcf.* [Online]. Available from: https://api.semanticscholar.org/CorpusID:117118678.

Milner, R., 1977. Fully abstract models of typed λ-calculi. *Theoretical computer science* [Online], 4(1), pp.1–22. Available from: https://doi.org/https://doi.org/10.1016/0304-3975(77)90053-6.

Milner, R., 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* [Online], 17(3), pp.348–375. Available from: https://doi.org/https://doi.org/10.1016/0022-0000(78)90014-4.

Plotkin, G., 1977. Lcf considered as a programming language. *Theoretical computer science* [Online], 5(3), pp.223–255. Available from: https://doi.org/https://doi.org/10.1016/0304-3975(77)90044-5.

Tait, W.W., 1967. Intensional interpretations of functionals of finite type i. *Journal of symbolic logic* [Online], 32(2), p.198–212. Available from: https://doi.org/10.2307/2271658.

Turing, A.M., 1937. Computability and λ-definability. *The journal of symbolic logic* [Online], 2(4), p.153–163. Available from: https://doi.org/10.2307/2268280.

Winskel, G., 1980. *Events in computation* [Online]. Ph.D. thesis. The University of Edinburgh. Available from: https://api.semanticscholar.org/CorpusID:19679577.

Appendix A

Code Repository Link

https://github.com/jayrabjohns/dissertation-refactoring-spcf

Appendix B

Term size growth with transformation

Table B.1: Execution time in seconds of affine transformation, recorded as the user-mode CPU time calculated by the 'time' command part of the GNU Core Utilities package. ? indicates no data due to memory constraints.

n	**m = 0**	**m = 1**	**m = 2**	**m = 3**	**m = 4**
1	0.2	0.3	0.4	1.2	193
2	0.3	0.3	0.6	148	?
3	0.3	0.3	3.7	?	?
4	0.3	0.3	22	?	?
5	0.3	0.3	83	?	?

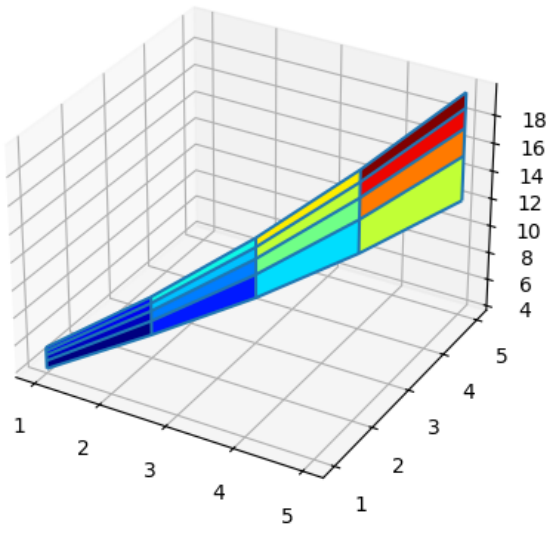
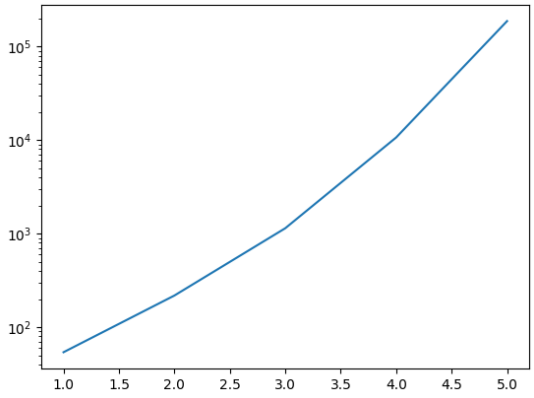
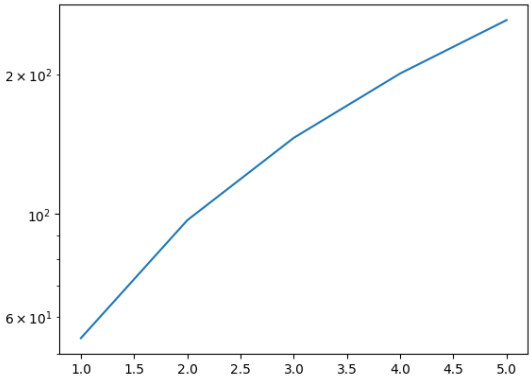


