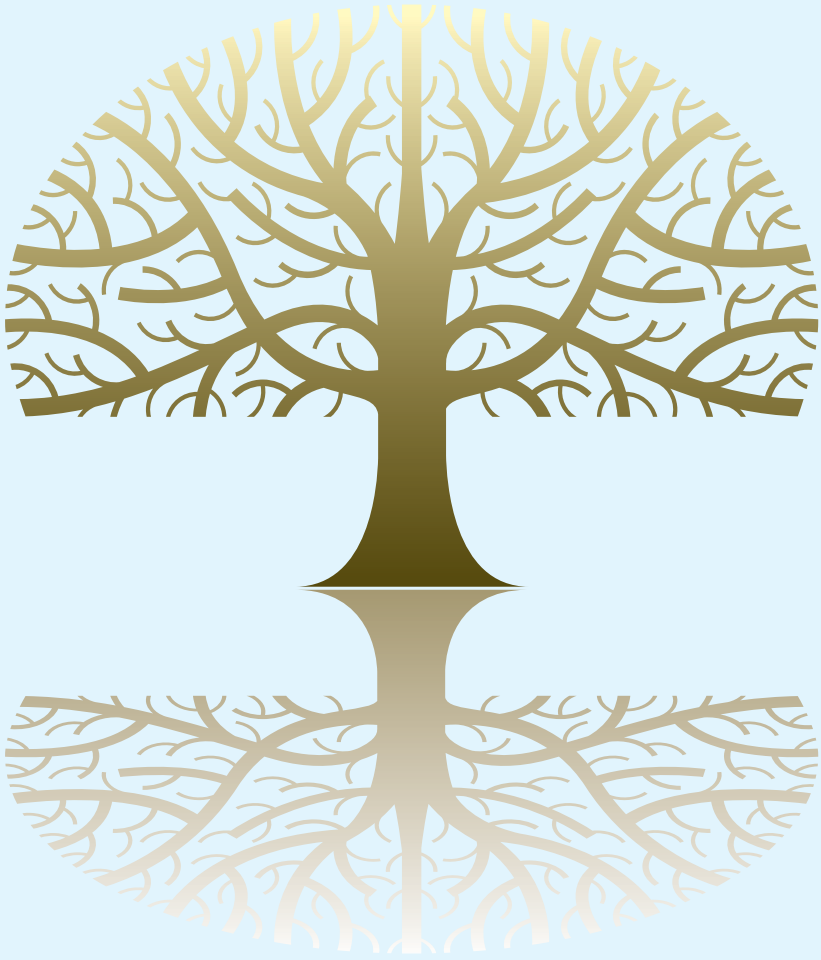
Primitive Direcursion and Difunctorial Semantics of Typed Object Calculus

Johan Glimming



Doctoral Thesis in Computer Science at Stockholm University, Sweden 2007

The first part of this thesis consists of two research papers and concerns the field of denotational semantics of typed object calculus, i.e., a formalism for object-based programming. It provides a category-theoretic semantics based on partial maps subject to an algebraic compactness assumption. This semantics interprets types as mixed-variant functors (so-called difunctors) and is therefore termed difunctorial. In addition, we prove computational soundness and adequacy results for typed object calculus via Plotkin's FPC, thus providing a class of computationally adequate models for an extended first-order typed object calculus (with recursive objects supporting method update, but not subtyping). Taken together, this provides a mathematical foundation for studying program algebras for object-based programming languages.

The second part (which appeared at CALCO 2007, Springer-Verlag) deals with recursion principles on datatypes, including the untyped lambda calculus as a special case. Freyd showed that, in certain domain theoretic categories, locally continuous functors have minimal invariants, which possess a structure that he termed dialgebra. This gives rise to a category of dialgebras and homomorphisms, where the minimal invariants are initial, inducing a powerful recursion scheme (direcursion) on a complete partial order. We identify a problem that appears when we translate (co)iterative functions to direcursion, and as a solution to this problem we develop a recursion scheme (primitive direcursion). This immediately gives a number of examples of direcursive functions, improving on the situation in the literature where only a few examples have appeared. By means of a case study, this line of work is connected to object calculus models.

Johan Glimming received his BSc (in Computer Science) from Uppsala University, and his MSc (in Mathematics and the Foundations of Computer Science) from University of Oxford. In 2005, he received the title of Licentiate of Philosophy at Stockholm University.

ISSN 1653-5723
ISBN 978-91-7155-550-2

Department of Numerical Analysis and Computer Science
STOCKHOLM UNIVERSITY, SWEDEN

Nada är en gemensam institution mellan
Kungliga Tekniska högskolan och Stockholms universitet.

Primitive Direcursion and Difunctorial Semantics of Typed Object Calculus

JOHAN GLIMMING

Avhandling som med tillstånd av Stockholms Universitet
framlägges till offentlig granskning för avläggande av filosofie doktorsexamen
onsdagen den 9 januari 2008 kl 14.00
i sal F3, Lindstedtsvägen 26 (infart Valhallavägen/Drottning Kristinas väg),
Kungliga Tekniska högskolan, Stockholm.

TRITA-CSC-A 2007:22
ISSN 1653-5723
ISRN-KTH/CSC/A–07/22–SE
ISBN 978-91-7155-550-2

TRITA-CSC-A 2007:22
ISSN 1653-5723
ISRN-KTH/CSC/A–07/22–SE
ISBN 978-91-7155-550-2

Stockholms Universitet
Numerisk analys och datalogi (KTH CSC)
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Stockholms Universitet framlägges till offentlig granskning för avläggande av doktorsavhandling den 9 januari 2008 kl 14.00 i sal F3, Lindstedtsvägen 26 (infart Valhallavägen/Drottning Kristinas väg), Kungl Tekniska högskolan, Stockholm.

Tryck: Universitetsservice US-AB

Abstract

This thesis concerns two closely related lines of research: (i) We contribute to the semantics of typed object calculus by giving (a) a denotational semantics using partial maps making use of an algebraic compactness assumption on the ambient category, (b) a notion of "wrappers" by which algebraic datatypes can be represented as object types, and (c) proofs of computational soundness and adequacy of typed object calculus via Plotkin's FPC (with lazy operational semantics), thus making every denotational model of FPC (with these properties) a computationally adequate model also for a first-order typed object calculus (with recursive objects supporting method update, but not subtyping). In this way, we give a mathematical foundation for studying program algebras for object-based programming languages, since a valid equation in the model is proved to induce operationally congruent terms in the language. For (c), we also develop a variation of Abadi and Cardelli's first-order typed object calculus with recursive object types and sum types (and some other extensions), and prove subject reduction for this calculus. (ii) The second part concerns recursion principles on datatypes including the untyped lambda calculus as a special case. Freyd showed that in certain domain theoretic categories, locally continuous functors have minimal invariants which possess a structure that he termed dialgebra. This gives rise to a category of dialgebras and homomorphisms, where the minimal invariants are initial, inducing a powerful recursion scheme (direcursion) on a complete partial order. We identify a problem that appears when (co)iterative functions (on a fixed parameterised datatype) are translated to direcursion (on the same datatype), and as a solution to this problem we present a recursion scheme (primitive direcursion), generalising and symmetrising primitive (co)recursion for endofunctors. To this end, we give a uniform technique for translating (co)iterative maps into direcursive maps. This immediately gives a plethora of examples of direcursive functions, improving on the situation in the literature where only a few examples have appeared. Moreover, an ad-hoc solution proposed elsewhere is avoided for the translated maps, while interesting new examples appear (bisimulations, higher-order coalgebra), also in the context of models of typed object calculus.

Contents

Acknowledgements

First and foremost, I would like to thank my supervisor Karl Meinke, my department, and Stockholm University for taking me on as a PhD student and for funding and supporting my research. I am also most grateful to my coauthor Neil Ghani for many enligthening discussions, for his wit and personal qualities and for his friendship. I am equally grateful to Faron Moller, who introduced me to computer science research and whose advice and encouragement I value greatly, and to Björn Lisper for his support and advice throughout the years. I am also indebted to Viggo Stoltenberg-Hansen for his courses in logic and domain theory in Uppsala, for many helpful discussions and for the interest he has shown in my work.

Over the years, I have had the opportunity to visit researchers in different parts of the world. I am grateful for their hospitality, encouragement and openhanded support. The suggestions I have received on those occasions, and the many awarding discussions, have greatly influenced and improved my work, and sometimes pointed me in new directions. Here I want to particularly mention: Ralph Back, Roland Backhouse, Marcelo Fiore, Andy Gordon, Furio Honsell, Graham Hutton, Patrik Jansson, Achim Jung, Marina Lenisa, Rasmus Møgelberg, Andy Pitts, Gordon Plotkin, John Power, Horst Reichel, Bernhard Reus, Alex Simpson, Hendrik Tews, Tarmu Uustalu and Varmo Vene. I am also grateful to Martín Abadi, Viviana Bono, Luca Cardelli, Benjamin Pierce, Ramesh Viswanathan, among others, for helpful correspondence. It has also been a privilege to get to know many of the people in the Uppsala-Stockholm logic group, researchers at the IT department in Uppsala and the colleagues at KTH/SU.

During the work on this thesis, I have received financial support from Lennander's foundation at Uppsala University, Wallenberg foundations, as well as scholarships from Stockholm University. They are hereby gratefully acknowledged.

Last but not least, I am grateful for the support of my friends and family, who are important well beyond this thesis. I am particularly thankful to my sister Ida for proof-reading sections of an earlier version of the text, and to my parents for their unfailing support. And to Pia, thank you for your love.

Chapter 1

Introduction

This thesis concerns the denotational semantics approach to the semantics of programming languages. This approach is based on assigning elements in suitable mathematical structures to the various language constructions. To this end, we will be interested in *object-based programming languages*, a class of programming languages based on entities known as *objects*, which consist of a family of self-referential *methods* acting on the object. These objects compute in response to *method invocation* or *method update*. The former stimulus is similar to function evaluation, but there is no argument or input, as the evaluation is with respect to the present *self* of the object, which provides a *local state* for the computations being carried out. Method update makes change possible, and a sequence of method updates and method invocations together give the ability to simulate in particular purely functional programs, so that any partial recursive function can be represented solely by objects. Moreover, the popular class-based paradigm, with representative languages such as Java [GJSB00], arises as a special case by restricting to method updates with constant methods (which are sometimes called fields).

From a software engineering point of view, object-oriented languages have been argued to offer advantages over the traditional structured/imperative or purely functional programming approaches. Firstly, these languages are based on a correspondence between computer simulated physical systems and the physical system itself. This correspondence has lead to the development of a vast number of software engineering methodologies and programming libraries and is often held to support activities such as analysis, design and maintenance of computer programs, i.e. as being resilient with software models. As a result, the object-oriented approach has been postulated to be "proven uniquely successful" [AC96].

It can therefore seem surprising that, from a mathematical viewpoint, quite little is known about these languages, and that formal methods for object-oriented and object-based languages are arguably still at their infancy (and in particular so from a program algebra and equational logic perspective). In this thesis, we focus our attention on the relationship between models of the untyped lambda calculus and object-based programs and study computation principles (recursion schemes) that we argue can improve this paradigm and our understanding of objects as mathematical structures, thus contributing in particular to the development of formal methods. Interestingly, some of our results are also of more general importance (recursion schemes on mixed-variant datatypes, with applications to bisimilarity), and this research sets the stage for further investigations on such topics.

We study the denotational semantics, or model theory, of object-based programming languages following some particular directions. Firstly, we have chosen a class of programming languages based on the existing formalism called typed object calculus (developed by Abadi and Cardelli [AC94a, AC94b, AC96] over a period of some years, see below). This formalism takes objects as *primitives*. In particular, there are only object types; ground types are not needed, since many useful datatypes can be defined using pure objects. Moreover, this calculus supports *subtyping*, the ability to use an *extended* object, i.e. one with additional methods, instead of an object of the specified type. This last feature is, however, not directly considered in this thesis, since it is a known open problem for the approach we have chosen [AC96, Aba07]. Secondly, we have taken an axiomatic categorical approach to denotational semantics. By contrast to the more traditional interpretation based on environments, we will use composition in a category to give a more abstract and versatile treatment. The axioms that we require are taken from a branch of domain theory known as axiomatic domain theory [Fio96b, Fre90, Fre91]. The most important axiom is *algebraic compactness* which states that initial algebras and final coalgebras coincide.

The operational semantics formally determines what we mean with concepts such as "method update" and "object" in the first paragraph. It does so by means of syntactical operations and the responses to such operations in any program context and by means of a rule system. In this respect, the operational semantics describes the meaning of the language by stating how it is executed on an abstract machine (or rewrite system) which manipulates syntactical expressions. Although an analysis of such a machine can be important, c.f. proof theory, the denotational semantics approach, as pioneered by Dana Scott and Christopher Stratchey [SS71], provides a more abstract explanation of programming concepts, independent of syntax. Moreover, finding mathematical structures that give meaning to

programming languages typically makes a more diverse mathematical toolkit available, which can be used to make progress on programming language design, attain soundness and suitable completeness properties, etc. For instance, the denotational semantics of untyped lambda calculus more or less gave birth to the area known as domain theory, which describes a program in terms of approximations. When we have a model, i.e. denotational semantics, we can be sure that no paradoxes exist in the equational theory (here: untyped lambda calculus). This was far from obvious [Bar84] before a model had been discovered by Dana Scott [Sco69], notably because of lambda terms which can be applied to themselves.

However, a model does not generally characterise the programming language notions (objects, functions, etc), and therefore does not say what these are in a more strict sense. In particular, it often happens that the model does not quite match the operational semantics since semantic completeness (truth implying provability for a particular model) cannot hold in general for programming languages with higher types and recursion (e.g. untyped lambda calculus or typed object calculus). To see this, note that completeness is here the property that two programs with same denotation must be provably equal in the equational theory given by the programming language. The problem is that in general there can be many normal forms (clearly not provably equal), which in the more abstract universe (i.e. in the model) determine the same relationship between input and output, and, hence, are the same functions. Since such extensionality is a most natural mathematical property, we have to let go of completeness. As an alternative, Milner and Plotkin formulated a notion of full abstraction in the 70ies. This involved a notion of *operational equivalence* which is (generally) different to the provability relation of the equational theory. Two programs (or program "snippets") are said to be equivalent if in all program contexts (programs with a hole, as it were), these two programs will (if inserted at the same hole) reduce to the same value using the operational semantics; otherwise, they both diverge. In this way, full abstraction could be formulated: the property that any two programs with equal denotations are also operationally equivalent, and vice versa. In particular, two normal forms which have equal denotations are therefore required to respond to all the syntactical operations in the operational semantics similarly.

