# Theory and Applications of Induction Recursion

# Case for Support

## Part 1A: Previous Research and Track Record - Prof. Neil Ghani

See also `http://www.cs.nott.ac.uk/~nxg`

Prof. Ghani obtained his PhD in Computer Science from the University of Edinburgh in 1995 where he worked on categorical models of rewriting. Subsequently he received grants from the Royal Society of London and the EU-funded EUROFOCS program to conduct post-doctoral research at the Ecole Normale Superieure in Paris. In January 1997, he became a Research Fellow at the University of Birmingham and then in September 1998 he became a Lecturer in the Department of Mathematics and Computer Science at the University of Leicester. In September 2005, he moved to the University of Nottingham where he became a Reader in January 2007. On July 1 2008, he was offered a professorship at the University of Strathclyde and asked to form the Mathematically Structured Programming research group there.

**Research Summary:** Prof. Ghani's research tries to understand the nature and structure of computation. This is quite a bold statement and inevitably, only partial answers will be forthcoming. Nevertheless, this statement shows his commitment to ask deep and fundamental questions so as to produce research which is of the highest calibre and which will stand the test of time rather than become obsolete within a few years. In particular, he has worked extensively in the following areas which form the pillars upon which this proposal is built.

- **Category Theory:** Category theory is a relatively new mathematical discipline which provides an abstract theory of structure and hence is key to Prof. Ghani's work. He has applied various categorical structures such as monads, comonads, coalgebras, enriched categories and Kan extensions to problems in computation. Of closest relevance to this proposal is his work on containers which provides a theory of concrete data types.
- **Type Theory:** Prof. Ghani uses type theory as an intermediate abstraction between functional programming and its categorical underpinnings. He has worked on features such as type systems, pattern matching and explicit substitutions which make the lambda-calculus closer to "real" functional languages. He also developed the subject of eta-expansions and showed it to be better behaved than the more traditional theory of eta-contractions. He solved the long standing open problem of the decidability of beta-eta-equality for sum types which had attracted the attention of a number of research groups across the world.
- **Functional Programming:** Humans are good at high level, abstract thinking and programming languages should reflect that so as to make it easier for humans to program. Functional languages offer that possibility so Prof. Ghani has been active in their development. Of clear relevance to this project is his work on initial algebra semantics for advanced data types such as nested types and GADTs. He has also worked on short cut fusion and the problem of how to compose monads.

**Publications:** Prof. Ghani has written over 45 papers which have been published in internationally refereed conferences and journals. These can be found from the above url. Especially relevant are papers on initial algebra semantics [GJ07, GJ08] and containers [AAG03, AGA05, AAGM05, AAG04, MAG07].

**Grants:** Prof. Ghani has been awarded the following research grants: i) Theory and Applications of Containers. EPSRC 2005-2008. Principal Investigator. Grant Value: 230,000 pounds. Code EP/C511964/1; ii) Midlands Graduate School. EPSRC 2004-2006. Principal Investigator. Grant Value: 12,000 pounds. Code GR/T06087/01; iii) Coalgebra and Recursion. The Royal Society of London, 2003-2005. Sole Investigator. Grant Value: 6,000 pounds; iv) Kan - A Categorical approach to Computer Algebra. EPSRC 2001 - 2004. Sole Investigator. Grant Value: 130,000 pounds. Code GR/R29604/01. End of grant IGR assessment: Tending to Outstanding; v) Categorical Rewriting: Monads and Modularity. EPSRC 2000 - 2002. Sole Investigator. Grant Value: 52,000 pounds. Code GR/M96230/01. End of grant IGR assessment: Outstanding; and vi) Eta-Expansions in Dependent Type Theory. EUROFOCS 1996. Sole Investigator. Grant Value: 10,000 pounds.

**Esteem Indicators:** Prof. Ghani is a current member of the EPSRC college charged with assessing the quality of grant applications. He was the director of the Midlands Graduate School in the Foundations of Computer Science. He is also on the steering committee for the British Colloquium on Theoretical Computer Science. He has succesfully supervised 2 PhD students, has two more in their third year and has supervised 2 RAs. He has been the external examiner for 2 PhD students. This year, he has been on the PC's of PPDP, CMCS and RTA.

## Part 1B: Previous Research Track Record - Dr Anton Setzer

See `http://www.cs.swan.ac.uk/~csetzer/`.

Anton Setzer got his first degree (Dipl.math, with distinction) and his PhD (magna cum laude) from the University of Munich. He received a docentship (Swedish habilitation) in Mathematical Logic from Uppsala University. He has been a lecturer at Swansea University since 2000 and was promoted to Reader in 2007.