Figure B.1: 3D projection of log term size of inj(M) against n, m. There is an exponential relationship in both directions, creating a curved plane. A colour spectrum is used to highlight that the plane is curved in two directions, whereas redder colours indicate a larger term size.



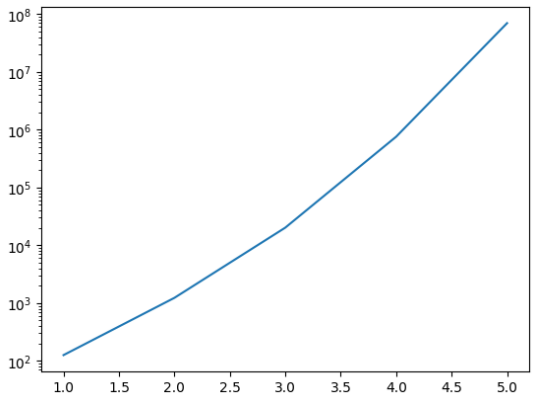
(a) Term size against the number of function parameters, where $n = 1$ is fixed



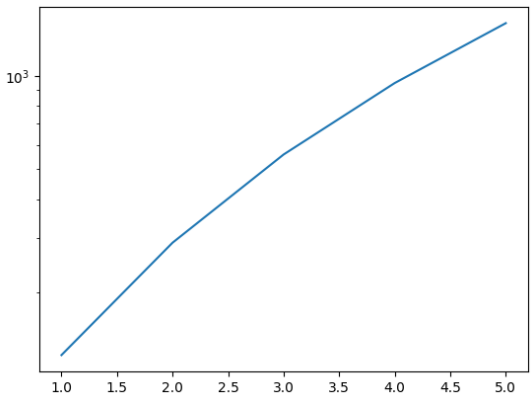
(b) Term size against the upper bound, where $m = 0$ is fixed

Figure B.2: Term size of inj(M)in relation to hyperparameters n, m. Note the log scale. Together these act as slices of the 3D plane in Figure B.1

.



(a) Log term size against the number of function parameters, where $n = 1$ is fixed



(b) Term size against the upper bound, where $m = 0$ is fixed

Figure B.3: Log term size of proj(inj(M)) in relation to hyperparameters n, m.