Untyped lambda calculus has no classical set-theoretic models beyond the trivial one-element model (as Scott showed [Sco69]). The same is true for other programming languages, such as polymorphic lambda calculus [Rey84]. Moreover, many typed programming languages, such as simply typed lambda calculus, have found natural and versatile interpretations using category theory. In particular, Lambek and Lawvere's notion of cartesian closed category turned out to be es-

sentially equivalent to such simple type theories. As a result, any cartesian closed category, such as **Set** of sets and total functions, can be used as a model of a simply typed lambda calculus (all we need is a functor). (Such categorical semantics of logic has driven the development of new logics [Pit01].) The situation for untyped lambda calculus is more complex in one important respect, since it requires a solution to an equation of the form $D \cong D^D$. This requirement is largely incompatible with cartesian closure and since an initial solution arises as the singleton set, and no other solution can exist because of cardinality arguments, sets are themselves not sufficient as models. This motivated Dana Scott to develop domain theory. He equipped D with a suitable ordering and constrained the space D^D to contain precisely the functions that preserves this ordering in a suitable sense. The ordering gives a level of refinement between programs, the undefined program $\perp$, being least. The structure on D is a *Scott-Ershov domain*. A substantial body of research has aimed at finding suitable categories of domains, having sufficient closure properties and structure for the interpretation of programming languages (or for other applications). This has turned out to be a remarkably complex task, and research on finding suitable categories still progresses (see e.g. [BBS04, FJM$^+$96]).

When we formulate a categorical semantics for typed object calculus we face similar problems, since typed object calculus is similar to untyped lambda calculus in requiring roughly the same domain equation to be solved (notably, the untyped lambda calculus is a special case). But in addition, the types demand suitable type structure in the category, and this places two rather orthogonal (and incompatible) demands on the universe of discourse. Because of this, the present thesis builds on previous work on axiomatic domain theory: our categorical semantics relies on carefully chosen axioms on an enriched ambient category, in the vein of e.g. Freyd [Fre90], Fiore [Fio96b] and Moggi [Mog89].

To summarise our approach to object-based programming language models, we identify what we regard as the most fundamental questions underlying this line of research:

- *What is an object?* Several operational semantics have been proposed, and there is also a natural self-application semantics which uses domain theory. However, the latter has so far not been combined with subtyping. Moreover, no fully abstract model is known, as self-application semantics, in particular, is not fully abstract.

- *What is a model of typed object calculus?* The developments of axiomatic domain theory, and computational effects via monads, makes possible to

view a model as a somewhat abstract notion, which depends on the choice of a particular mathematical universe, monad, etc. It remains uncertain, however, what exactly the appropriate axioms are, and what categorical structures must be considered in this case? Moreover, it is not evident what constitutes a *fully abstract* model, what the appropriate observational congruence relation is, what (bi)simulation relations on programs are of interest, how those relations are characterised denotationally, and what properties can be established for them.

- *How can object-based programming languages be improved?* In particular, present techniques of combining two objects are quite limited, and the computation schemes on object-based programs have to date been remarkably limited compared to the complex recursive domain equations that are generally involved in models. In particular, what are the recursion schemes on objects, and how can these be used in programming and in formal methods? How much structure from the denotational semantics should be pushed back into the operational semantics in the future, and what proof principles should be added?

In approaching these underlying major questions, any of which is too great to be fully covered in a single treatise, other more technical questions inevitably appear (c.f. untyped lambda calculus). To this end, this thesis consists of three research papers, organised in chronological order. The first develops a denotational semantics based on partial maps, with an algebraic compactness assumption on the ambient category. The second develops an interpretation of a variation of Abadi and Cardelli's typed first-order calculus, which is proved to possess the subject reduction property, and then gives a proof showing that every computationally sound and adequate model of Plotkin's metalanguage FPC, automatically gives a model of this typed object calculus, which again has these two properties. Our result in other words establish that work on FPC models carry over to typed object calculus semantics in a strong technical sense (adequacy), while also showing that some care must be taken, since the property breaks unless a lazy operational semantics is considered for FPC. It follows that any model of lazy FPC can be used for studying program transformations (program algebras) for typed object calculus. The third paper develops a new recursion scheme which is shown to be useful for denotational semantics of such calculi, although it is also of more general interest e.g. for untyped lambda calculus and bisimilarity. This third paper provides a main technical result given together with a case study showing its relevance to denotational semantics of typed object calculus.

1.1 Aims

This thesis contains two main themes, which are demonstrated to be closely connected: (i) semantics of typed object calculus and (ii) recursion schemes.

1.1.1 Semantics of Typed Object Calculus

In this thesis, we consider object-based programming languages in the sense of Abadi and Cardelli's typed object calculus [AC96], to which operational semantics we make some amendments. We give a denotational semantics of typed object calculus using a category of domains and partial maps. The use of partial maps has some appeal (which we discuss more in detail in Paper I), and we are able to develop a functorial semantics where types are modelled as symmetric functors and terms as indexed families of morphisms. This demonstrates an intriguing connection to Freyd's mixed-variant recursion principle, and shows that we can define objects by the recursion principle afforded by the categorical interpretation.

An alternative approach to models is given in chapter four (Paper II), where we show that, by using a translation, and by excluding subtyping, object types can be seen as certain kinds of recursive types in Plotkin's FPC (fixed-point calculus) (with lazy evaluation strategy). We establish a computational adequacy result and various soundness properties for this translation. This gives immediately that any model for lazy FPC gives a computationally adequate model for typed object calculus without subtyping.

Although the thesis is not directly concerned with methods for formally proving object-based program correct, such methods are strongly connected to model theory, since new proof methods can often be discovered through a model (e.g. [Pit96]). We have left as future work to carry out more comprehensive practical studies, extend the framework to suit formal methods (e.g. by considering relational specifications, refinement, etc) and to extend the equational theory of object calculus with suitable proof principles. We also have omitted a treatment of full abstraction and bisimilarity (along the lines of [Gor98] and [Vis98]), which will appear elsewhere.

1.1.2 Recursion

Solutions to recursive domain equations involving function spaces can be given as initial $\hat{G}$-algebras in suitable categories, where $\hat{G}$ is an endofunctor given by symmetrising G. This was shown by Freyd [Fre90, Fre91] (based on work by Smyth

and Plotkin [SP82]) and later refined by Fiore [Fio96c] in a framework of enriched category theory. Initial $\hat{G}$-algebras (also called *dialgebras*) generalise usual algebras and coalgebras. Moreover, a recursion principle arises for $\hat{G}$-algebras. Henceforth, this principle is called *direcursion*; it is one of the main topics of the present thesis.

More specifically, we will investigate the relationships between (co)iteration and direcursion for a *fixed datatype*. With (co)iteration we mean the unique homomorphisms associated to the initial (final) $\hat{G}(\mu\hat{G}, \_)$-(co)algebras by Bekič's Lemma, i.e., (co)iterative maps on this particular parameterised datatype. Since the carrier of this (co)algebra coincides with the solution $O = \mu\hat{G}$, we address the question of how these schemes compare to direcursion for the same functor G. Our result shows that by generalising direcursion (by precomposing with an injection map or postcomposing with a projection map), we can express all such (co)iterative maps as a canonical direcursive map such that the same computation is carried out at every stage. We call this generalisation *primitive direcursion* since it is primitive recursion for symmetric functors $\hat{G}$, i.e., it is simultaneously primitive recursion and primitive corecursion for recursive types. One aim of the present thesis is to develop this notion and to provide examples of how it can be used, particularly in the semantics of object-based programming languages. To this end, primitive direcursion as developed in chapter five (Paper III) forms a main theoretical result of this thesis.

1.2 Related Work

This study could have taken many different starting points. One could have been to emphasise formal methods, namely proof techniques and practical examples of program equivalences. This is not our present approach, partly because the model theory of object-based programming languages itself deserves a thorough investigation in the light of the limited research in the area (the only model beyond the self-application approach that we follow is to the best of our knowledge the one due to Abadi and Cardelli [AC96], for which see below). In this section we will discuss in more detail how our present approach compares to existing work in related research areas.

1.2.1 Models of Programming Languages

As this thesis is concerned with sequential programming languages without global state, we will consider next some related work in denotational semantics of func-

tional, class-based and object-based sequential languages.

Functional Languages

The first models for untyped lambda calculus were discovered by Scott [Sco69, Sco73], although the graph model is due to Plotkin [Plo72]. In particular, the model D_∞ was based on a colimit of complete lattices (which forms a cartesian closed category such that D^D has same cardinality as D), but can also be carried out in **CPO**. Additional results were developed by e.g. Wadsworth [Wad76], including the characterisation of the compact elements. Meyer [Mey82] compared various approaches to models of untyped lambda calculus and proposed the so-called environment model. Note that the untyped lambda calculus has a model which is a canonical (initial/final) solution to a recursive domain equation, though it arises from a chain created from a given object D [SP82]. That is, the solution is free but not initial.

Abramsky [Abr90] noted that the untyped lambda calculus is not commonly implemented in programming languages. From this observation, he developed a theory of lazy lambda calculus, which is based on weak head reduction and weak head normal forms. In other words, terms are not reduced under lambda binders, and terms such as $\lambda x.\Omega$ are already in (weak head) normal form and regarded as values. Furthermore, Abramsky developed a notion of applicative bisimilarity which he showed characterises the observational equivalence for lazy lambda calculus. He also proved an internal full abstraction result and established that lazy lambda calculus extended with parallel combinators is (inequationally) fully abstract with respect to his domain-theoretic model. Other results included finding a canonical domain equation $D \cong (D^D)_\perp$ (for non-strict exponential in **CPO**). Abramsky's approach has subsequently been generalised by Honsell et al [HL99] and Fiore [Fio96c], among others.

Egidi et al [EHR92] is instead concerned with the lazy call-by-value lambda calculus and its associated notion of applicative bisimilarity, based on Abramsky's work. This language turns out to also have a canonical recursive domain equation, namely $D = [D, D]_\perp$ where $[D, D]$ is the space of strict (Scott-continuous) functions. This makes the work by Egidi et al strongly connected to the present thesis, which is based on generalisation of their domain equation. It also highlights the fact that typed object calculus has an evaluation strategy which is somewhere in between call-by-value and call-by-name [AC96], since the argument is always the object itself. For lazy call-by-value lambda calculus, an application MN is evaluated by first reducing M into an abstraction, and then reducing N into an ab-

straction before substitution. Note that in lazy lambda calculus, the substitution happens before N has been reduced.

Type theories date back at least to the philosopher Bertrand Russel [Hin97] and were introduced mainly to avoid paradoxes coming from e.g. self-application. The simply typed lambda calculus is due to Church [Chu40], based on a typed combinatory logic due to Curry a few years before. Henkin [Hen50] proved a completeness theorem for simply typed lambda calculus (based on work by Gödel) and later formulated what came to be known as Henkin models. Such models consist of typed applicative structures (an operation $\cdot : B^A \times A \to B$ for all interpreted types A and B, essentially) subject to extensionality. Such models can be characterised using environment models, which use the syntactical set of terms to assert that there are enough points. An alternative is the combinatory model that requires two constants s and k at each suitable type, subject to certain axioms. Subsequently, it was realised (most notably by Lambek) that cartesian closed categories generalise the notion of Henkin models. Later, Lambek established a correspondence between cartesian closed categories and simply typed lambda calculus [Lam80], showing that theories formulated over simply typed lambda calculus correspond to product-preserving functors on cartesian closed categories, giving a similar situation to Lawvere's algebraic theories (which capture finitary universal algebra, i.e. the first-order case) [Law64], but for "higher types". (For more details, see e.g. [LS86, BW85].) In summary, there has been a vast development of cartesian closed categories and their use for models of typed lambda calculus, even though these are, after all, quite basic type structures. In this thesis, we will as much as possible build on this work, in particular by developing a categorical semantics which mimics Lawvere's functorial semantics. However, we leave as further work to express models as functors from a classifying category ($Cl(\varsigma)$, to wit), and we will not go as far as e.g. [Jac99] and pursue a fibred category theoretic semantics, since the recursive types will still be comparably simple (not having dependencies on terms, or polymorphism).

There has been quite a bit of research on algebraic approaches to formal methods of functional programming languages. We will refer to this line of work as the algebra of programming-school (it is elsewhere also known as Bird-Meertens formalism or "squiggol"). This approach is detailed in a book by Bird et al [BdM97] on the topic, but the underlying ideas trace back at least to Bird's theory of lists [Bir87, Bir89]. Between these milestones, much research was devoted to the use of recursion schemes in functional programming [Mee86, Mee96, Mee92, BJJM99, Fok94, MH95, MJ95, Hoo96, Mal90, Mee92] and to finding suitable programming languages to make use of such schemes [Jan00, JC94, CF92, Hag93]. This

line of work is relevant for the present thesis as it was the main inspiration for studying Freyd's direcursion scheme in the first place. Moreover, once recursion schemes and models have been developed, the question of suitable proof principles [Acz88, TR98, Rut00] and their application to functional languages [GH05, Gor94]) should be addressed. Regarding such proof principles, direcursion is less well-understood than (co)induction (although [Pit96] provides a proof technique, this view is also held by Fiore [Fio96b], who began to improve on it in [Fio96c]). Secondarily, the question regarding which programming languages should be used to express such recursion principles arises. For object-oriented languages, but not for object-based languages and direcursion, this latter question has already been addressed [Wei02].

Class-Based Languages

Coalgebraic techniques are by now well-established for reasoning about class-based programming languages [Rei95, Jac98, HHJT98]. There are some relationships between this work and the present for object-based languages but there are also some profound differences which we now will outline.

In the coalgebraic approach to classes, an endofunctor $F : \mathbf{Set} \to \mathbf{Set}$ is taken as the interface of a class, and a structure map $c : S \to F(S)$ constitute a class "implementation". The set S consists of the states, and an object is one particular element $s \in S$. We immediately get a notion of bisimulation from the fact that c is a F-coalgebra. Moreover, since there is a final coalgebra $(\nu F, f)$ in **Set**, there is also a greatest bisimulation, i.e. bisimilarity. Hence it is possible to state that two objects are bisimilar, meaning that after no sequence of transitions, those two objects can be observed to be distinct, although they could generally be distinct elements of S at any point. Thus this approach benefits from being quite natural.