**Research Contributions:** Some of his major research achievements which are related to this project are: i) the determination (in his thesis) of the precise proof theoretic strength of Martin-Löf Type Theory with W-type and one universe - at that time a 25 year long open problem; ii) the development of the Mahlo universe, which was at that time the proof theoretically strongest extension of Martin-Löf Type Theory, and the determination of its precise proof theoretic strength; iii) the development of a closed formalisation of inductive-recursive definitions with Peter Dybjer; and iv) the use of induction recursion to develop a theory of partial recursive functions in dependent type theory.

**Grants:** He was awarded the following grants: i) Proof-theoretically strong extensions of Martin-Löf Type Theory and applications (sole investigator, 4 K) from the Nuffield Foundation, Dec 2001 - Dec 2003; ii) Extensions of dependent type theory – induction, interaction, universes (PI, 123 K) from EPSRC, Jan 2003 - Nov 2006; and iii) subsite leader in Swansea of the EU project TYPES for proofs and programs (7 K).

**Publications and Esteem:** Details of Dr Setzer's publications can be found from the above url. Especially relevant are papers on induction recursion [DS99, Dyb00, DS03, DS06]

He has been invited to the conference "Mathematical Logic" in Oberwolfach 1995, 1998, 2002, and 2005, the Dagstuhl meetings "Dependent type theory meets practical programming" and "Proof Theory in Computer Science" 2001. He was plenary speaker at the Logic Colloquium 1998 in Prague and at the British Logic Colloquium 2004 in Leeds, special session speaker at the Logic Colloquium 1997 in Leeds, and the Logic Colloquium 2004 in Torino. He was invited speaker at JIG4 in Kobe, Japan 2000, at the conference 100 years of intuitionism in Cerisy, France, 2007. He is an editor of the journal Logical Methods in Computer Science (LMPS). He has successfully supervised one PhD student and another is currently in their second year. He has been external examiner for one PhD student (Leeds) and member of the committee for another one (Gothenburg, Sweden).

## Part 1C: Previous Research Track Record - Dr Thorsten Altenkirch

See also `http://www.cs.nott.ac.uk/~txa`

Thorsten Altenkirch is a Reader at the University of Nottingham and co-chair (with Graham Hutton) of the recently founded Functional Programming Laboratory, currently comprising four academics, three researchers and ten PhD students. His main research interests are Type Theory, Functional Programming, Categorical Methods and Quantum Computing.

**Research Contributions:** In his PhD work he developed a normalisation proof for the Calculus of Constructions with large eliminations and presented a complete formalisation of Girard's Normalisation proof for System F. Jointly with Streicher and Hofmann he developed a categorical account of Normalisation by evaluation and extended this to impredicative theories. Later in joint work with Hofmann, P.Scott and Dybjer he used this approach to show the decidability of typed $\lambda$ calculus with copoducts. His work on monadic representations of typed $\lambda$ calculi (jointly with Reus) is also frequently cited. He developed a decidable variant of extensional Type Theory which is the base of recent joint work with McBride et al on *Observational Type Theory*. Jointly with Abbott, Ghani and McBride he developed and *Container Theory* and applied this to problems in generic programming.

**Grants:** He was awarded four grants: i) Modelling Irreversible Quantum Computation (EPSRC, GR/S30818/01, PI 60k); ii) Midlands Graduate School in the Foundations of Computer Science, (EPSRC GR/T06087/01, CoI, 12k), iii) Observational Equality For Dependently Typed Programming (EPSRC, EP/C512022/1, PI 250k); and iv) Theory and Applications Of Containers (EPSRC grant EP/C511964/1, CoI 230k).

**Publications and Esteem:** Dr Altenkirch has over 40 publications, details of which can be found at the above url. Especially relevant to this project are his work on containers [AAG03, AGA05, AAGM05, AAG04, MAG07] and generic programming [MAM06, AM03, AMM07] He has organized the annual TYPES meetings in 1998 and 2007 (and edited the proceedings) and specialized workshops on Dependently Typed Programming, in 2004 (Dagstuhl seminar 04381) and in 2008 in Nottingham. He has served on several PCs e.g. *Computability in Europe 2008* and *Typed Lambda Calculus and Applications 2006* and is co-chairing the PC for *Programming Languages meet Program Verification*. He has supervised 3 PhD students and 1 RA. He is a current member of the EPSRC college charged with assessing the quality of grant applications.