Appendix C

Example term transformation

Listing C.1: Interpreter logs from applying injection and projection terms. The upper bound on values n = 2, effectively representing the booleans.

```
f = \p:(NatxNat) => Case <Case <1, p>, <⊥, ⊥>>

proj(inj(f)) =
\b:(NatxNat) =>
  Case <
    π1(Case <Catch (\p:(NatxNat) => Case <Case <1, p>, <⊥, ⊥>>),
        <<0, <(\a:Nat => Case <Case <1, <0, (Case <0, a>)>>, <⊥, ⊥
           >>), (\a:Nat => Case <Case <1, <1, (Case <0, a>)>>, <⊥, ⊥
           >>)>>,
        <1, <(\a:Nat => Case <Case <1, <(Case <0, a>), 0>>, <⊥, ⊥
           >>), (\a:Nat => Case <Case <1, <(Case <0, a>), 1>>, <⊥, ⊥
           >>)>>>>
      ),
    <(Case <Case <0, b>,
      <((Case <0, π2(Case <Catch (\p:(NatxNat) => Case <Case <1, p>,
         <⊥, ⊥>>),
          <<0, <(\a:Nat => Case <Case <1, <0, (Case <0, a>)>>, <⊥, ⊥
             >>), (\a:Nat => Case <Case <1, <1, (Case <0, a>)>>,
             <⊥, ⊥>>)>>,
          <1, <(\a:Nat => Case <Case <1, <(Case <0, a>), 0>>, <⊥, ⊥
             >>), (\a:Nat => Case <Case <1, <(Case <0, a>), 1>>,
             <⊥, ⊥>>)>>>>)>
        ) <(Case <1, b>)>
        ),
        ((Case <1, π2(Case <Catch (\p:(NatxNat) => Case <Case <1,
           p>, <⊥, ⊥>>),
          <<0, <(\a:Nat => Case <Case <1, <0, (Case <0, a>)>>, <⊥, ⊥
             >>), (\a:Nat => Case <Case <1, <1, (Case <0, a>)>>,
             <⊥, ⊥>>)>>,
          <1, <(\a:Nat => Case <Case <1, <(Case <0, a>), 0>>, <⊥, ⊥
             >>), (\a:Nat => Case <Case <1, <(Case <0, a>), 1>>,
             <⊥, ⊥>>)>>>>)>
        ) <(Case <1, b>)>
        )>>
```

```
),
(Case <Case <1, b>,
  <((Case <0, π2(Case <Catch (\p:(NatxNat) ⇒ Case <Case <1, p>,
     <⊥, ⊥>>),
      <<0, <(\a:Nat ⇒ Case <Case <1, <0, (Case <0, a>)>>, <⊥, ⊥
         >>), (\a:Nat ⇒ Case <Case <1, <1, (Case <0, a>)>>,
         <⊥, ⊥>>)>>,
      <1, <(\a:Nat ⇒ Case <Case <1, <(Case <0, a>), 0>>, <⊥, ⊥
         >>), (\a:Nat ⇒ Case <Case <1, <(Case <0, a>), 1>>,
         <⊥, ⊥>>)>>>>)>
   ) <(Case <0, b>)>
   ),
   ((Case <1, π2(Case <Catch (\p:(NatxNat) ⇒ Case <Case <1,
      p>, <⊥, ⊥>>),
      <<0, <(\a:Nat ⇒ Case <Case <1, <0, (Case <0, a>)>>, <⊥, ⊥
         >>), (\a:Nat ⇒ Case <Case <1, <1, (Case <0, a>)>>,
         <⊥, ⊥>>)>>,
      <1, <(\a:Nat ⇒ Case <Case <1, <(Case <0, a>), 0>>, <⊥, ⊥
         >>), (\a:Nat ⇒ Case <Case <1, <(Case <0, a>), 1>>,
         <⊥, ⊥>>)>>>>)>
   ) <(Case <0, b>)>
   )>>
)>>
```

Appendix D

Example program execution

D.1 Program Definition

Listing D.1: A program to find the factorial of a number in our implementation of SPCF.

```
addLeftTerm = \f:Nat->Nat->Nat => \x:Nat => \y:Nat => if0 x then y else
↪  (succ (f (pred x) y));
add = \x:Nat => \y:Nat => (fix addLeftTerm) x y;

mulTerm = \f:Nat->Nat->Nat =>
          \x:Nat =>
          \y:Nat =>
            if0 y
              then x
              else (add x (f x (pred y)));

mul = \x: Nat => \y:Nat => (fix mulTerm) x (pred y);

factorial = \f: Nat->Nat => \n:Nat => if0 n then 1 else (mul n (f (pred
↪  n)));
fact = \n: Nat => (fix factorial) n;

eval (fact 3);
```