The object-based programming languages cannot use endofunctors on **Set**, since the involved structures are mixed-variant rather than polynomial functors. This means that we must move to the category **CPO** (or a similar category). In this setting, the theory of universal coalgebra is far less well-understood (see e.g. [Len98]). There have been efforts to remedy this, e.g. [TR98, Pit96, Fio96c], but the structures appearing for object-based programs are not generally coalgebraic, but rather dialgebraic [Fre90], meaning that they are simultaneously coalgebraic and algebraic (while also enriched over domains). Moreover, bisimulations take a different form than prior work, since for object-based languages the object itself constitutes the state. In the most basic case, one can endow an object type with a coalgebraic structure through the self-application morphism. However, this

does not capture the observational equivalence intended. Therefore, the situation is less obvious; further analysis is required (the author has recently began to analyse what bisimulations exist and what are their properties - but these results will appear elsewhere, and are linked to the full abstraction problem).

Some research has been devoted to giving denotational semantics for a parallel object-oriented programming language [dB91, AdB91, Rut90, AdBKR89]. This line of work uses a category of complete metric spaces, i.e. a different approach to denotational semantics than the one pursued in the present thesis. Their approach uses Banach's fixpoint theorem which ensures that certain "contractive" endomaps have unique fixpoints (this being the analogue of the existence of least fixpoints of continuous functions in the domain-theoretic realm). A notable difference with respect to domain theory is that two denotations are assigned a distance, indicating (quantitatively rather than qualitatively) the extent to which a program is a better approximation than another program (see De Bakker et al [DD96] for an overview of this approach). In this setting it is also possible to solve recursive "domain" equations. The latter is done following work by America and Rutten [AR89] who introduced a notion of locally contractive endofunctor in the vein of work by Smyth and Plotkin [SP82] in domain theory. Subsequently, Turi and Rutten [TR98] showed that the category **CMS** is in fact an algebraically compact category. As a consequence, it would have been possible also to use complete metric spaces instead of domain-theoretic categories in this thesis (see in particular the denotational semantics we give in Paper I, but also the recursion scheme in Paper III).

One early approach to the semantics of class-based programming languages was the so-called recursive record semantics [Car88, KR94, Wan94, Coo87, Coo89, CP89, Bou04]. This semantics is based on a typed lambda calculus with records and record types, and objects are values which are defined using a fixed point operator on terms (i.e. to create an object we apply the fixed point operator on a record-valued function abstracted on self). This essentially hard-wires the self into the term, and renders updates impossible. Subsequently, semantics of class-based programming languages have often been given by providing an "encoding" into a suitable typed lambda calculus (see e.g. [BCP99, PT94]); recursive records is one example. Various encodings with existentials have also been studied (e.g. [PT94, BCP99]). The target calculi can be powerful systems, including e.g. Cardelli's $\mathbf{F}_{<:}$ [Car91] or $\mathbf{F}_{\mu<:}$. Yet, Thomas Streicher has argued that this approach to semantics was potentially "built on sand" [Str02] because the target calculi are not simpler than the source calculi. An alternative would instead be to develop denotational semantics directly, or to use a target calculus which has existing models (such as Plotkin's FPC). In this thesis, both alternatives will be investigated, but

the emphasis is on denotational models.

Object-Based Languages

Unlike class-based languages such as Java, an object-based language treats the methods of an object as part of the state, and not as belonging to a fixed set of classes. The most important related work in this area is the typed object calculus developed by Abadi and Cardelli [AC94a, AC94b, AC96]. Closely related is the work by Mitchell et al on the lambda calculus of objects [Mit90, FHM94, Fis96], which partly predates Abadi et al's work. While typed object calculus follows the Church-style in terms of typing, the latter uses Curry-style and also supports both method extension and method extraction. The last two features are not usually part of Abadi and Cardelli's calculus, which instead supports (full) subtyping (and polymorphism). However, several papers have been devoted to combining subtyping with method extension purely in terms of operational semantics and encodings (e.g. [Liq98, BL95, BBL96]). Most notably, Liquori [Liq97] developed an extension of Abadi and Cardelli's adding precisely method extension in the spirit of the lambda calculus of objects. Subsequently, Di Gianantonio et al [DHL98] studied self-inflicted method extension, i.e., the ability of an object to add more methods to itself as a result of method invocation.

Object-based programming languages can be modelled using self-applicative functions, as was noted already by Kamin [Kam88] in his denotational semantics for Smalltalk-80 (see also [KR94]). This kind of semantics differs from the recursive record semantics [Car88, KR94, AC96], where the recursion is in the output or covariant position while the contravariant occurrence of "self" is replaced by having a separate state type, and a fixed point operator at the level of terms. (Note that contravariance is present also in the existential encodings of objects (e.g. [PT94, BCP99]), but there it is hidden under an existential quantifier.)

Reus and Streicher [RS04] take self-application semantics as the starting point for their denotational semantics, and their work is in this respect similar to the approach chosen in the present thesis. They are concerned with certain kinds of program logics for object-based programming languages. Particularly, they prove (in [RS06]) that a Hoare logic developed earlier by Abadi and Leino [AL97] is sound, and they investigate the existence of recursively defined specifications of object-based programs. The Hoare logic approach was also investigated by Tang [Tan02] in his PhD thesis. The present thesis does, however, not concern Hoare logic. Instead, the emphasis is on the development of equational program logics,

and for these purposes categorical semantics and recursion schemes are of central importance.

Aceto et al [AHIK00] proves an computational adequacy result for Abadi and Cardelli's per model [AC96] and a functional first-order typed object calculus (with recursive types). This result is similar to the result in Paper II, but for a model based on an untyped universe on which partial equivalence relations are used for interpreting types. Schwinghammer [Sch05] gave a computational adequacy proof as part of his dissertation (which appeared after our Paper II, and cites it). His proof uses a formal approximation relation rather than an encoding into lazy FPC, and is for an untyped universe (with self-application semantics) and an imperative object calculus.

1.2.2 Process Algebra

Although this thesis concerns sequential programming languages, related work on process algebra is abundant. In particular, several formalisms in the π-calculus family (for which see e.g. [MPW92, SW01, Mil99]) have been used to model imperative or concurrent objects. For example, Walker [Wal95] developed a translation of POOL (an object-oriented language with parallelism) into π-calculus while Sangiorgi [San98a] gives a translation of Abadi and Cardelli's first-order (functional) typed object calculus into π-calculus such that subtyping rules are valid via a sorting discipline extension of the π-calculus. Moreover, Sangiorgi proved a computational adequacy result for his translation, and argues that the π-calculus is useful as a metalanguage for typed object-based programming languages. Gordon and Hankin [GH98] extended Abadi and Cardelli's typed object calculus with primitives for concurrency. Their system is based on an imperative object calculus, but in addition provides facilities for assigning names to objects, for parallel composition, and name scoping operators from the π-calculus. (As far as we know, there is no known denotational semantics for this quite complex language.) More recently, Kleist [Kle00] has translated (typed and untyped) object calculus into π-calculus, again with a computational adequacy result. He has also given further arguments that process algebra techniques are useful for reasoning with object-based programs.

Here, we will offer some arguments towards the view that typed object-based programming languages should be studied in their own right, and not solely via reductions to calculi of mobile processes such as π-calculus:

- Whereas π-calculi models are tailored for handling the nominal aspects of

the calculi (see e.g. Ghani and Yemane [GVY04] for a denotational semantics based on functor categories), treatments of untyped lambda calculus such as Barendregt [Bar84] aims specifically at abstracting away from syntactical aspects of substitution by means of the well-known variable convention. One could argue that it is an extra complication to handle such nominal aspects when they are not strictly needed, and that a mathematical model should aim precisely at abstracting from such aspects (when they are not an explicit part of the language under consideration, such as in loc. cit.). To this end, we view denotational semantics as serving the purpose of providing an syntax-independent characterisation of what an object is (here, an element in a suitable domain given from a recursive domain equation).

- λ-calculi (including typed and untyped, eager and lazy ones) have been encoded into π-calculus (e.g. [MPW92]), and similar translations have also been studied for Abadi and Cardelli's object calculus [Kle00]. For untyped (lazy) lambda calculus, it has been shown that these translations in general cannot be fully abstract (i.e the translation equates all operationally equivalent terms and no other terms), unless non-confluent extensions of untyped lambda calculi are considered [SW01]. This gives an indication that π-calculus encodings are not appropriate for giving semantics of untyped lambda calculus. Similar results are obtained by Kleist and Hüttel [Kle00] for object calculus (i.e. full abstraction of their untyped encoding did not hold). In addition, encodings does not deal with concepts such as approximation/convergence, totality, etc, which becomes available through a syntax-independent denotational semantics based on e.g. domain theory, as developed in Paper I. [1] Note also that both untyped lambda calculus and (typed or untyped) object calculus can encode π-calculus, since these languages are Turing complete.

- Kleist [Kle00] has argued that process algebra techniques can be used for reasoning with object-based programming languages (although he also states that his coauthors disagree with him on this assessment). However, even if process algebra techniques can help to further our understanding of object-

[1] It is, however, interesting to note that much research has been devoted to higher order operational techniques (see e.g. [GP97]), and that notions such as approximations (compact elements) have made their way into operational semantics. We will not here judge whether a purely operational approach suffices, but we observe that much of the developments in the area uses domain theoretic or coalgebraic techniques and notions, indicating that such techniques are not independent of denotational semantics research and are likely to benefit from progress of the latter.

> based programming languages (or at least provide tools for reasoning with them), this should not preclude studies of the latter in their own right. In particular, it is important to find suitable mathematical abstractions, rather than relaying the interpretation of the sequential language via a process/mobility calculi and a complicated encoding from one syntax to another, since, in the end, this typically reveals quite little about what objects really are in the ambient mathematical universe, and hence does not support the development of proof techniques tailored specifically for objects to the extent of a direct mathematical model.

It is suggested, therefore, that process algebra encodings alone are not sufficient or appropriate for giving a full understanding of sequential object-based programming languages, and that the latter require their own denotational semantics to be developed and analysed.

1.2.3 (Co)Recursion and (Co)Induction

A substantial amount of research has been devoted to finding and expressing notions of recursion, and dually corecursion, in suitable category-theoretic terms. First, the idea of a category-theoretic notion of iteration on the natural numbers is due to Lawvere, who introduced the notion of natural number object [Law64]. (While iteration was of course well known to any recursion theorist at the time, Lawvere provided an elegant characterisation of it as a universal property.) From this notion, it followed that parametric iteration and also primitive recursion could be expressed in a topos [LS86]. Subsequently, with the study of (co)algebraic datatypes, it was realised that this situation is hardly unique for the functor $(\cdot) + 1$, but actually happens for arbitrary functors which possesses initial algebras and final coalgebras as well. (This was subject to an understanding of (co)completeness, e.g. [SP82].) This in turn gave birth to a plethora of publications within the functional programming community, aiming to develop suitable formal methods for reasoning with such schemes. One notable reasoning principle is initiality and finality itself, which is called *fusion* (see Paper I and Paper III).

Much work has been devoted to generalising (co)iteration further, e.g. [Bar03, Len99, CHL03, UVP01, Ven00], including parametric variants [Mos01, Par02, Par01, Gib93], generalisations to cover bialgebraic structures, etc. All these developments are relevant to the present thesis, although our focus on dialgebras rather than (co)algebras renders the situation more complex in at least one respect: we have to deal with both contravariant and covariant variables, and are also forced

to work in a domain-theoretic universe (or some similar mathematical structure with the algebraic compactness property) since existence of initial/final dialgebras cannot otherwise be granted.

A milestone in the development of categorical recursion schemes was made through Freyd's seminal papers [Fre90, Fre91, Fre92]. Freyd generalised the work by Smyth et al [SP82] and demonstrated that by separating the contravariant and covariant variable, a remarkably powerful mixed variant recursion scheme arises. This scheme was subsequently studied by Pitts [Pit96, Pit94] who developed proof principles based on it. Around that time, Meijer et al were able to provide some examples for functional programming languages, demonstrating that the scheme could be useful [MH95, WW03, FS96]. Some other notable theoretical results are: the reduction to inductive types as given by Freyd [Fre90] and the relationship to dinaturality [Fre91], the derivation of an associated proof principle [Pit94, Pit96], programming examples dealing with higher-order abstract syntax [WW03], lambda calculus interpreters [MH95], and circular datatypes [FS96]. However, Freyd's scheme, called *direcursion* in this thesis, remains relatively unexplored as regards to termination properties and its relationships to other recursion schemes (and programming examples have so far been rather scarce). This thesis, and particularly Paper III, is a step towards improving this situation.

1.2.4 Coalgebraic Semantics

Coalgebraic (or final) semantics can be viewed as being the dual of initial algebra semantics. However, final semantics uses a functor induced by the operational semantics whereas initial algebra semantics is driven by the syntax (and notably by a functor induced by the signature of the language). The duality between coalgebra and algebra stretches quite far. For example, congruence is the dual of bisimulation (a notion due to Park [Par81]), induction the dual of coinduction, and so forth. The coalgebraic approach to semantics of programming languages have been developed by a number of researchers over the years, including Aczel, Rutten, Turi, Lenisa, Honsell, and Jacobs (see e.g. [Len98], which includes a comprehensive survey of final semantics and coalgebraic methods).

Final semantics has its roots in research on set theory, which was (to some extent) motivated by work on process algebra. Forti and Honsell [FH83] and Aczel [Acz88] studied anti-foundation axioms (including AFA aka X_1). The purpose of these axioms was to be able to explain "circular phenomena" such as those that appeared in Milner's process algebra [Mil89]. Circular here refers to self reference that arises from trying to solve equations (or equation systems) of sets. Let us

consider an example:

$$X = \{0, X\}$$

The well-foundedness of the membership relation in Zermelo-Fraenkel set theory means that no such set can exist. Nevertheless, one could argue that infinite "sets" of the form $\{0, \{0, \{0, ...\}\}\}$ could be regarded as solutions. In general, we wish to solve such equations in order to give natural set-theoretic explanations of process algebras (or to generally account for other seemingly "viciously circular" phenomena, see e.g. [BM96]). Therefore, the axiom of foundation was replaced by alternative axioms such as Aczel's AFA (or, equivalently, Forti et al's X_1). The theory of coalgebra developed substantially through the work by Aczel [Acz88], which used a category **Class** where the objects are subclasses of the universe of non-well-founded sets (satisfying AFA), and the morphisms are (tagged) class functions. On this category, he identified two kinds of functors: the standard functors (namely those that preserves weak pullbacks) and the functors which are uniform on maps (in a technical sense). For each of these classes, he proved (in his Final Coalgebra Theorem and Special Final Coalgebra Theorem, respectively) that the functors possess a final coalgebra. In the first case, this amounts to a quotient construction and in the second case, it gives a characterisation of the final coalgebra as a maximal fixpoint of an operator on sets induced by the functor under consideration. Aczel's results sparked much of the development that then followed, including work by Rutten and Turi, which demonstrated that the duality between algebra and coalgebra can be taken quite far, and the extensive study of final semantics carried out by Lenisa [Len98].