# Part 2: Proposed Research and its Context

## 2.1 Introduction

**What is Induction Recursion:** Recursion is one of the most fundamental concepts in computation. Its importance lies in the ability it gives us to define computational agents in terms of themselves - these could be recursive programs, recursive data types, recursive algorithms or any of a myriad of other structures. The original treatments of recursion go back to the 1930s where the concept of computability was formalised via the theory of general recursive functions. It is virtually impossible to overestimate how recursion has contributed to our ability to compute and to understand the process of computation.

Is it possible that there is anything fundamental left to say about recursion? We believe there is. Our central insight is this: when defining a function $f : A \to B$ recursively, $A$ is usually fixed in advance. But what if it is not? What if, as we build up the function $f$ recursively, we also build up the type $A$ inductively? The study of functions defined in this way is called *induction recursion* and this proposal aims to develop the theory and applications of induction recursion.

**The Project:** Although Dybjer and Setzer's type-theoretic foundation for induction recursion [DS99, Dyb00, DS03, DS06] was a fundamental innovation, induction recursion has yet to become widely used. We believe this is because i) the conceptual and theoretical foundations of induction recursion need further development; and ii) more examples of induction recursion are needed to illustrate its use to a broader audience. Therefore we intend to pursue a twin-tracked research programme based upon a synthesis of theoretical and applied research:

– **Theoretical Foundations:** We will develop the syntax and semantics of induction recursion by i) extending Dybjer and Setzer's type theoretic foundations for induction recursion; and ii) developing a complementary categorical semantics for induction recursion. This will provide complementary algebraic and logical techniques to address the fundamental and inter-related questions of i) what can be done with induction recursion; and ii) how it can be done in as clean and cogent a form as possible.

– **Practical Applications:** We will demonstrate the potential for induction recursion to make a fundamental impact on programming language research by developing a number of naturally occurring applications. Principally, these applications will be in the areas of advanced data types, generic programming and program language design. These are ideal application areas as induction recursion not only solves problems that researchers currently study, but also suggests new research directions within these areas.

**Calibre and Ambition:** The foundational nature of induction recursion, and the consequent potential for applications in a variety of different fields, attest to the quality and calibre of this project. Our ambition is demonstrated by our central belief that induction recursion can join other abstractions such as dependent types, monadic programming and initial algebra semantics in becoming a mainstream technique within the programming languages community.

## 2.2 Background Scientific and Technological Relevance:

**Type Theory:** Induction recursion can be traced back to the idea of a universe as introduced in Martin-Löf type theory [ML84]. A universe is a type $U$ of names for sets (i.e. 'small' types), together with a function $T : U \to \mathsf{Set}$ that maps a name to the set it denotes.[1]. Since these names may depend on data, i.e. elements of a given set in the universe, the type $U$ must be defined *simultaneously* with the function $T$. Subsequently, various universe principles have been studied. In particular Palmgren [Pal91, Pal98], Rathjen [Rat00, Rat01, RGP98] and Setzer [Set00, Set08a, Set08b] have studied the super-universe operator and related principles.

**Induction Recursion:** Around 1995, Peter Dybjer began a program of formulating explicitly a scheme of definition for universes. This led Setzer and Dybjer [DS99, Dyb00, DS01, DS03, DS06] to give a closed-form syntax for induction recursion based upon three combinators. They also identified two interpretations of this syntax: a reflective interpretation whereby semantic structure is internalised within a universe; and an initial algebra interpretation whereby inductive recursive definitions arise as fixed points of functors of a certain type. However, while Setzer and Dybjer's combinators cover existing interesting examples, it is unknown whether they are sufficiently expressive to capture 'all' inductive recursive definitions. Even to frame this question, one needs an independent mathematical characterisation of induction recursion - such a characterisation is currently missing. Worse, we don't yet have a normalisation proof which is essential if induction recursion is to be used as a foundational principle underlying a programming language. Finally, it is unclear how the reflective and initial algebra interpretation of induction recursion relate to each other. These fundamental questions need answers if induction-recursion is to fulfil its potential.

[1] As usual, we denote by $\mathsf{Set}$ the collection of 'small' types

**Category Theory:** Initial algebra semantics is one of the cornerstones of theoretical computer science as it gives a clean and language-independent means for comparing and evaluating different schemes for defining data types. The desire of programmers to use more sophisticated data types has led to several extensions of initial algebra semantics, e.g. Joyal's theory of species [Joy87], Fiore's generalised species [Fio05], the dependent polynomials of Hyland and Gambino [GH04], and the theory of containers and indexed containers [AAGM05, AGA05, MA08] developed by the applicants. All of these theories aim to understand how one can define set-valued functions $I \rightarrow \mathsf{Set}$ as initial algebras of functors $(I \rightarrow \mathsf{Set}) \rightarrow I \rightarrow \mathsf{Set}$ - significantly the type $I$ is fixed in advance. Induction recursion, which allows the type $I$ to be defined simultaneously with the function $I \rightarrow \mathsf{Set}$, is not constrained by this limitation. Extending initial algebra semantics to cover inductive-recursive structures thus extends the benefits of initial algebra semantics to a much wider class of types.