D.2 Program Type Judgements

Listing D.1: Interpreter logs for typing the program in D.1

```
Type judgement for addLeftTerm = \f:o->o->o => \x:o => \y:o => if0 x then
↪  y else (succ (f (pred x) y))
[f]: o->o->o
[x]: o
[y]: o
[pred x]: o
```

```
[f (pred x)]: o->o
[f (pred x) y]: o
[succ (f (pred x) y)]: o
[if0 x then y else (succ (f (pred x) y))]: o
[\y:o => if0 x then y else (succ (f (pred x) y))]: o->o
[\x:o => \y:o => if0 x then y else (succ (f (pred x) y))]: o->o->o
[\f:o->o->o => \x:o => \y:o => if0 x then y else (succ (f (pred x) y))]:
↪  (o->o->o)->o->o->o

Type judgement for add = \x:o => \y:o => (fix addLeftTerm) x y
[x]: o
[y]: o
[fix addLeftTerm]: o->o->o
[(fix addLeftTerm) x]: o->o
[(fix addLeftTerm) x y]: o
[\y:o => (fix addLeftTerm) x y]: o->o
[\x:o => \y:o => (fix addLeftTerm) x y]: o->o->o

Type judgement for mulTerm = \f:o->o->o => \x:o => \y:o => if0 y then x
↪  else add x (f x (pred y))
[f]: o->o->o
[x]: o
[y]: o
[pred y]: o
[f x]: o->o
[f x (pred y)]: o
[add x]: o->o
[add x (f x (pred y))]: o
[if0 y then x else add x (f x (pred y))]: o
[\y:o => if0 y then x else add x (f x (pred y))]: o->o
[\x:o => \y:o => if0 y then x else add x (f x (pred y))]: o->o->o
[\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x (pred y))]:
↪  (o->o->o)->o->o->o

Type judgement for mul = \x:o => \y:o => (fix mulTerm) x (pred y)
[x]: o
[y]: o
[pred y]: o
[fix mulTerm]: o->o->o
[(fix mulTerm) x]: o->o
[(fix mulTerm) x (pred y)]: o
[\y:o => (fix mulTerm) x (pred y)]: o->o
[\x:o => \y:o => (fix mulTerm) x (pred y)]: o->o->o

Type judgement for factorial = \f:o->o => \n:o => if0 n then 1 else mul n
↪  (f (pred n))
[f]: o->o
[n]: o
```

```
[1]: o
[pred n]: o
[f (pred n)]: o
[mul n]: o->o
[mul n (f (pred n))]: o
[if0 n then 1 else mul n (f (pred n))]: o
[\n:o => if0 n then 1 else mul n (f (pred n))]: o->o
[\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n))]: (o->o)->o->o

Type judgement for fact = \n:o => (fix factorial) n
[n]: o
[fix factorial]: o->o
[(fix factorial) n]: o
[\n:o => (fix factorial) n]: o->o

Type judgement for eval {fact 3}
[3]: o
[fact 3]: o
[Evaluation] -- fact 3
```