It is interesting to note that well-founded sets can be viewed as approximations of non-well-founded sets, the latter thus being construed as limits of the former (c.f. domain theory) [Acz88] (this approach was taken up by Hallnäs [Hal85] and further by Lindström [Lin89] within Martin-Löf type theory). Furthermore, Scott [Sco60] developed a model of Zermelo-Fraenkel set theory without the foundation axiom, which may have inspired his work on domain theory.

Next, we will sketch why these developments are relevant for the semantics of programming languages (such as object calculi). Given a term algebra, which we can regard as the programming language itself, the equational theory over the associated signature induces a quotient algebra, which we call the term model. This term model need not be fully abstract, since the equivalence classes are, after all, just the provably equal terms. However, the observational equivalence may itself be a congruence, and we can then quotient the term model once again to arrive at another, albeit still syntactical, model. However, we will generally not be able to

construct a fully abstract model in this way, since the term model, and consequently also a second quotient with respect to observation equivalence, identifies all programs that diverge. On the other hand, observational congruence will typically distinguish many non-terminating programs, e.g., programs that terminate at two different depths with respect to applicative contexts. Thus, a model constructed in this way will also identify all non-terminating programs and, for the lazy lambda calculus [Abr90] and indeed for typed object calculus, this is not the route one generally wishes to take. Consider, for example, the lazy lambda calculus term **YK**. This should be seen as an "infinite process", and should clearly not be identified with the term $\Omega = \omega\omega = (\lambda x.xx)\lambda x.xx$. These problems can be avoided if we consider a coalgebraic semantics instead.

The final coalgebra (in **Set**) over the signature induced by a given programming language includes infinite terms. By constructing a suitable bisimulation relation, one can quotient this coalgebra to arrive at a fully abstract model. This is done by using a coinductive extension (corecursion) into the final coalgebra for the behaviour functor (which is induced by the operational semantics). This approach is well-suited for object-based programming; previous work by Gordon et al [GR96a, GR96b, Gor98] provides significant progress in this direction. However, this kind of semantics has already been studied comprehensively (by e.g. Lenisa et al [Len96]), and for that reason, we favour other results in this thesis. Coalgebraic semantics can also be combined with domain-theoretic semantics (see e.g. [Abr90]), particularly to obtain interesting fully abstract models. In that context, we are faced with at least two additional problems: 1) a quotient in **Set** need not enrich, i.e., need not be equipped with any (complete) ordering; 2) it is not straightforward to equip the recursive function space datatypes in question with a suitable coalgebra structure. We will return to several of these issues in the further work section of chapter 6.

1.2.5 Domain Theory

Plotkin [Plo85] proposed an alternative approach to domain theory, based on considering cpos without least elements and partial maps on these. He introduced the category **pCPO**, which is better fitted for standard formulation of recursion theory. Around the same time, Moggi et al [LM84] introduced the notion of partial cartesian closed category. Later the notion of Kleisli exponential [Mog89] was introduced as well (apparently not known by that name until the work of Simpson [Sim92]). Subsequently, Fiore wrote a dissertation on axiomatic domain theory [Fio96b] which spelled out enriched notions of algebraic compactness (as induced

by Freyd's work), analysed these in detail, and gave an axiomatic account of eager FPC (i.e. Plotkin's metalanguage with recursive datatypes). In particular, he included a characterisation of a class of order-enriched computationally adequate models of FPC. One novelty in Fiore's work (with respect to Freyd's) is the use of categorical abstraction of partiality (generalising the approach advocated by Plotkin in [Plo85]). For the present thesis, we have not used Fiore's recasting of Freyd's work into enriched category theory, nor have we capitalised on the abstact notion of partiality used by Fiore (which we merely survey very briefly in Paper II). In Paper I, we even fix the ambient category to be **pCPO** of partial continuous maps, although we have as much as possible explicated our assumptions on the universe of discourse, thus making alternative concrete categories viable as well. (To this end, Turi and Rutten [TR98] showed that the category of complete metric spaces and non-expansive maps is algebraically compact with respect to locally contractive functors, i.e., satisfies our most important assumption on the ambient category.) It is possible also to consider an even more abstract approach to domain theory using Kleisli exponentials (with respect to a suitable strong monad), building on Simpson's work [Sim92] as manifested also in more recent work (e.g. [Mø06]) and also discussed in Fiore's dissertation for models of FPC.

1.3 Contributions

The following original technical contributions to the literature are made in this thesis:

1. We develop a generalisation of Freyd's direcursion termed primitive direcursion, which generalises primitive (co)recursion. This recursion scheme provides a new technique for defining elements in models of untyped lambda calculus and is also linked to previous work in (higher-order) bisimilarity, as our examples shows; it is a main result of the present thesis as is its use in the semantics of object-based programming languages. We show that it closes a gap between parametric (co)iterative maps and direcursive maps. These results are presented in chapter five, which consists of a paper that was accepted for the Conference on Algebra and Coalgebra in Computer Science (CALCO 2007), and subsequently appeared in Springer-Verlag, Lecture Notes of Computer Science [Gli07b].

2. We prove a computational adequacy result and give an encoding into Plotkin's typed lambda calculus with recursive types (FPC) under a lazy evaluation

strategy. This result establishes that a program algebra of typed object calculus can be anchored in a computationally adequate model of lazy FPC (or indeed in a categorical axiomatisation of such a model, c.f. work by Fiore and Plotkin [Fio96b, FP94]), since a valid equation in the model will correspond to operationally congruent terms in the language. This paper, which is presented in chapter four, appeared as a technical report [Gli07a] at the School of Computer Science and Communication, Royal Institute of Technology, Sweden.

3. We give a denotational semantics using recursive domain equations, but based on an axiomatised category of partial maps. To the best of our knowledge, this is the first categorical semantics of an object-based programming language and as such contributes with a category-theoretic foundation which can be used in the study of program algebra, program transformations, etc. An important, but non-technical, contribution of this paper is that it, apparently for the first time, identifies and demonstrates the usefulness of Freyd's recursion scheme (direcursion) for object-based programming language models, and thus paves the way for refined programming language designs and formal methods. This was joint work [GG05] with Dr. Neil Ghani and was published in Electronic Notes in Theoretical Computer Science, Elsevier, 2005. It appears as chapter three in this thesis.

4. We provide a first case study with examples that include definition of objects in typed object calculus. These examples also include the encoding of algebraic datatypes as objects (Paper I) and, moreover, a family of other examples of using direcursion and primitive direcursion for defining and reasoning with objects (Paper III).

1.4 Overview

In the second chapter, we provide a background survey which explains the context in which we make our contributions. This chapter also establishes terminology and category theoretic preliminaries.

In chapter three (Paper I), we develop a model of typed object calculus using a domain theoretic category of partial maps. This is similar to the work by Fiore [Fio96b] although we work in a concrete category. Moreover, we develop "wrappers" which give a connection between algebraic datatypes and object-based programs, and similarly for coalgebraic datatypes. As a result, simple datatypes,

as well as bisimulations, carry over to an object calculus model. This is achieved using direcursion. The latter capitalises on the observation that direcursion can be used for defining and reasoning with objects in such models (a topic explored further in the examples given in chapter five).

In chapter four (Paper II), we define a typed object calculus with recursive objects. We have modified a calculus due to Abadi and Cardelli by adding a sum type so that booleans and natural numbers can be represented directly using suitable object types, and by combining recursive types with object types into a single type of recursive objects. This calculus is proven to possess subject reduction (i.e., types are preserved by reduction). The main result in this chapter is a translation of typed object calculus into Plotkin's metalanguage FPC. We show that a lazy evaluation strategy is required for such a translation, and, moreover, that this translation turns every computationally adequate and sound model of FPC into such a model for typed object calculus.

In chapter five (Paper III), we introduce Freyd's notion of dialgebra and the associated recursion principle called *direcursion*. In this chapter we develop primitive direcursion, which extends Freyd's work so that additional maps can more easily be defined. Moreover, we show that primitive direcursion is related to parametric (co)iteration, thus giving a link between direcursion and (co)iteration which will later be used for characterising bisimilarity and defining object-based programs.

The final chapter six summarises our conclusions and briefly lists some topics for further investigation. In particular, such topics include further studies of direcursion (e.g. in order to allow for parameters and to characterise suitable classes of definable maps) and, for typed object calculus, the problem of full abstraction and the addition of a new communication operator.

Chapter 2

Categories and Domains

This chapter provides the theoretical background for the thesis, while also surveying some related work.

2.1 Categories

We will assume knowledge merely of the basic notions of category and functors, and therefore review other relevant concepts from elementary category theory in this section. Details can be found in e.g. [Mac97] on most topics, but notions such as involutory category, dialgebra and difunctor, (co)algebra, which may be less familiar, are introduced here, while notation and terminology is given more generally[1].

Convention 2.1.1. *We will write $|\mathbb{C}|$ for the "set" of objects of a category $\mathbb{C}$, and $\mathbb{C}(A, B)$ for the homset, elsewhere also written $Hom(A, B)$ [Mac97]. The domain and codomain of a morphism (arrow) f in $\mathbb{C}(A, B)$ will also be shown by writing $f : A \to B$, and we write $A \in |\mathbb{C}|$ to say that A is an object in $\mathbb{C}$. As usual, the identity morphism on an object A will be written as A, 1_A, or id_A, omitting subscripts when possible.*

A category $\mathbb{C}$ is *small* if its morphisms can be indexed by a set. In particular $|\mathbb{C}|$ is a set in this case. $\mathbb{C}$ is *locally small* if each homset $\mathbb{C}(A, B)$ (i.e. morphisms from A to B) can be indexed by a set, for each pair of objects $A, B \in \mathbb{C}$. We say that any other category is *large*. Though Mac Lane [Mac97] includes the small

[1] Further references to category theory include [AL91, BW85, BW99, Bor94, Cro93, Joh02, LS86, LS97, LR02, McL92, Pie91, Poi92, Tay99, Oos95, Wal92].

and locally small categories among the large ones, these distinctions are otherwise standard (and regarded as quite independent of the axiomatic set theory under consideration). Most categories that we will consider will be locally small but not small (e.g. the category of sets), although functor categories are sometimes even large. However, if $\mathbb{C}$ is small and $\mathbb{D}$ is locally small, then it follows that the functor category $[\mathbb{C}, \mathbb{D}]$ is locally small. The distinction between small and large categories is needed, since a "comprehension principle" of e.g. Zermelo-Fraenkel set theory [Cie97, Gol96] hinders us to speak of a "set of all sets", as Russell's paradox demonstrates. Some classes formed by this principle will not be sets, but rather proper classes. Hence, there are corresponding precise set-theoretic notions to the above kinds of categories. Large categories such as **Class** (the category of classes and class functions over a suitable set-theory) have been used in previous work [Len96, HL00] for studying free construction principles and recursive type equations, with a view on programming language semantics. There has been work on formally relinquishing the dependence category theory in this way have on an axiomatic set theory. For example, Mac Lane [Mac97] formalises a minimal set theory with a universe, while Lawvere abandons set theory altogether and takes the category of all categories as being the most fundamental notion [Law66]. For this thesis, we stick to the standard distinctions as given above, and assume that the set-theoretic universe is a Zermelo-Fraenkel-like set theory.

In this thesis, we often use locally small categories (such as **Set** and **CPO**). **Set** and **CPO** are examples of *concrete categories*, namely categories $\mathbb{C}$ for which there exists a functor $U : \mathbb{C} \to \mathbf{Set}$ which is faithful. Recall that a functor F is *faithful* if for each $A, B \in |\mathbb{C}|$ and every pair $f, g : A \to B$ $F\ f = F\ g$ implies $f = g$. Similarly, a functor is *full* if when for each $A, B \in |\mathbb{C}|$ and $h : F\ A \to F\ B$ there exists h' in $\mathbb{C}$ such that $h = F\ h'$. Both kinds of functors are closed under composition. We say that a $\mathbb{D}$ is a subcategory of $\mathbb{C}$ if there is a faithful inclusion functor J (and the objects/morphisms in $\mathbb{D}$ are also in $\mathbb{C}$). If such a functor J is full, we say that $\mathbb{D}$ is a *full subcategory*. This means that if $A, B \in \mathbb{C}$, then automatically $\mathbb{C}(A, B) = \mathbb{D}(A, B)$. Thus, a full subcategory of some given category is determined solely by giving its objects.

The category **Cat** has small categories as objects and all functors between these categories as arrows. On the other hand we write **CAT** for the category of all large categories, which also includes the small ones. We will use the standard functor $(\_)^{op}$ on **Cat** which maps a category $\mathbb{C}$ to its opposite (or dual) category $\mathbb{C}^{op}$ (with all arrows reversed, c.f. [Mac97]). For a functor $F : \mathbb{C} \to \mathbb{D}$, i.e., a morphism in **Cat**, we have $F^{op} : \mathbb{C}^{op} \to \mathbb{D}^{op}$, making $(\_)^{op}$ a covariant functor. It takes a morphism $f : A \to B$ to $Ff : A \to B$ since $(F\ f^{op})^{op} = F\ f$.

Left inverses of a morphism are called *retractions*. Morphisms with retractions are known as *split monics*. Some morphims have right inverses (also called *sections*) – the terminology is *split epic*. These are special cases of *monic* morphisms, i.e. morphisms which are left-cancellable, and of *epic* morphisms, namely those that are right-cancellable. (See e.g. [Mac97, McL92].) A morphism which is both split monic and split epic is known as an *isomorphism*. (Though, it suffices to show that it is monic and split epic.) Note that in **Set**, monos (i.e. monic morphisms) are injective functions and epis (i.e. epic morphisms) are surjective functions. This is however not generally the case for arbitrary concrete categories.

Much of this thesis will be based on a particular kind of bifunctor. To avoid confusions that otherwise tend to arise (i.e., reference to a "bifunctor on $\mathbb{C}$" becomes ambiguous), we introduce specific terminology, in the vein of Freyd's associated notions [Fre90]:

Definition 2.1.1 (Difunctor)**.** *We say that a functor $G : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{C}$ is a* difunctor.