**Programming:** Abstraction is vital not only in mathematics - it is essential in programming where identifying common structure is needed to ensure code is clear, clean and concise. Programmers already take for granted the ability to define their own data types, to use monads to structure programs and forms of dependent types (as they occur in functional programming and in template Metaprogramming) to more accurately describe computational entities. They are now beginning to need the ability to define their own universes. We intend to build upon this convergence of ideas between theoreticians and programmers by demonstrating what universe technology is available to them, what can be done with it, and introducing them to universe operators, super universes and other constructions they are currently unaware of. And, with advances in dependently typed programming languages, we can give programmers code to experiment and play with. This is key to ensuring that induction recursion breaks out of theory and becomes accessible to programmers.

## 2.3 Relevance to Beneficiaries, Dissemination, Management and Planning etc

**Relevance to Beneficiaries:** This, perhaps more than many projects, is an ambitious project which has the potential to have a significant impact on a large number of researchers. Theoretical computer scientists, e.g. category theorists, type theorists and logicians will be interested in the fundamental nature of induction recursion. Further, programmers will be interested because of the critical mass of programming language examples of induction recursion that seem to be appearing — they will particularly appreciate both the applications we develop and the code we will provide. The potential impact of this research can also be assessed by it's adventure, timeliness and novelty which we now discuss.

*Adventure:* In order to ensure that induction recursion becomes recognised as a fundamental programming device, following other mathematical abstractions such as monads, dependent types, and initial algebra semantics, we have designed an adventurous proposal which concentrates on establishing fundamental results which are likely to stand the test of time, and on exploiting these results by applying them to current problems in programming languages. This is evidenced by the scope of the project which ranges from fundamental but theoretical questions to the production of applications, associated code and the incorporation of induction recursion into the programming language Epigram.

This is clearly not incremental research. Prof Ghani and Dr Altenkirch have not worked on induction recursion before and their new perspective will stimulate new thinking about, and applications of, induction recursion. Further, previous research on induction recursion has tended to remain within theoretical domain. The central thrust of this proposal - to turn induction recursion into a widely used programming technique - is thus a new direction for induction recursion.

*Timeliness:* This is an excellent moment to undertake this research. The categorical study of data types (species, generalised species, dependent polynomials, indexed containers) has advanced to the stage where the tools are now in place to tackle induction recursion. Perhaps even more fundamentally, dependently typed programming languages in the shape of Epigram and Agda have advanced to the stage where our ideas can be implemented in code and hence the benefits of induction recursion can be made directly available to programmers as code, i.e. in a form they understand.

*Novelty:* This is novel research. We intend not to confine ourselves to either the theoretical domain or the programming domain. Rather, for us, there is a symbiotic relationship between mathematics, programming, and the design of programming languages, and any attempt to sever this connection will diminish each component. At the centre of this relationship is the desire to use mathematics to understand the nature of computation, and then to reflect that understanding in programming languages. Hence our proposal is novel in taking cutting edge mathematics and turning it straight into cutting edge programming techniques.

**Dissemination:** We have a track record of submitting papers to internationally leading journals and conferences and shall use them as the principal vehicle for dissemination of our results. We are also active participants in more informal meetings such as the *British Colloquium on Theoretical Computer Science, Midlands Graduate School, Scottish Programming Languages Seminar*, and *Fun in the Afternoon*. A web-site will detail the state of the project. We are active members of relevant email lists and will also use them to disseminate our results. We will collaborate extensively which will further aid dissemination. Finally, dissemination will be enhanced by our planned twice yearly project meetings (see below) which will be open to other researchers to attend and contribute to.

**Management, Planning and Coordination:** We have carefully planned this project. Firstly, the core of the project (**ST 1-6**) will be undertaken by Prof Ghani and Dr Hancock with regular feedback from Drs Setzer and Dr Altenkirch. Nevertheless, designating primary responsibility for the research to lie with Prof Ghani and Dr Hancock will help ensure timely completion of the research. In addition, to ensure that Dr Setzer and Dr Altenkirch can contribute most effectively to the project, work packages constituting a PhD student have been designed to match their strengths - Dr Setzer's will focus on the theory of data types (**SW 4-6**) while Dr Altenkirch's will focus on generic programming (**NT 1-3**). Dr Setzer will also undertake advanced research on the foundations of induction recursion (**SW 1-3**) and Dr Altenkirch will investigate the use of induction recursion as a foundational principle for a programming language (**NT 4**). Our planning is also reflected in the specific reasons we give for each of the objectives (**ST 1-6, SW 1-6, NT 1-4**). Finally, while many of the objectives will influence others, none are prerequisites. Hence lack of progress on one objective will not prohibit progress on other objectives, eg without a complete syntax, we can still use Dybjer and Setzer's IR codes.