D.3 Program Evaluation

Listing D.1: Interpreter logs for the evaluation of the factorial of 3 D.1

```
Evaluating fact with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  +--("fact",\n:o => (fix factorial) n)
+--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--|
   +--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f
   ↪  x (pred y)))

Apply (\n:o => (fix factorial) n) [3/"n"]

Evaluating factorial with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)
```

```
Apply (\a:o => if0 a then 1 else mul a ((fix (\f:o->o => \n:o => if0 n
↪  then 1 else mul n (f (pred n)))) (pred a))) [3/"a"]
if 3
 then 1
 else mul 3 ((fix (\b:o->o => \c:o => if0 c then 1 else mul c (b (pred
 ↪  c)))) (pred 3))

Finding predecessor of 3
Apply (\a:o => if0 a then 1 else mul a ((fix (\b:o->o => \c:o => if0 c
↪  then 1 else mul c (b (pred c)))) (pred a))) [2/"a"]
if 2
 then 1
 else mul 2 ((fix (\d:o->o => \b:o => if0 b then 1 else mul b (d (pred
 ↪  b)))) (pred 2))

Finding predecessor of 2
Apply (\a:o => if0 a then 1 else mul a ((fix (\d:o->o => \b:o => if0 b
↪  then 1 else mul b (d (pred b)))) (pred a))) [1/"a"]
if 1
 then 1
 else mul 1 ((fix (\c:o->o => \d:o => if0 d then 1 else mul d (c (pred
 ↪  d)))) (pred 1))

Finding predecessor of 1
Apply (\a:o => if0 a then 1 else mul a ((fix (\c:o->o => \d:o => if0 d
↪  then 1 else mul d (c (pred d)))) (pred a))) [0/"a"]
if 0
 then 1
 else mul 0 ((fix (\b:o->o => \c:o => if0 c then 1 else mul c (b (pred
 ↪  c)))) (pred 0))

Evaluating mul with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",1)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\x:o => \y:o => (fix mulTerm) x (pred y)) [1/"x"]
Apply (\a:o => (fix mulTerm) 1 (pred a)) [1/"a"]

Finding predecessor of 1
```

```
Evaluating mulTerm with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",1)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\a:o => \b:o => if0 b then a else add a ((fix (\f:o->o->o => \x:o
↪  => \y:o => if0 y then x else add x (f x (pred y)))) a (pred b)))
↪  [1/"a"]
Apply (\c:o => if0 c then 1 else add 1 ((fix (\b:o->o->o => \c:o => \d:o
↪  => if0 d then c else add c (b c (pred d)))) 1 (pred c))) [0/"c"]
if 0
 then 1
 else add 1 ((fix (\a:o->o->o => \c:o => \d:o => if0 d then c else add c
 ↪  (a c (pred d)))) 1 (pred 0))

Evaluating mul with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",2)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\x:o => \y:o => (fix mulTerm) x (pred y)) [2/"x"]
Apply (\a:o => (fix mulTerm) 2 (pred a)) [1/"a"]

Finding predecessor of 1

Evaluating mulTerm with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",1)
```

```
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\a:o => \b:o => if0 b then a else add a ((fix (\f:o->o->o => \x:o
↪  => \y:o => if0 y then x else add x (f x (pred y)))) a (pred b)))
↪  [2/"a"]
Apply (\c:o => if0 c then 2 else add 2 ((fix (\b:o->o->o => \c:o => \d:o
↪  => if0 d then c else add c (b c (pred d)))) 2 (pred c))) [0/"c"]
if 0
 then 2
 else add 2 ((fix (\a:o->o->o => \c:o => \d:o => if0 d then c else add c
 ↪  (a c (pred d)))) 2 (pred 0))

Evaluating mul with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",3)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\x:o => \y:o => (fix mulTerm) x (pred y)) [3/"x"]
Apply (\a:o => (fix mulTerm) 3 (pred a)) [2/"a"]

Finding predecessor of 2

Evaluating mulTerm with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",2)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)
```

```
Apply (\a:o => \b:o => if0 b then a else add a ((fix (\f:o->o->o => \x:o
↪  => \y:o => if0 y then x else add x (f x (pred y)))) a (pred b)))