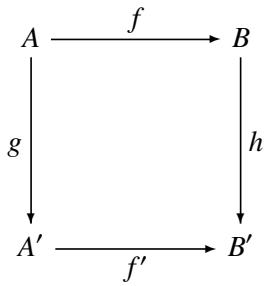
Difunctors satisfy the usual bifunctor properties, but in the mixed variant setting these have the following well-known form [Mac97], restated here for convenience:

Corollary 2.1.1 (Properties of Difunctors)**.** *Suppose G is a difunctor. Then the following is true:*

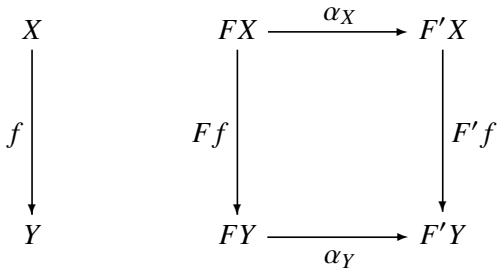
(i) $G(1_A, 1_B) = 1_{G(A,B)}$
(ii) $G(g \circ g', h \circ h') = G(g', h) \circ G(g, h')$
(iii) *if* $g : A \to B$ *and* $h : C \to D$, *then* $G(g, h) : G(B, C) \to G(A, D)$

Note that for $G(A, B) = B^A$ we have for morphisms $G(g, h)(f) = h \circ f \circ g$, which explains the contravariance in this case. For $g : A \to B$ and $h : C \to D$, we therefore have $G(g, h) : G(B, C) \to G(A, D)$.

For any category $\mathbb{C}$, $\mathbb{C}^2$ is the associated *arrow category* whose objects are morphisms f in $\mathbb{C}$, and whose morphisms in $\mathbb{C}^2(f, f')$ are pairs (g, h) of morphisms in $\mathbb{C}$ such that the following diagram commutes:



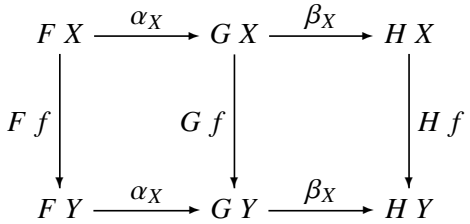
Definition 2.1.2 (Natural Transformation)**.** *Given functors $F, F' : \mathbb{C} \to \mathbb{D}$, a natural transformation $\alpha : F \Rightarrow F'$ consists of a family of arrows $\alpha_X : FX \to F'X$ for $X \in |\mathbb{C}|$ such that for every arrow $f : X \to Y$ in $\mathbb{C}$ the following diagram commutes:*



The morphisms α_X are known as the components of the natural transformation.

For a fixed set S, an evaluation map $e_X : X^S \times S \to X$ is natural (in X). The diagonal map $\Delta_X : X \to X \times X$ is also natural (in X), as is projection $\pi : X \times Y \to X$ (in both X and Y), etc. (These maps will be detailed below.) Many other examples arise in categorical semantics, as [Pit01] shows in detail.

Natural transformations are the morphisms of a category $[\mathbb{A}, \mathbb{B}]$ where the objects are all the functors $F : \mathbb{A} \to \mathbb{B}$. Composition of $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$ is given by $\gamma : F \Rightarrow H$ determined by the following diagram:



That is, the components of γ are the composites of the components of α and β for the same object. This is termed *"vertical" composition* (because the categories are fixed), and written as $\cdot$. In more concise terms, we have that $\gamma_X = (\alpha \cdot \beta)_X = \alpha_X \circ \beta_X$.

We write $[\mathbb{A}, \mathbb{B}]$ (or $\mathbb{B}^{\mathbb{A}}$) for this so-called *functor category* and $\mathrm{Nat}(F, G)$ for a particular homset (which, generally, need not be small). Note that there are identity natural transformations on each functor F, the components of which are the identity morphisms $id_{F(X)}$ on $F\ X$.

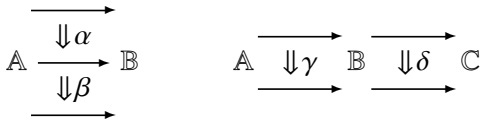
There is also *vertical composition* of natural transformations. Let $F, G : \mathbb{A} \to \mathbb{B}$ and $F', G' : \mathbb{B} \to \mathbb{C}$ be functors on the given categories. For natural transformations $\gamma : F \Rightarrow G$ and $\delta : F' \Rightarrow G'$, we define $\delta \circ \gamma : F'F \Rightarrow G'G$, the vertical composite, by giving the components $(\delta \circ \gamma)_X = G'\gamma_X \circ \delta_{FX} = \delta_{GX} \circ F'\ \gamma_X$, i.e. the diagonal of this square (which commutes because of naturality of δ):

$$\begin{array}{ccc} F'F(X) & \xrightarrow{\delta_{FX}} & G'F(X) \\ {\scriptstyle F'\gamma}\downarrow & & \downarrow{\scriptstyle G'\gamma} \\ F'G(X) & \xrightarrow[\delta_{GX}]{} & G'G(X) \end{array}$$

See e.g. [Mac97] for the proof that this defines a natural transformation.

A special case which is useful for our purposes, is the composition of a natural transformation $\eta : F \Rightarrow G$ for $F, G : \mathbb{A} \to \mathbb{B}$, with a functor $H : \mathbb{C} \to \mathbb{B}$. Then, the functor is regarded as the identity transformation $H : H \Rightarrow H$. We have $\eta H : FH \Rightarrow GH$. Similarly $H' : \mathbb{C} \to \mathbb{A}$ gives $H'\eta : H'F \Rightarrow H'G$. Note as a special case that the composition of a functor with itself can with good conscience be written as FF (or even F^2), and similarly in the general case (as we have indeed done, see below).

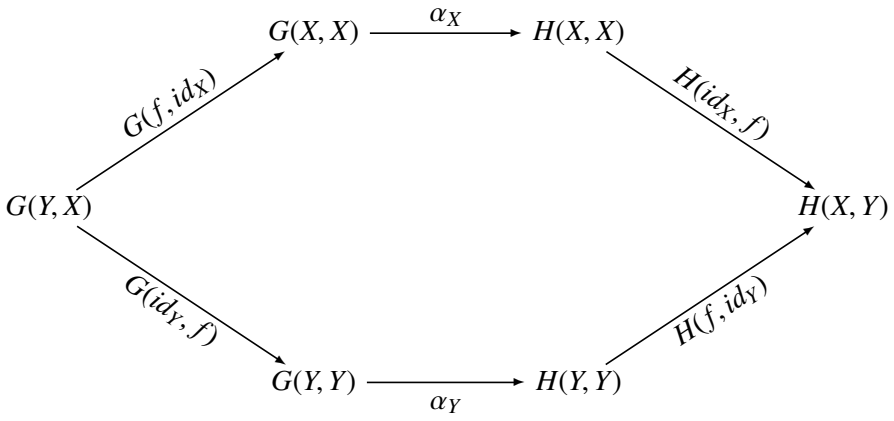
In summary, we have seen two different ways of composing natural transformations, as the following illustration shows:



There is also a weaker notion of "naturality" for functors $G, H : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{D}$:

Definition 2.1.3 (Dinaturality)**.** *Given categories $\mathbb{C}, \mathbb{D}$ and functors $G, H : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{D}$, a* dinatural transformation *$\alpha : G \Rrightarrow H$ is a family of arrows $\alpha_X : G(X, X) \to H(X, X)$ indexed over $|\mathbb{C}|$, such that for every arrow $f : X \to Y$ in*

$\mathbb{C}$*, the following diagram commutes:*



Note that any natural transformation $G \Rightarrow H$, i.e. family of mappings natural in both the contravariant and the covariant component, respectively, is a dinatural transformation (by restricting to the diagonal components), but generally not conversely.

Examples.

- *A fixpoint operator $Y : X^X \to X$ is a dinatural transformation. Simpson [Sim93] showed that for many common domain-theoretic categories, the least fixpoint operator is uniquely characterised as a certain such transformation.*

- *The map eval (defined below) gives the components of an* extranatural transformation*, i.e. a dinatural transformation as above, but where H is dummy in both its arguments, i.e. is a constant object (it is also known as a* wedge*). The general case is the counit of an adjunction with parameters, see [Mac97].*

2.1.1 Adjunctions

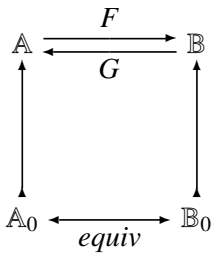
One of the most important concepts that category theory has helped to develop, is possibly the notion of adjunction [Mac97, LS86]. Adjoints occur "throughout mathematics", and aspects of the idea were known before the advent of category theory (loc. cit.).

One way of introducing adjunctions is to consider partially ordered sets (as in [AJ94, LS86, Cro93], and elsewhere), noting that every preorder induces a trivial category with a morphism $f : A \to B$ when $A \leq B$ in the preorder (associativity

and identities follows immediately from transitivity and reflexivity). A functor $F : \mathbb{A} \to \mathbb{B}$ between two preordered sets $(\mathbb{A}, \leq_A)$ and $(\mathbb{B}, \leq_B)$ is a monotonic (order-preserving) mapping because of preservation of composition. In this concrete case, a functor $U : \mathbb{B} \to \mathbb{A}$ is said to be a *right adjoint* to F if we have that the following holds:

$$F(A) \leq_{\mathbb{B}} B \text{ if and only if } A \leq_{\mathbb{A}} U(B)$$

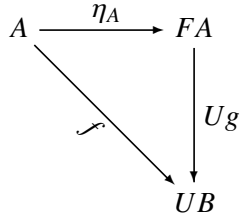
This is classically called a *Galois connection* between the orderings. It follows immediately that $UF : \mathbb{A} \to \mathbb{A}$ is a *closure operation*, namely that $A \leq_{\mathbb{A}} UF(A)$, $UFUF(A) \leq_{\mathbb{A}} UF(A)$, and that UF is again monotone. The dual construction of FU is called an *interior operation* instead. Adjunctions are generalisations of Galois connections, while monads and comonads generalise closure and interior operations, respectively. An important consequence of an adjoint situation as above, is that a one-to-one correspondence arises between the elements $UF(A) \cong A$ and the elements $FU(B) \cong B$ where $\cong$ means that $UF(A) \leq_{\mathbb{A}} A$ and $A \leq_{\mathbb{A}} UF(A)$ and similarly for the other isomorphism (for a partial order, we have therefore equality). This is a principle of "unity of opposites", which determine an equivalence, *equiv*, between said subset of $\mathbb{A}$ and subset of $\mathbb{B}$:



As a final remark, note that if we write $F \dashv U$ for the above situation, and analogously also have $F' \dashv U'$, then it follows that $F'F \dashv U'U$, i.e. Galois connections are closed under composition. (The reader will note a similarity with vertical composition above.) Next, we will generalise and formalise this idea in category theoretic language, particularly by using natural transformations.

Definition 2.1.4 (Adjoint)**.** *Let $\mathbb{A}$ and $\mathbb{B}$ be categories, with $F : \mathbb{A} \to \mathbb{B}$ and $U : \mathbb{B} \to \mathbb{A}$ functors. We say that F is a* left adjoint *to U and write $F \dashv U$, or equivalently that U is a* right adjoint *to F while writing $U \vdash F$, provided there is a natural transformation $\eta : 1_{\mathbb{A}} \Rightarrow UF$ such that for any objects $A \in |\mathbb{A}|$ and $B \in |B|$,*

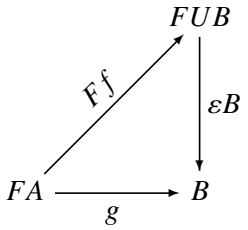
and any morphism $f : A \to UB$, there is a unique morphism $g : FA \to B$ such that



commutes.

This definition is remarkably powerful – note that it states the existence of a family of maps, followed by a universal quantification over maps of certain form, such that a uniqueness property holds. In effect, it gives a translation of maps $f : A \to UB$ into maps $g : FA \to B$, such that g is the unique solution to the above diagram. The data (F, U, η) is called an *adjunction*, and η is said to be the *unit* of the adjunction.

Proposition 2.1.1. *Let F and G be as in the previous definition, i.e. $F \dashv U$ etc. Then there is a natural transformation $\varepsilon : FU \Rightarrow 1_{\mathbb{B}}$ such that for any $g : FA \to B$, there is a unique arrow $f : A \to UB$ such that*



commutes.

For a proof, see e.g. [BW99]. The transformation ε is called the *counit* of the adjunction. When we write $(F, U, \eta, \varepsilon)$ it is understood that η and ε are the unit and counit of $F \dashv U$, respectively.

An important consequence of adjoints (or *adjointness* as it were) is that they establishe a strong bijective correspondence between maps (generalising the preorder situation discussed above):

Proposition 2.1.2 (Adjunction)**.** *We have $F \dashv U$ for functors as in the previous definition (with $\mathbb{A}$ and $\mathbb{B}$ locally small), if and only if there is a bijective correspondence between the homsets*

$$\mathbb{B}(FA, B) \cong \mathbb{A}(A, UB)$$

natural in both A and B.

For the standard proof see e.g. [Cro93] or [Mac97]. Note that the left hand-side gives a functor

$$\mathbb{A}^{op} \times \mathbb{B} \xrightarrow{F^{op} \times 1_{\mathbb{B}}} \mathbb{B}^{op} \times \mathbb{B} \xrightarrow{\mathbb{B}(-,=)} \mathbf{Set}$$

where $\mathbb{B}(-,=)$ is the the mixed variant hom-functor (see [Mac97]). The right-hand side produces

$$\mathbb{A}^{op} \times \mathbb{B} \xrightarrow{1_{\mathbb{A}} \times U} \mathbb{A}^{op} \times \mathbb{A} \xrightarrow{\mathbb{A}(-,=)} \mathbf{Set}$$

analogously, so that we can indeed speak of a natural transformation having an inverse (i.e. *natural isomorphism*).

We will write $\bar{f} : X \to UY$ for the morphism which corresponds to the morphism $f : FX \to Y$. We will analogously write $\bar{g} : FX \to Y$ for the morphism that corresponds to $g : X \to UY$ via the bijection on homsets. We call both of these maps a *transpose*, although there is some ambiguity in this (standard) terminology and notation.