*Coordination:* During the first three years, we have planned twice yearly intensive meetings rotating between the sites. These meetings will be open so that interested academics may attend giving further opportunities for input, feedback and dissemination. During that period, we also plan one individual visit between sites per person per year for more focussed work on specific topics. A wiki and blog will be used for interaction between participants and to create a record of progress. In addition, Prof Dybjer, one of the inventors of induction recursion, has agreed to provide yearly assesment of the project (based upon a written report of our progress) and suggestions of how to take the research forward.

*Feasability:* We are the right people to execute this research. Dr Setzer is one of the inventors of induction recursion while all three investigators and Dr Peter Hancock (the named RA) are internationally known for their research in category theory, type theory and programming languages (Haskell, Agda and Epigram). Dr Hancock has already worked with Prof Ghani, and Drs Altenkirch and Setzer. Further, the Universities of Strathclyde, Nottingham and Swansea all have research groups dedicated to understanding the mathematical foundations of computation so as to develop programming languages of the future. Our contacts (detailed under dissemination) further enhance the feasability of this project.

## 2.4 Programme and Methodology

An inductive recursive definition is the definition of a function $T : U \to D$ where i) the set $U$ is defined simultaneously with the function $T$; and ii) $D$ is often $\mathsf{Set}$ but can be more general and cover sets with extra structure, families of sets with extra structure, or even functors with extra structure. Further, $T$ often occurs negatively in the definition of $U$, so there is a delicate use of contravariance in inductive recursive definitions. Dybjer and Setzer provide a grammar for generating a type $OP_D$ of what are called IR codes which represent inductive recursive definitions. We will fulfill the potential of induction recursion to become a mainstream programming technique as follows:

**Theoretical Research**

**1. Semantics of Induction Recursion:** The semantics of induction recursion is not yet well understood. This is unfortunate because problems that are more amenable to the sort of algebraic reasoning that arises from the semantics of type theory remain difficult to tackle when one has only type theoretic tools at one's disposal. We will therefore develop a categorical semantics for induction recursion to provide algebraic techniques that complement the type theoretic and logical ones we already have. Concretely, we will proceed with the following objectives:

- **ST1: An Independent Characterisation of Induction Recursion:** Let us define a set-indexed family of elements of $X$ by $\mathsf{Fam}X = \Sigma I : \mathsf{Set}\,.I \to X$. We propose to interpret an IR code as an endofunctor mapping certain families to families. We seek categorical properties which characterise those functors arising as the interpretation of an IR code. This is analogous to the characterisation of normal functors as those which preserve (wide) pullbacks and filtered colimits [Joy87, Has02]. Crucially, this objective will give a clear and unambiguous meaning to induction-recursion, provide semantic tools to reason about induction recursion and will establish that induction recursion is a fundamental concept, rather than simply a syntactic pattern for capturing a number of interesting examples.
- **ST2: Initial Algebras:** One idea of Dybjer and Setzer is that the essence of induction recursion is the ability to form initial algebras of the functors generated by their grammar. Thus we can understand induction recursion by asking what categorical structure guarantees such fixed points exist. While families can be represented by objects in the arrow category $\mathsf{Set}^{\to}$, we have observed that the delicate treatment of contravariance in induction recursion means we must work with the subcategory whose morphisms are pullback squares. Thus, we intend to give an initial algebra treatment of induction recursion based upon finding colimits of chains of pullbacks. More generally, this amounts to constructing initial algebras for functors defined over the Cartesian morphisms of a fibration. This objective allows us to port the well-known benefits of initial algebra semantics to induction recursion.
- **ST3: Reflection:** Universe types reflect, or internalise, certain closure properties of the external universe. Induction recursion itself is a closure property, and Setzer has defined a universe-type that itself reflects this closure property, dubbed a Mahlo universe because of connections with a hierarchy of ordinals introduced by the logician Paul Mahlo in 1908. Going further, Setzer has introduced universes expressing even stronger forms of

reflection. But what exactly is reflection? This is a significant and, as far as we know, unexplored question which we intend to study. This objective will shed light on the thorny problem of defining elimination rules for universes and, more generally, result in new techniques for crafting universes for computational applications.