↪  [3/"a"]
Apply (\c:o => if0 c then 3 else add 3 ((fix (\b:o->o->o => \c:o => \d:o
↪  => if0 d then c else add c (b c (pred d)))) 3 (pred c))) [1/"c"]
if 1
 then 3
 else add 3 ((fix (\a:o->o->o => \c:o => \d:o => if0 d then c else add c
 ↪  (a c (pred d)))) 3 (pred 1))

Finding predecessor of 1
Apply (\b:o => \e:o => if0 e then b else add b ((fix (\a:o->o->o => \c:o
↪  => \d:o => if0 d then c else add c (a c (pred d)))) b (pred e)))
↪  [3/"b"]
Apply (\f:o => if0 f then 3 else add 3 ((fix (\e:o->o->o => \a:o => \c:o
↪  => if0 c then a else add a (e a (pred c)))) 3 (pred f))) [0/"f"]
if 0
 then 3
 else add 3 ((fix (\b:o->o->o => \d:o => \a:o => if0 a then d else add d
 ↪  (b d (pred a)))) 3 (pred 0))

Evaluating add with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",2)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
|     +--("c",1)
|     +--|
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\x:o => \y:o => (fix addLeftTerm) x y) [3/"x"]
Apply (\a:o => (fix addLeftTerm) 3 a) [3/"a"]

Evaluating addLeftTerm with environement:
("factorial",\f:o->o => \n:o => if0 n then 1 else mul n (f (pred n)))
+--("addLeftTerm",\f:o->o->o => \x:o => \y:o => if0 x then y else (succ
↪  (f (pred x) y)))
|  +--("add",\x:o => \y:o => (fix addLeftTerm) x y)
|  |  +--("a",3)
|  |  +--|
|  +--("fact",\n:o => (fix factorial) n)
|     +--("c",1)
```

```
|     +--|
+--("mulTerm",\f:o->o->o => \x:o => \y:o => if0 y then x else add x (f x
↪  (pred y)))
   +--("mul",\x:o => \y:o => (fix mulTerm) x (pred y))
   +--("n",3)

Apply (\a:o => \b:o => if0 a then b else (succ ((fix (\f:o->o->o => \x:o
↪  => \y:o => if0 x then y else (succ (f (pred x) y)))) (pred a) b)))
↪  [3/"a"]
Apply (\c:o => if0 3 then c else (succ ((fix (\b:o->o->o => \c:o => \d:o
↪  => if0 c then d else (succ (b (pred c) d)))) (pred 3) c))) [3/"c"]
if 3
 then 3
 else succ ((fix (\a:o->o->o => \c:o => \d:o => if0 c then d else (succ
 ↪  (a (pred c) d)))) (pred 3) 3)

Finding successor of (fix (\a:o->o->o => \c:o => \d:o => if0 c then d
↪  else (succ (a (pred c) d)))) (pred 3) 3

Finding predecessor of 3
Apply (\b:o => \e:o => if0 b then e else (succ ((fix (\a:o->o->o => \c:o
↪  => \d:o => if0 c then d else (succ (a (pred c) d)))) (pred b) e)))
↪  [2/"b"]
Apply (\f:o => if0 2 then f else (succ ((fix (\e:o->o->o => \a:o => \c:o
↪  => if0 a then c else (succ (e (pred a) c)))) (pred 2) f))) [3/"f"]
if 2
 then 3
 else succ ((fix (\b:o->o->o => \d:o => \a:o => if0 d then a else (succ
 ↪  (b (pred d) a)))) (pred 2) 3)

Finding successor of (fix (\b:o->o->o => \d:o => \a:o => if0 d then a
↪  else (succ (b (pred d) a)))) (pred 2) 3

Finding predecessor of 2
Apply (\c:o => \e:o => if0 c then e else (succ ((fix (\b:o->o->o => \d:o
↪  => \a:o => if0 d then a else (succ (b (pred d) a)))) (pred c) e)))
↪  [1/"c"]
Apply (\f:o => if0 1 then f else (succ ((fix (\e:o->o->o => \b:o => \d:o
↪  => if0 b then d else (succ (e (pred b) d)))) (pred 1) f))) [3/"f"]
if 1
 then 3
 else succ ((fix (\a:o->o->o => \c:o => \b:o => if0 c then b else (succ
 ↪  (a (pred c) b)))) (pred 1) 3)

Finding successor of (fix (\a:o->o->o => \c:o => \b:o => if0 c then b
↪  else (succ (a (pred c) b)))) (pred 1) 3

Finding predecessor of 1
```

```
Apply (\d:o => \e:o => if0 d then e else (succ ((fix (\a:o->o->o => \c:o
↪  => \b:o => if0 c then b else (succ (a (pred c) b)))) (pred d) e)))
↪  [0/"d"]
Apply (\f:o => if0 0 then f else (succ ((fix (\e:o->o->o => \a:o => \c:o
↪  => if0 a then c else (succ (e (pred a) c)))) (pred 0) f))) [3/"f"]
if 0
 then 3
 else succ ((fix (\b:o->o->o => \d:o => \a:o => if0 d then a else (succ
 ↪  (b (pred d) a)))) (pred 0) 3)
"6"
```