There are a number of important theorems for adjoints (some due to Peter Freyd), which will not be used directly in this thesis, (but see e.g. [Mac97] for (co)limit preservation implications generalising the preorder situation as shown in e.g. [Cro93]). We conclude this section by giving a first example of an adjunction. For this purpose, we consider functors on **Set**, noting that we have a covariant powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ and an inverse image functor $(-)^{-1} : \mathbf{Set} \to \mathbf{Set}$, defined as follows:

$$\begin{array}{lcllcl} \mathcal{P}(X) & = & \{Y \ : \ Y \subseteq X\} & (X)^{-1} & = & \mathcal{P}(X) \\ \mathcal{P}(f)(X) & = & \{f(x) : x \in X\} = f_*(X) & (f)^{-1}(Y) & = & \{x \ : \ \exists y \in Y. f(x) = y\} \end{array}$$

As a consequence, we have both a left and a right adjoint to the inverse image functor $(-)^{-1}$ (these functors are named $\exists$ and $\forall$, respectively). Thus, we have for each function $f : A \to B$ in **Set** also functions $\exists f : \mathcal{P}(A) \to \mathcal{P}(B)$ and $\forall f : \mathcal{P}(A) \to \mathcal{P}(B)$. These functions can explicitly be given as follows:

$$\exists f(A') \ = \ \{f(a) \ : \ a \in A'\} \qquad \forall f(A') \ = \ \{y \in B \ : \ f^{-1}(\{y\}) \subseteq A'\}$$

The more abstract definitions of universal and existential quantification using adjoints, due to Lawvere [Law69], have turned out to be ubiquitous in categorical logic and in topos theory, see e.g. [Joh02, McL92]. This example also illustrates the usefulness of a categorical approach to semantics, like the one pursued for programming languages in this thesis (see [Cro93, Pit01] for further motivation). One

can argue that category theory used in this way abstracts from concrete situations the essence or the character of the mathematical objects involved, and associates reasoning principles to the resulting abstractions that it can express (from adjunctions, universal properties, etc). Used in the appropriate way, category theory thus seems to play an important part of mathematics, particularly for denotational semantics of programming languages, where it can further our understanding of the involved mathematical structures and help us discover novel examples of such. Moreover, there is a close correspondence between category theory and various type theories, since the limits/colimits that a category may have are essentially means by which new types can be constructed from existing types. The latter will be detailed in the following section.

2.1.2 (Bi)Cartesian Closure

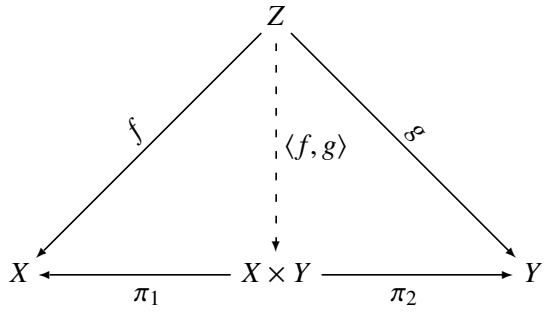
Most interesting categories carry additional categorical structure such as products and coproducts. For example, the familiar category **Set** has a rich categorical structure. Rather than expressing products etc abstractly as (co)limits (surveyed in a later section), we will here review them in more concrete terms, given their importance. For convenience, we fix an arbitrary category $\mathbb{C}$ in this section.

Definition 2.1.5 (Initial and final object, elements)**.** *An* initial object *is an object* $0 \in |\mathbb{C}|$ *such that for any object* $X \in |\mathbb{C}|$*, there exists a unique morphism* $! : 0 \to X$*. Dually, a* final/terminal object 1 *is an object in* $\mathbb{C}$ *for which there exists a unique* $! : X \to 1$ *in* $\mathbb{C}$ *for each object* $X \in \mathbb{C}$*. A morphism* $e : 1 \to X$ *is known as a* global element*, written* $e \in X$*, or* $e \in_1 X$*. An arbitrary morphism* $f : E \to X$ *is known as an E-element, or generalised E-element, written* $f \in_E X$*.*

Note that the global elements in **Set** each correspond to a member of the associated set. Moreover, the initial object 0 is the empty set, and the final object 1 is the singleton (unique up to isomorpism). It is easy to prove that both initial and final objects are unique up to isomorphism in any category.

Definition 2.1.6 (Binary product)**.** *A product of objects X and Y is an object* $X \times Y$ *together with arrows ("projections")* $\pi_1 : X \times Y \to X$ *and* $\pi_2 : X \times Y \to Y$ *such that for any arrows* $f : Z \to X, g : Z \to Y$ *there exists a unique arrow* $\langle f, g \rangle$ *making the*

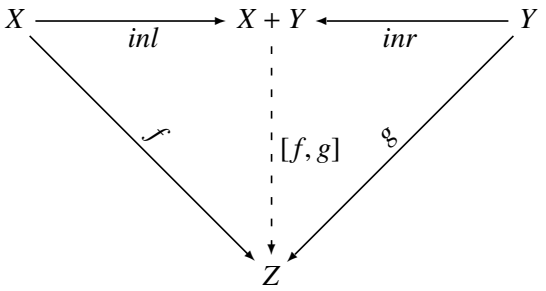
following diagram commute:



If a product exists for any two such objects in $\mathbb{C}$*, we say that* $\mathbb{C}$ has binary products.

Binary products arise precisely when the diagonal functor $\Delta : \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ has a right adjoint, i.e. $\Delta(-) \dashv \Pi(-, A)$, with $\Delta(X) = (X, X)$ (and similarly for morphisms). Note that products are created by the bifunctor $\Pi : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$. The dual situation is the following:

Definition 2.1.7 (Binary coproduct)**.** *A coproduct of objects X and Y is an object X+Y together with arrows ("injections") inl* : $X \to X+Y \to X$ *and inr* : $Y \to X+Y$ *such that for any arrows* $f : X \to Z, g : Y \to Z$ *there exists a unique arrow* $[f, g]$ *making the following diagram commute:*



If a coproduct exists for any two such objects in $\mathbb{C}$*, we say that* $\mathbb{C}$ has binary coproducts.

Coproducts arise precisely when the diagonal functor Δ has a left adjoint, i.e.

$$\mathbb{C}(\Sigma(A, A'), B) \cong \mathbb{C} \times \mathbb{C}((A, A'), \Delta B)$$

natural in A and B). We have from the adjunction $\Sigma(-, A) \dashv \Delta(-)$ a bifunctor $\Sigma : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ which maps two objects to their coproduct. This can also be written

as $\mathbb{C}(A, B) \times \mathbb{C}(A', B) \cong \mathbb{C}(A + A', B)$ using the homsets. More details are given in e.g. [Mac97].

Binary (co)products generalise to the finite case:

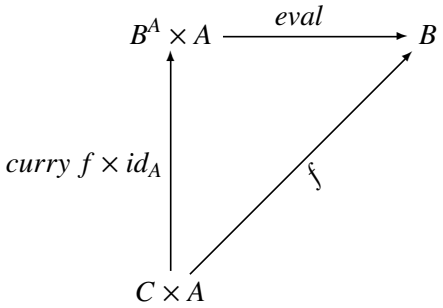
Definition 2.1.8 (Finite products and coproducts)**.** *A category $\mathbb{C}$ is said to* have finite products *if to any finite number of objects $A_1, ..., A_n$ there exists a product diagram as above, but with n projections $\pi_1, ..., \pi_n$, subject to the same universal property. In particular, there is a nullary product, which is the terminal object of the category. Dually, $\mathbb{C}$ is said to* have finite coproducts *if the same holds for coproducts and its universal property. For the latter, we write in_i for the injections, and note that the nullary sum must be an initial object.*

In fact, we have:

Theorem 2.1.1. *Any category $\mathbb{C}$ with binary products and terminal object has finite products, and a category with binary coproducts and initial object has finite coproducts.*

Proof. See [Mac97]. The second statement arises dually. □

Definition 2.1.9 (Exponentials)**.** *Let $\mathbb{C}$ be a category with finite products and terminal object. The exponential of B by A is an object B^A (also written $[A, B]$) together with arrow $eval : B^A \times A \to B$ such that for every $f : C \times A \to B$ there exists an arrow $curry\ f : C \to B^A$ such that the following diagram commutes:*



A category has exponentials *if for any objects A, B the above is the case.*

Definition 2.1.10 (Cartesian closed categories, ccc)**.** *A* cartesian closed category *(ccc) is a category $\mathbb{C}$ with finite products, exponentials and a terminal object. More concisely, $\mathbb{C}$ is a ccc if (1) it has products, and (2) for each $A \in |\mathbb{C}|$ the product functor $- \times A$ has a right adjoint being the exponential $-^A$, where the counit ε is*

the evaluation map. Moreover, we say that a category is bicartesian closed *if it is cartesian closed and has finite coproducts, i.e.* Δ *has both a left and a right adjoint, and* $\times$ *has a right adjoint.*

Note that **Set** and **Cat** are examples of cartesian closed categories.

Remark 2.1.1 (Stable initial object)**.** *Suppose* 0 *is an initial object in* $\mathbb{C}$*. Then it does not generally follow that* $X \times 0$ *is an initial object as well [Pit01]. If this property holds, we say that the category has* stable initial objects*. This is automatically the case for any cartesian closed category (as it follows from the adjunction:* $X\times 0 \to Y \cong X \to Y^0 \cong X \to 0$*), and stable initiality is equivalent to* $\pi_2 : X\times 0 \to 0$ *being an isomorphism for each object* X*.*

Remark 2.1.2 (Zero)**.** *If the initial object* 0 *and the terminal object* 1 *are isomorphic, then this object* $0 \cong 1$ *is said to be a* zero *(elsewhere called also a* biterminator*) and the category is said to be* punctuated*. For such a category* $X \times 0 \cong X$*, so it clearly cannot have stable initial objects unless it is trivial. Such categories will be dealt with in Paper I and Paper III of this thesis.*

One sometimes also speaks about weak products, coproducts and other universal arrows, in which case uniqueness of the the mediating morphism is not required, merely its existence (see [Mac97])

Proposition 2.1.3. *In a bicartesian closed category, there is at most one morphism* $f : X \to 0$ *for any object* X*. Moreover, the following holds:*

$$A + 0 \cong A \qquad A \times 0 \cong 0 \qquad A^0 \cong 1 \qquad A + B \cong B + A$$

$$(A + B) + C \cong A + (B + C) \qquad (A + B) \times C \cong (A \times C) + (B \times C)$$

$$A^{B+C} \cong A^B \times A^C$$

Proof. See e.g. [LS86] for the first part (uniqueness of f). Proofs for the isomorphisms are straightforward. □

The sixth isomorphism above expresses distributivity of binary products over binary coproducts. This is also known as *stable coproducts* [Pit01], which arises in any bicartesian closed category, as it is exactly the property that each functor $A \times (-)$ preserves binary coproducts (and this functor has of course a right adjoint).

Proposition 2.1.4. *The following holds in any cartesian closed category:*

$$A \times 1 \cong 1 \qquad A \times B \cong B \times A \qquad (A \times B) \times C \cong A \times (B \times C)$$

Moreover, we have:

$$A^1 \cong A \qquad 1^A \cong 1 \qquad (A \times B)^C \cong A^C \times B^C \qquad A^{B \times C} \cong (A^C)^B$$

When explicitly needed, we will write $c_{A,B}$ for the isomorphism $A \times B \cong B \times A$, and $a_{A,B,C}$ for the isomorphism for associativity of $\times$ in a ccc (dropping subscripts when we can). Note also that we have naturality in each component of $c_{A,B}$ and $a_{A,B,C}$.

Having surveyed some of the properties of (bi)cartesian closed categories, we note that any suitable category for solving domain equations (in a strict sense made precise in a later section) will necessarily have to be punctuated, although in our case it will be a subcategory of a cartesian closed category. This means that we often have to settle for a weaker structure known as symmetric monoidal category (as explained in e.g. [Mac97]. In other words, we have a tensor, i.e. bifunctor $\otimes$, together with natural transformations for associativity, symmetry, and left and right identity with respect to a given unit object I). In Paper I and and Paper III, we will assume a symmetric monoidal closed category, which is a symmetric monoidal category where in addition the tensor has a right adjoint.

2.1.3 Algebraic Structures

Fix a category $\mathbb{C}$.

Definition 2.1.11 (Algebra, Coalgebra)**.** *Given a covariant functor $F : \mathbb{C} \to \mathbb{C}$, an F-algebra is a pair (A, α) where $\alpha : F\,A \to A$. A is the carrier and α the structure map. The dual notion is that of F-coalgebra, i.e., pairs with a reversed arrow $\alpha : A \to F\,A$. The arrows between (co)algebras are called* F-homomorphisms *(or (co)algebra maps), i.e., arrows h such that for algebra maps the left diagram commutes and for coalgebra maps the right diagram commutes:*

$$\begin{array}{ccc} FA & \xrightarrow{Fh} & FB \\ {\scriptstyle\alpha}\downarrow & F\text{-}\mathbf{Alg} & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[h]{} & B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{h} & B \\ {\scriptstyle\alpha}\downarrow & F\text{-}\mathbf{Coalg} & \downarrow{\scriptstyle\beta} \\ FA & \xrightarrow[Fh]{} & FB \end{array}$$

*Algebras and their homomorphisms form a category F-**Alg** (dually F-**Coalg** for coalgebras). An initial object (when such exists) in F-**Alg** is called an* initial algebra*. We write $(\mu F, \iota_F)$ for such an object. A final object, denoted $(\nu F, \iota_F^\circ)$, in F-**Coalg** category (when such exists) is instead called a* final coalgebra*. (We omit suffixes when we can.)*

Note that algebra structures allow construction of (generalised) elements in the carrier, whereas coalgebra structures allow destruction of them. This has recently led to an intense study of coalgebras as a means for retrieving/observing information from a dynamic system (the information being the "behaviour" provided by the functor F other than the states themselves in the carrier), which has given rise to a research area sometimes referred to as universal coalgebra [Rut00]. Moreover, note that coalgebras are closely connected to labelled transition systems and to process algebras, as explained in loc. cit.