**2. The Syntax of Induction Recursion:** We will provide theorems that justify Dybjer and Setzer's grammar.

- **ST4: Completeness of the Syntax:** We conjecture that Dybjer and Setzer's grammar is in fact complete. We will prove this by showing that every functor satisfying the semantic conditions outlined above is generated by an IR code. We also have an alternative approach based upon our following observation: In Cartmell's [Car86] early work on generalised algebraic theories the declarations in a signature take one of three forms: constants, type valued functions and functions whose values are elements of types. It can be no coincidence that these three forms of declarations correspond exactly to the three combinators of Dybjer and Setzer! This objective guarantees that the syntax of induction recursion is robust enough to allow semantic constructions to be reflected within it.
- **ST5: Using Induction Recursion:** To make induction recursion usable, other non-theoretical questions arise. For example, can Setzer and Dybjer's $\delta$-combinator, which seems to encompass almost all the strength of induction recursion, be analysed into smaller and more tractable components? Are there other combinators which, while not adding to the expressive strength of induction recursion, make it easier to use by encapsulating common inductive recursive idioms? An example may be composition. Are there well behaved fragments of induction recursion analogous to, say, the strictly positive fragment of inductive types? Definitions in which $T$ is covariant (so that the associated functor can be defined on all morphisms and not just Cartesian morphisms) could comprise such a fragment. This objective will make induction recursion more accessible to theoreticians and programmers.
- **ST6: Terms and Programs:** Can induction recursion be used to define and reason about programs between data types defined by inductive recursive definitions? This seems possible as correctness of the internalisation of a type constructor within the reflective form of induction recursion is just the statement that the decoding function captures the terms of the type constructor. Thus induction recursion has within it the seeds of a theory of terms as well as types. Concretely we will define morphisms between inductive recursive definitions and obtain a full and faithful embedding of morphisms as natural transformations. This objective allows us to reason about how programs can create, transform, manipulate and consume inductive recursive data structures.

**3. Related Theoretical Questions:** There are many other questions, of which the following seem particularly important: (**SW1:**) We want to show that type theory with induction recursion is strongly normalising. This objective is valuable as many properties follow from strong normalisation and because strong normalisation is required to use induction recursion within a programming language and within proof assistants; (**SW2:**) We will tackle the conjecture that the hierarchy of higher order universe operators described in Palmgren [Pal98], which are definable using Setzer and Dybjer's combinators, in fact exhausts the strength of their system. If true, this objective enables us to reason about induction recursion by reasoning about the potentially simpler theory of universe operators; and (**SW3:**) We want to determine the precise proof theoretic strength of induction recursion. This objective tells us how strong a logical principle induction recursion is, and helps determine its precise relationship to other theories.

**Applied Research**

Induction recursion offers us the power to define a variety of sophisticated structures such as universes, universe operators, and super universes. While universes have begun to be used in programming applications in the literature, the same cannot be said of the other structures. This affords us a tremendous opportunity to deploy these powerful ideas. Indeed, they are only the first in a landscape whose exploration and mapping-out will greatly enhance the expressive power available to programmers. We will begin with the following three areas of research:

**1. Advanced Datatypes:** Currently, advanced forms of data types such as nested types and generalised algebraic data-types (GADTs) are analysed as fixed points of higher order functors. However, this analysis does not express the concrete nature of these data types. As a result, algorithms which rely on the concrete nature of data structures (e.g. searching and sorting algorithms) cannot be extended to cover such advanced data types. We will show that induction recursion can serve as a better foundation for such advanced data types and will demonstrate this empirically by offering examples of algorithms which can only be written (or written with greater ease and clarity) with induction recursion.

- **SW4: Nested Data Types:** The usual presentation of the data type of untyped $\lambda$-terms is as the least solution of the equation $L(X) = X + L(X) \times L(X) + L(X+1)$ for the functor $L$. One way to see this data type as a concrete data structure is to view it as a container, i.e. as given by a type $S$, and a function $P : S \to \mathsf{Set}$ that assigns to each shape the type of positions at which data can be stored in a data structure with that shape. In the case of untyped $\lambda$-terms, this means the following definition of $S$ and $P$

$$\begin{array}{llll} S = 1 + S \times S + (\Sigma s : S)P(s) \to 2 & & & \\ P(in_1(0)) & = 1 & P(in_3(s,f)) & = (\Sigma p : P(s))\ \textbf{if}\ f(p)\ \textbf{then}\ 0\ \textbf{else}\ 1 \\ P(in_2(s,s')) & = Ps + Ps' & & \end{array}$$

This is an inductive recursive definition! That is, the shapes $S$ and the positions $P$ are defined simultaneously. Further, the structure of $S$ and $P$ is precisely aligned to the natural structure of the untyped lambda-terms. This example is general enough to suggest that all of the usual nested types can be defined by induction recursion.