Lemma 2.1.1 (Lambek's Lemma)**.** *An initial algebra $(\nu F, \iota_F)$ is an isomorphism $\nu F \cong F(\nu F)$ (called fixpoint or invariant). Dually for the final coalgebra $(\nu F, \iota_F^\circ)$.*

Proof. The proof, credited to Joachim Lambek, is based on noting that ι_F is also a F-homomorphism:

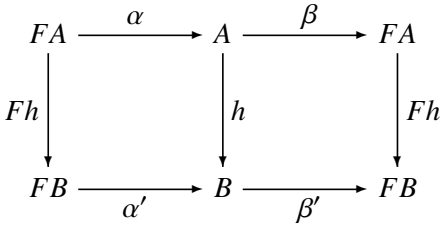
$$\begin{array}{ccccc} F\mu F & \xrightarrow{F!} & F^2\mu F & \xrightarrow{F\iota_F} & F\mu F \\ \downarrow{\scriptstyle \iota_F} & & \downarrow{\scriptstyle F\iota_F} & & \downarrow{\scriptstyle \iota_F} \\ \mu F & \overset{!}{\dashrightarrow} & F\mu F & \xrightarrow[\iota_F]{} & \mu F \end{array}$$

We have the unique homomorphism ! into $(F\mu F, F\iota_F)$. But when we compose it with ι_F, we have $\iota_F \circ ! = id$ because id is the unique homomorphism from the initial algebra to itself. For the other direction, note that the square for ! gives $! \circ \iota_F = F\iota_F \circ F! = F(\iota_F \circ !) = F(id) = id$. This establishes that ! is a two-sided inverse and therefore ι_F is an isomorphism. We write ι_F^{-1} for ! therefore. □

The categorical notions of algebra/coalgebra can be generalised in several directions. Work by Turi and Plotkin [TP97] and Freyd [Fre90] has shown that the following generalisations are fruitful:

Definition 2.1.12 (Bialgebra)**.** *For endofunctors F, G, a $\langle F, G\rangle$-*bialgebra *is a triple (A, α, β) where (A, α) is an F-algebra and (A, β) is an G-coalgebra. We also have*

a category $\langle F, G\rangle$-**Bialg** *whose objects are bialgebras, and whose morphisms* h *are morphisms* $h : A \to B$ *between carriers of bialgebras* (A, α, β) *and* (B, α', β') *such that the following diagram commutes:*

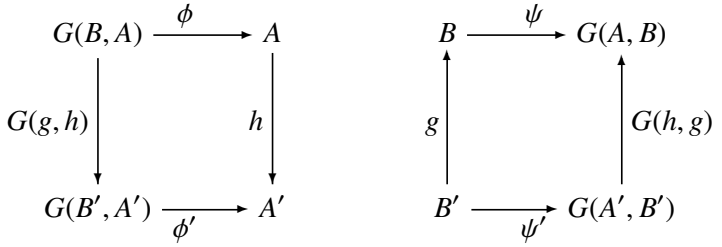


The notion of bialgebra given above has been studied also in forms where the two structure maps are dependent on each other, see loc. cit. for the notion of λ-bialgebra, and also [CHL03] for a further generalisation. (In this work, conditions for when F lifts to a coalgebra functor (and dually) has been studied, so that either map becomes a homomorphism. Sufficient conditions for this involve distributive laws [Bec69], hence the notation λ.) We will use bialgebras of the more simple form above when we give categorical semantics. We will also use the following generalisation of (co)algebra due to Freyd (loc. cit):

Definition 2.1.13 (Dialgebra [Fre90])**.** *A* G-dialgebra *for bifunctor* $G : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{C}$ *is a quadruple* (A, B, ϕ, ψ) *of objects* A, B *and associated arrows* $\phi : G(B, A) \to A$ *and* $\psi : B \to G(A, B)$*.*

Note that in the case when G is dummy in its contravariant argument (i.e., an endofunctor F on $\mathbb{C}$), this definition gives precisely that (A, ϕ) is an F-algebra and, independently, that (B, ψ) is a F-coalgebra. Dialgebras for a bifunctor G form a category G-**Dialg** with the following morphisms:

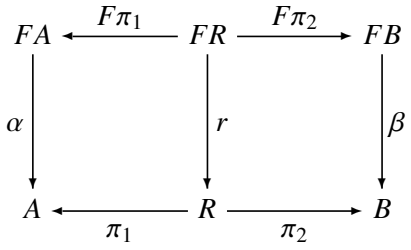
Definition 2.1.14 (Dialgebra map [Fre90])**.** *Given* G*-dialgebras* (A, B, ϕ, ψ) *and* (A', B', ϕ', ψ')*, a* G-homomorphism *(or* dialgebra map/dimap*) is a pair of arrows* $(h : A \to A', g : B' \to B)$ *such that the following diagrams commute:*



More details about dialgebras are given in section 2.4.3, as well as in Paper I and Paper III.

Finally, the notion of congruence and bisimulation on, respectively, an F-algebra or F-coalgebra are important for this thesis. The former states that a relation is compatible with the structure map of an algebra, and the latter similarly for a coalgebra. Here lies a problem, since we have not said what a relation over a pair of objects is. For the category of sets, the required relations would be $R \subseteq \mu F \times \mu F$ or $R \subseteq \nu F \times \nu F$, but in other categories, such as domain theoretic (or order-enriched) categories, only "approximable" relations, arises as subobjects, i.e. monics $r : R \rightarrowtail \mu F \times \mu F$ (which are subject to Scott-continuity etc). As we will work in an enriched setting, where sets underlie cpos, we here settle for **Set**-theoretic relation rather than insisting on the relations themselves being part of the same ambient category (like in [TR98], for example). This implies a certain stratification between proof principles and our universe of discourse (c.f. Pitts work [Pit96]). (One might want to close this gap in the future: for a discussion see e.g. [Fio96a], and for some progress towards it, see [Fio96c].)

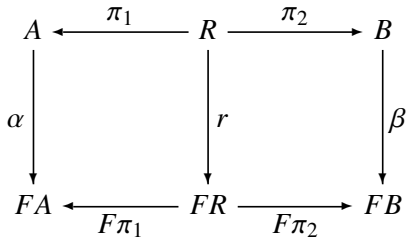
Definition 2.1.15 (Congruence)**.** *Given an endofunctor F : **Set** $\rightarrow$ **Set** and F-algebras (A, α) and (B, β), an F-*congruence relation *between (A, α) and (B, β) is a relation $R \subseteq A \times B$ such that there exists another F-algebra (R, r) that make the following diagram commute (with projections restrictions of those on the product):*



Note that this says that the projections are F-homomorphisms, as indicated.

Definition 2.1.16 (Bisimulation)**.** *Given an endofunctor F : **Set** $\rightarrow$ **Set** and F-coalgebras (A, α) and (B, β), an F-*bisimulation relation *between (A, α) and (B, β) is a relation $R \subseteq A \times B$ such that there exists another F-coalgebra (R, r), making the following diagram commute (where π_1 and π_2 are the restrictions of the usual*

projections for the product):



*Note that this says that the projections are F-homomorphisms as indicated. The greatest such relation R on a single F-coalgebra is called F-*bisimilarity*, if it exists.*

The notion of bisimulation gives a technique for proving that two states (elements of carriers) are bisimilar, i.e. contained in the greatest bisimulation (any bisimulation obviously being sufficient for this to happen). Rutten [Rut00] establishes many results regarding coalgebras and bisimulations in **Set**. His work was predated by the development of process algebra (notably the work of Robin Milner), coalgebra [Acz88] and applicative bisimilarity [Abr90] (on domain-theoretic categories), see loc. cit. and [TR98] for some references and more detailed history.

2.2 Limits and Colimits

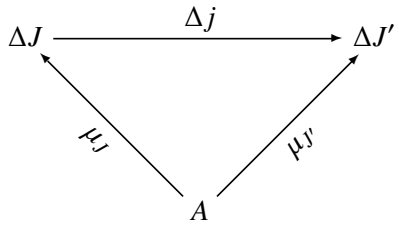
Limits (classically called inverse or projective limits) and colimits (direct or inductive limits) generalise many important constructions. The present thesis rests on previous results by e.g. Smyth and Plotkin, and by Scott in the context of solving recursive domain equations. These constitute theoretical machinery which implies existence of the domain-theoretic models studied in the thesis. Also, limits are the categorical counterpart of the algebraic concept of an equationally specified subset of a product [BW99] and can be defined as such (i.e. by using equalisers). Alternatively, they can be concisely defined using (co)universal arrows. Here, we will examine a more direct approach which is equivalent and more concrete.

To begin with, say that a functor $\Delta : \mathbb{J} \to \mathbb{C}$ is a $\mathbb{J}$-diagram where $\mathbb{J}$ is the index category (or shape of the diagram), although in fact it suffices to use graph morphisms as in e.g. [AC98].

Definition 2.2.1 (Cone, cocone)**.** *A* cone *under a diagram $\Delta : \mathbb{J} \to \mathbb{C}$ from A is a pair (A, μ) where μ is a natural transformation $\mu : K_A \Rightarrow \Delta$ where K_A is a constant functor with $K_A X = A$, and $A \in |\mathbb{C}|$. The dual notion of* cocone *is as follows: a*

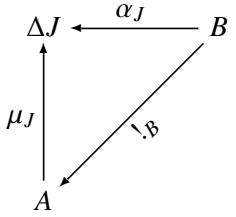
cocone over a diagram $\Delta : \mathbb{J} \to \mathbb{C}$ *to* A *is a pair* (A, ν) *consisting of* $A \in |\mathbb{C}|$ *and* $\nu : \Delta \Rightarrow K_A$.

The components of the respective natural transformations are often called the *legs* of the (co)cones. The constant functor K_A determines the *vertex* A of the cone, and for each $J \in |\mathbb{J}|$, a component $\mu_J : A \to \Delta J$ gives a morphism in $\mathbb{C}$ from the vertex. Moreover the definitions mean that, for any morphism $\Delta j : \Delta J \to \Delta J'$, we have that the leg μ_J into ΔJ, the leg $\mu_{J'}$ into $\Delta J'$, and Δj form a commuting diagram:



Definition 2.2.2 (Universal cone and cocone)**.** *A cone* (A, μ) *under* Δ *is a* limiting cone *(also universal cone) if for each other cone* $\alpha : \Delta \to K_B$*, there exists a unique mediating morphism* $!_B : B \to A$ *such that for each* $J \in \mathbb{J}$ *we have* $\mu_J \circ !_B = \alpha_J$. *A cocone* (A, ν) *over* Δ *is a* colimiting cocone *(also universal cocone), if for each cocone* $\beta : K_B \to \Delta$*, there exists a unique mediating morphism* $!_B : A \to B$ *such that for each* $J \in \mathbb{J}$ *we have* $\beta_J \circ !_B = \nu_J$.

The following diagram shows the situation for the universal cone, for any given pair of projections μ_J and α_J as in the definition, with $J \in \mathbb{J}$:



This clearly amounts to saying that the universal (co)cone is the initial object in what obviously is a category of (co)cones. Moreover, the universal cone can be viewed as the "greatest" cone and the universal cocone as the "least". This intuition comes from considering a preorder category where a morphism $e : A \to B$ witnesses that $A \leq B$. Hence, a universal cone is a cone such that any other cone is "smaller" (and dually for cocones).

Definition 2.2.3 ((Co)limit)**.** *When it exists, the limit* $\varprojlim \Delta$ *of a diagram* Δ *is given as the vertex A of the universal cone* (A, μ)*. When it exists, the colimit* $\varinjlim \Delta$ *is the vertex* A' *of the universal cocone* (A', ν)*.*

The construction, which implicitly selects initial and terminal objects of a (co)cone category, gives immediately the following:

Proposition 2.2.1 (E.g. [Mac97])**.** *Limits and colimits are unique up to isomorphism.*

Definition 2.2.4 ((Co)completeness)**.** *We say that a category is* complete *if it has limits for all diagrams* Δ *(and small-complete if this is true when* $\mathbb{J}$ *is a small category). Dually, we say that a category is* cocomplete *if it has the colimits.*

Note that we have to be careful with regards to the size of $\mathbb{J}$. Unless we say otherwise, we assume (as is standard) that it is a small category (but for certain coalgebraic semantics large shapes must also be considered). Even for small diagrams, completeness is quite a strong property, which implies that the category has arbitrary products (i.e. with respect to an arbitrary diagonal operator, and dually so for cocompleteness and coproducts), see [Mac97]. We have spoken about functors preserving products. Now we can express the general notion:

Definition 2.2.5 ((Co)continuous functor)**.** *Fix a category* $\mathbb{C}$ *and shape* $\mathbb{J}$*. If for all functors* $\Delta : \mathbb{J} \to \mathbb{C}$ *we have*

$$F \varprojlim \Delta = \varprojlim F\Delta,$$

then we say that F preserves limits of shape $\mathbb{J}$*. If for all such* Δ *we have*

$$F \varinjlim \Delta = \varinjlim F\Delta,$$

then we say F preserves colimits of shape $\mathbb{J}$*. If F preserves all limits for all* $\mathbb{J}$ *it is said to be* continuous*. If it similarly preserves all colimits for all* $\mathbb{J}$*, it is said to be* cocontinuous*.*

Initial F-algebras arise from certain colimits under suitable assumptions on the involved functor and categories. The first assumption is that $\mathbb{J} = \omega$, i.e. the ordinal ω considered as a category, i.e. just a chain $0 \to 1 \to 2...$ (plus identities).

Convention 2.2.1. *We will write* $(D_i, d_i)_{i \in \omega}$ *for a diagram* $\Delta : \omega \to \mathbb{C}$*, where* $d_i : D_i \to D_{i+1}$*, for which* d_i *are morphisms in* $\mathbb{C}$ *(see [SP82]). We also write* $d_{m,n}$ *for the composite* $d_{n-1} \circ ... \circ d_m : D_m \to D_n$*. A diagram* $\Delta : \omega \to \mathbb{C}$ *is obviously induced by this data.*

The main idea is outlined as follows:

$$F \underset{\longrightarrow}{\mathrm{Colim}}\, \Delta \cong \underset{\longrightarrow}{\mathrm{Colim}}\, F\Delta = \underset{\longrightarrow}{\mathrm{Colim}}\, \Delta^{-} = \underset{\longrightarrow}{\mathrm{Colim}}\, \Delta$$

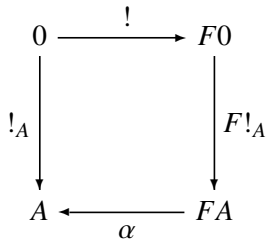
First, $\Delta : \omega \to \mathbb{C}$ is assumed, and we construct Δ by iterating the functor on the initial object, giving $0 \xrightarrow[!]{} F(0) \xrightarrow[F!]{} F^2(0)$, and so forth. Next, we apply the functor to the limit of such a diagram (which we must assume exist in the category $\mathbb{C}$). Next, we must use a "commutativity" property (preservation of limits), after which a basic observation about manipulating the diagram gives the fixpoint. It remains to show that the fixpoint is the initial one. Now, we give the proofs in detail.

These proofs are due to Smyth and Plotkin [SP82], based in turn on work by Wand [Wan79]. (Both these papers stem from work by Scott [Sco72] on models of untyped lambda calculus using special colimits.)