– **SW5: GADTs:** We want to go further and consider GADTs, in which nested types, equality types and existential quantification are combined. Once we know how nested types can be treated using induction recursion, we can use the treatment of the identity type as an (indexed) inductive recursive definition and the treatment of existential quantification as a $\Sigma$-type, to get an inductive recursive representation of GADTs.

– **SW6: Final Coalgebras via Induction Recursion:** A similar situation arises in connection with approximations to infinite objects that inhabit certain coinductive types. If we start with a functor $F$ representing the constructors of the data type, then the data type of finite and infinite trees formed from these constructors can be given from the solution of the equation $F_* = Id + F_* \circ F$. Notice how this presentation of the trees grow downwards as more and more information about the infinite object becomes available. This leads to the following definition of a container $(S_*, P_*)$ for infinite terms (where $F$ is given by a container $(S, P)$)

$$\begin{array}{llll} S_* = 1 + (\Sigma s : S_*) P_*(s) \to S & & & \\ P_*(in_1(0)) & = 1 & P_*(in_2(s,f)) & = (\Sigma p : P_*(s)) P(f(p)) \end{array}$$

**2. Generic Programming:** Generic programming [BGHJ07, BG03, Jan97] aims to define functions that make use of the structure of the type of their arguments. In the object-oriented community, this is called generative programming. The payoff is in reusability, and modularity. Simple examples are generic functions for printing, parsing, traversing or searching data structures. Generic functions cannot be defined for inputs of arbitrary types, so one first delineates the collection of types on which the function is defined. Examples of such collections include polynomial types (closed under product and sum), the differentiable types (with a well-defined notion of one-hole context), and more extensively, the strictly positive types. A number of researchers (including ourselves) working on dependently typed programming have already pointed the direct applicability of universes to generic programming [BDJ03, AMM07]. However, the use made of universes so far has hardly begun to scratch the surface of the available power. We will rectify this as follows:

– **NT1: Type Classes and Universes:** Type classes are an ad-hoc model of types equipped with operations. Universes formalise this by reflecting those operations within $D$, e.g. a universe of types with a print operation could be modelled by a universe with $T : U \to D$ where $D = \Sigma X : *.X \to String$. We will go further in applying universes to reasoning about generic programs. Firstly, we can equip such a universe with *laws* that these operations should satisfy by reflecting them within $D$. Further, we can apply algebraic reasoning to reasoning about type classes based upon our categorical semantics of induction recursion. Both incorporating laws and providing algebraic techniques for reasoning in generic programming are novel, in that current research focuses on the construction of generic programs rather than providing the meta-theory for reasoning about generic programs.

– **NT2: Type Class Operators and Universe Operators:** So far, universe operators have not been applied to generic programming. However, we can see some promising examples, e.g. the `derive` mechanism of Haskell automatically constructs definitions of functions on a large collection of data from a smaller collection. Thus the `derive` operator takes as input a universe equipped with some operations and returns a new universe with new operations - it is a universe operator. Universe operators also appear to be central in Haskell's `instance` declarations, e.g. the fact that the list type constructor preserves showability means if a universe is equipped with a show operation, we can extend it to a new universe, closed under the list constructor, which again has a show operation. Thus, universe operators offer the tantalising ability to treat type classes as first class citizens!

– **NT3: Change of $D$:** The Dybjer-Setzer syntax for induction recursion pivots on data-types $OP_D$ of IR codes. By analysing and manipulating these codes, we can systematically enhance the data-types arising from these codes with further structure: for example by the addition of new constructors, together with embeddings from the old datatype into the new. Concretely this means finding techniques for moving from $OP_D$ to $OP_{D'}$ – note that this is no easy task as $OP_D$ is not functorial in $D$. This question has not been studied either theoretically or via applications and seems to open a new form of genericity in generic programming.

**3. NT4: Implementations, Agda and Epigram:** We want to enable programmers to exploit the wide-ranging opportunities induction recursion offers and, to do that, we will implement our ideas in the programming languages Agda (which has gained substantial interest from other communities since the introduction of Agda2) and Epigram. This will also offer partial correctness of our constructions via type checking and, further, will make our work more accessible by providing code for programmers to experiment with.

But we will go further! We want to do more than simply use these languages to implement our ideas. We want to set ourselves the higher goal of influencing the development of actual programming languages. Therefore we will work with Dr Conor McBride, who is currently redesigning Epigram [MM04], to incorporate induction recursion as the fundamental, principled, mechanism for recursive definitions in Epigram. This research will be the primary responsibility of Dr Altenkirch as he is an expert in the implementation of dependently typed programming languages. Seeing our ideas come to fruition as part of a programming language is one very exciting end-point of our research.

## References

[AAG03] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Procs. FOSSACS 2003*, volume 2620 of *LNCS*, pages 23–38, 2003.

[AAG04] M. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using w-types. In *Procs. ICALP 2004*, volume 3142 of *LNCS*, pages 59–71, 2004.

[AAGM05] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. $\delta$ is for data - differentiating data structures. *Fundamenta Informaticae*, 65:1–28, 2005.

[AGA05] M. Abbott, N. Ghani, and T. Altenkirch. Containers - constructing strictly positive types. *Theoretical Computer Science*, 341(1):3–27, 2005.

[AM03] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Procs. IFIP TC2 Conference on Generic Programming.

[AMM07] T. Altenkirch, C. Mcbride, and P. Morris. Generic programming with dependent types. In *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. 2007.

[BDJ03] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[BG03] R. Backhouse and J. Gibbons, editors. *Generic Programming - Advanced Lectures*, volume 2793 of *LNCS*. Springer, 2003.

[BGHJ07] R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors. *Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer, 2007.

[Car86] J. Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[DS99] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Procs., TLCA 1999*, pages 129–146, 1999.

[DS01] P. Dybjer and A. Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science*, volume 2183 of *LNCS*, pages 93 – 113, 2001.

[DS03] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Logic*, 124(1-3):1–47, 2003.

[DS06] P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66:1 – 49, 2006.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

[Fio05] M. Fiore. Mathematical models of computational and combinatorial structures. In *Procs. FoSSaCS 2005*, volume 3441 of *LNCS*, pages 25–46, 2005.

[GH04] N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *Types for proofs and programs*, volume 3085 of *LNCS*, pages 210–225, 2004.

[GJ07] N. Ghani and P. Johann. Initial algebra semantics is enough! In *Procs. of TLCA 2007*, number 4583 in LNCS, pages 207–222, 2007.

[GJ08] N. Ghani and P. Johann. Foundations for structured programming with gadts. In *Proceedings of Principles and Programming Languages (POPL), 2008*, pages 297–308, 2008.

[Has02] R. Hasegawa. Two applications of analytic functors. *Theor. Comput. Sci.*, 272(1-2):113–175, 2002.

[Jan97] Patrik Jansson. *Functional Polytypic Programming: Use and Implementation*. PhD thesis, Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1997.

[Joy87] A. Joyal. Foncteurs analytiques et espèces de structures. In *Combinatoire énumerative*, number 1234 in LNM, pages 126–159, 1987.

[MA08] P. Morris and T. Altenkirch. Inductive families from indexed containers. In *Submitted to POPL 2008*, 2008.

[MAG07] P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *International Journal of Foundations of Computer Science*, 2007. Accepted.

[MAM06] P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *Procs. TYPES 2004*, LNCS, 2006.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[MM04] C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[Pal91] E. Palmgren. *On Fixed Point Operators, Inductive Definitions and Universes in Martin-Löf's Type Theory*. PhD thesis, Uppsala University, March 1991.

[Pal98] E. Palmgren. On universes in type theory. In *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 191–204. Clarendon Press, 1998.

[Rat00] M. Rathjen. The strength of Martin-Löf type theory with a superuniverse. part I. *Archive for Mathematical Logic*, 39(1):1–39, Jan 2000.

[Rat01] M. Rathjen. The strength of Martin-Löf type theory with a superuniverse. part II. *Archive for Mathematical Logic*, 40(3):207–233, Apr 2001.

[RGP98] M. Rathjen, E. Griffor, and E. Palmgren. Inaccessibility in constructive set theory and type theory. *Annals of Pure and Applied Logic*, 94(1-3):181–200, 1998.

[Set00] A. Setzer. Extending Martin-Löf type theory by one Mahlo-universe. *Arch. Math. Log.*, 39:155 – 181, 2000.

[Set08a] A. Setzer. Universes in type theory part I – Inaccessibles and Mahlo. In *Logic Colloquium '04*, pages 123 – 156, 2008.

[Set08b] A. Setzer. Universes in type theory Part II – Autonomous Mahlo and $\Pi_3$-reflection. Submitted. Available via http://www.cs.swan.ac.uk/∼csetzer/index.html, 2008.