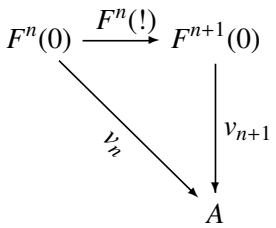
Lemma 2.2.1 ([SP82])**.** *Let $F : \mathbb{C} \to \mathbb{C}$ be a functor where $\mathbb{C}$ is a ω-cocomplete category. Let Δ be the ω-chain $(F^i(0), d_i = F^i(!))_{i\in\omega}$ where $!$ is the unique morphism $0 \to F(0)$. Then, the following holds:*

- *Every F-algebra (V^A, α) gives a cocone (A, v^A) over Δ.*

Proof. First, we have the diagram



which trivially commutes. We define $v_0 =!_A$, $v_1 = \alpha \circ F(!_A)$ (we drop the subscript for v temporarily) and by induction $v_{n+1} = \alpha \circ F(v_n)$. We prove by induction that each diagram of the following form commutes:



We have for $n + 2$:

$$\begin{aligned} & v_{n+2} \circ F^{n+1}(!), \text{ by definition} \\ &= \alpha \circ F(v_{n+1}) \circ F^{n+1}(!) \\ &= \alpha \circ F(v_{n+1} \circ F^{n}(!)), \text{ by I.H.} \\ &= v_{n+1} \end{aligned}$$

□

It remains to prove that there is a unique F-homomorphism given from the universal cone, thus giving an initial F-algebra. This requires a suitable mediating morphism, and thus some additional notation becomes handy:

Convention 2.2.2. *For $\Delta = (D_i, d_i)_{i\in\omega}$ an ω-chain, we will write Δ^- for the ω-chain given by removing the first object and morphism in the chain, i.e. $\Delta^- = (D_{i+1}, d_{i+1})_{i\in\omega}$. For such Δ, we write $F\Delta$ for $(FD_i, Fd_i)_{i\in\omega}$. For a cocone (A, μ) we write (A, μ^-) for $\mu^- : \Delta^- \Rightarrow K_A$ with component μ_{i+1} at the object i of $\mathbb{J}$. Note, finally, that $(FA, F\mu)$ is a cocone over $F\Delta$, where $F\mu : F\Delta \Rightarrow FK_A = K_{FA}$ for endofunctor F on $\mathbb{C}$.*

Theorem 2.2.1 ([SP82])**.** *Let $F : \mathbb{C} \to \mathbb{C}$ be a ω-cocontinuous functor where $\mathbb{C}$ is a ω-cocomplete category. Let Δ be the ω-chain $(F^i(0), d_i = F^i(!))_{i\in\omega}$ where $!$ is the unique morphism $0 \to F(0)$. Then the following holds:*

- *The colimiting cone (A, μ) yields existence of an initial F-algebra (C, ι_F) where $\iota_F : FA \to A$ is the mediating morphism from $(FC, F\mu)$ to (C, μ^-).*

Proof. Let (A, v^A) and (B, v^B) be the cocones arising from arbitrary F-algebras (A, α) and (B, β), as in the previous theorem. Suppose we have a F-homomorphism $h : (A, \alpha) \to (B, \beta)$. We will show that we then have a mediating morphism $v^A \to v^B$. We proceed by induction on a natural number n, to show that for all $i \in \omega$ we have $v^B_i = h \circ v^A_i$ as required. For $n = 0$ we have $v^B_0 = !_B$ so by initiality the statement holds. For $n + 1$ we have

$$\begin{aligned} & h \circ v^A_{n+1}, \text{ by definition of } v^A_{n+1} \text{ in the previous theorem} \\ &= h \circ \alpha \circ F(v^A_n), \text{ by homomorphism property of } h \\ &= \beta \circ F(h) \circ F(v^A_n) \\ &= \beta \circ F(h \circ v^A_n), \text{ by I.H.} \\ &= \beta \circ F(v^B_n), \text{ by the definition of } v^B_{n+1} \text{ in the previous theorem} \\ &= v^B_{n+1} \end{aligned}$$

Our candidate for the initial F-algebra is the mediating morphism (and indeed F-algebra) ι_F from $(FC, F\mu)$ to (C, μ^-). Let (B, ν^B) be the cocone from a given F-algebra (B, β) as in the previous theorem. We have shown that any homomorphism also gives a mediating morphism. Since F is ω-cocontinuous, also $(FC, F\mu)$ is colimiting, and uniqueness follows. It remains to show that there *exists* such an F-homomorphism $h : (C, \iota_F) \to (B, \beta)$. Let h be mediating morphism from (C, μ) to (B, ν^B). We will show that $h \circ \iota_F$ and $\beta \circ Fh$ are both mediating morphisms from $(FC, F\mu)$ to $(FB, (\nu^B)^-)$, since the required F-homomorphism property then follows. For the first case, $h \circ \iota_F \circ F\mu_i = \nu^B_{i+1}$ by definition of h. For the second case:

$$\begin{aligned}
&\beta \circ Fh \circ F\mu_i \\
&= \beta \circ F(h \circ \mu_n), \text{ by } h \text{ being mediating morphism} \\
&= \beta \circ \nu^B_n, \text{ by definition of } \nu^B \text{ from previous theorem} \\
&= \nu^B_{n+1}
\end{aligned}$$

This completes the proof. □

To summarise: it is established that the carrier μF (= C above) of the initial F-algebra $(\mu F, \iota_F)$ is constructible as a certain colimit, and that the unique homomorphism to some other F-algebra arises as the unique mediating morphism to the respective cocone for that F-algebra. We close the section by dualising these important results:

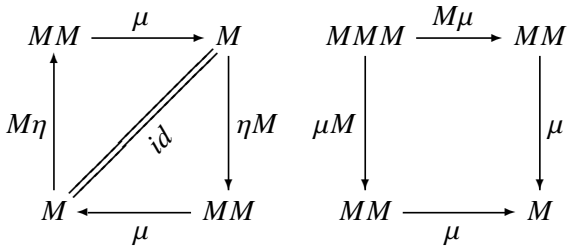
Corollary 2.2.1. *A ω-cocomplete category has final F-coalgebras. Their carriers are the limits of shape ω^{op} given by iteration F on* 1. *Their structure maps $\iota^\circ_F : \nu F \to F(\nu F)$ are given by the mediating morphism from ν^- to $F(\nu)$.*

These results for initial F-algebra and final F-coalgebra can readily be seen as generalisations of the existence of a least fixpoint of a (suitably) continuous function on a (semi-) complete lattice, which dualises also in the lattice theoretic case (this is due to Tarski, see [Tar55, DP02]). For lattices, we require existence of least element (here initial object), suitable supremum (here ω-(co)completeness), and preservation of suprema (infima), i.e. continuity (here (co)complete functor).

2.2.1 Monads and Comonads

Recall the closure operation for UF in the setting of preorder categories. The notion has a generalisation, which, in category theory, corresponds to a monoid at the level of functors and natural transformations:

Definition 2.2.6 (Monad and comonad)**.** *A* monad $M = (M, \eta, \mu)$ *where* $M : \mathbb{C} \to \mathbb{C}$ *is an endofunctor and* $\eta : 1_{\mathbb{C}} \Rightarrow M$ *and* $\mu : MM \Rightarrow M$ *are natural transformations, such that identity/unit and associativity laws hold, i.e. the following diagrams commute:*



A comonad $M = (M, \varepsilon, \delta)$ *is subject to the dual commutative diagrams for natural transformations* $\varepsilon : M \Rightarrow 1_{\mathbb{C}}$ *and* $\delta : M \Rightarrow MM$*.*

The notion of monad finds its origins in homological algebra (see [Mac97]). It was studied by Moggi [Mog88], and subsequently became very popular throughout programming language research and functional programming, since it gives a structuring principle for programs based on the "effects" that they produce. Examples of monads are the trivial identity monad *id*, but also the state monad $S_A(X) = A \multimap (A \times X)$, which we next describe in more detail. Given two maps $f : X \to S_A(Y)$ and $g : Y \to S_A(Z)$, we can uncurry them into $\bar{f} : A \times X \to A \times Y$ and $\bar{g} : A \times Y \to A \times Z$, which case we see that $g \bullet f = \overline{\bar{g} \circ \bar{f}}$, using transpose again to recover the curried map $X \to S_A(Z)$. This explains how μ is defined for S_A, i.e. via the so-called Kleisli composition (written as $\bullet$, see below). The unit η is determined by the mapping $X \mapsto A^{A \times id_X}$, which is easily seen to define a natural transformation $id \Rightarrow S_A$. This monad is particularly important since it can be used to pass a parameter through a computation while also allowing that parameter to be updated.

There is a close connection between (co)monads and adjunctions, as every adjunction $F \dashv U$ induces both a monad and a comonad in the following way: $(UF, \eta, U\varepsilon F)$ and $(FU, \epsilon, F\eta U)$ where η and ϵ are the unit and counit of the adjunction, respectively. For details, see e.g. Barr et al [BW85].

2.2.2 Kleisli and Eilenberg-Moore Categories

Each monad M on a category $\mathbb{C}$ determines two important categories: the Kleisli category $\mathbb{C}_M$ and the Eilenberg-Moore category $\mathbb{C}^M$. We define these in turn:

Definition 2.2.7 ((co)Kleisli category)**.** *Given a monad* M *on a category* $\mathbb{C}$*, let* $\mathbb{C}_M$ *denote the category with objects all the objects in* $\mathbb{C}$*, and* $\mathbb{C}_M(A, B)$ *containing*

all arrows in $\mathbb{C}(A, M(B))$*. Given* $f \in \mathbb{C}_M(A, B)$ *and* $g \in \mathbb{C}_M(B, C)$*, we define the composite* $g \bullet f$ *to be the map* $\mu_C \circ Mg \circ f$ *from* $\mathbb{C}$*. Note that identities on an object* A *are given as certain maps* $A \to M(A)$*, namely as* η_A*. It immediately follows that* $\mathbb{C}_M$ *is a well-defined category, since* $\bullet$ *inherits associativity from the monad* M*. The* co-Kleisli category *arises dually for a comonad (for which we use the same notation).*

Note that the Kleisli category arises from the bijection between monads and adjunctions, since an adjunction gives a translation of a homset $\mathbb{C}(FA, FB)$ into $\mathbb{C}(A, UFB)$, in particular (see [BW85]).

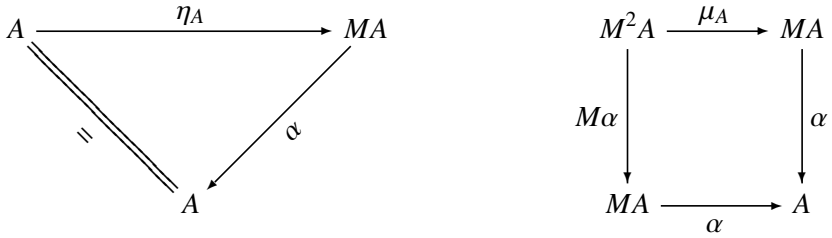
Each monad defines a lifting functor $\widehat{(\cdot)} : \mathbb{C} \to \mathbb{C}_M$ by $\widehat{X} = X$ and $\widehat{f} = \eta_B \circ f$ for any $f : A \to B$, where indeed $\widehat{f} : A \to M(B)$ is an arrow in $\mathbb{C}_M$. Similarly there is always a forgetful functor $U : \mathbb{C}_M \to \mathbb{C}$ which is M on objects (i.e. $U(X) = M(X)$) and maps an arrow $f : A \to M(B)$ to $U(f) : M(A) \to M(B)$. It is immediate that these constructions give functors. Moreover, it follows that $\widehat{\cdot} \dashv U$, i.e. the lifting is the left adjoint of the forgetful functor. This has an important consequence, which we will use below:

Corollary 2.2.2. *Let* $\mathbb{C}$ *be a category and* M *a monad on* $\mathbb{C}$*. Suppose* I *is the colimit of the diagram* Δ *in* $\mathbb{C}$*. Then* $\widehat{I}$ *is a colimit of the diagram* $\widehat{\Delta}$ *in* $\mathbb{C}_M$*.*

For a proof of this, and more details about the above topics, see e.g. Mac Lane [Mac97].

There is a close connection between algebras (actually Lawvere theories) and monads, at least in the first-order case [RB86]:

Definition 2.2.8 (Eilenberg-Moore category)**.** *Each monad* $M = (M, \eta, \mu)$ *with* $M : \mathbb{C} \to \mathbb{C}$ *gives rise to a category of algebras, known as the* Eilenberg-Moore category*. The objects of this category, denoted* $\mathbb{C}^M$*, are pairs* (A, α) *with* α *an* F*-algebra, subject to the commutativity of the following diagrams:*



Morphisms in $\mathbb{C}^M$ *are usual* F*-homomorphisms. Note that composition and identities are given as in* M-**Alg**.

For M-**Alg**, there is a forgetful functor $U^M : M\text{-}\mathbf{Alg} \to \mathbb{C}$ mapping an M-algebra (A, α) to the object O, and homomorphisms to the underlying morphism in $\mathbb{C}$. Such a forgetful functor obviously also exists for $\mathbb{C}^M$ as $U^M : \mathbb{C}^M \to C$ as it is a full subcategory of M-**Alg**. In the second case, there is a left adjoint $F^M : \mathbb{C} \to \mathbb{C}^M$ which maps an object (carrier) to the associated *free (Eilenberg-Moore) algebra*. It is defined as follows:

$$\begin{array}{lcl} F^M(A) & = & (MA, \mu_A) \\ F^M(f) & = & Mf \end{array}$$

Note that for $f : A \to B$, we have indeed that $F^M(f)$ is an M-homomorphism in M-**Alg** as the following diagram shows

$$\begin{array}{ccc} MA & \xrightarrow{\mu^A} & M^2A \\ {\scriptstyle Mf}\downarrow & & \downarrow{\scriptstyle M^2f} \\ MB & \xrightarrow[\mu_B]{} & M^2B \end{array}$$

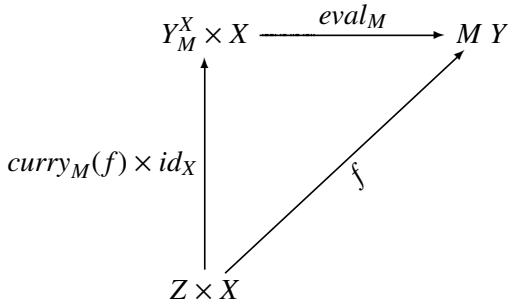
The above adjunction $U^M \dashv F^M$ determines M via the definition of the monad associated to an adjunction in the previous section. (In fact, every monad can be defined by a pair of adjunctions [Mac97].)

2.2.3 Kleisli Products and Exponentials

While the above notions related to monads are well known in the category theorists toolbox, the following is a more recent development, that has turned out to be useful in the semantics of programming languages. It is a notion of function space parameterised by a monad. More precisely, we say that a category $\mathbb{C}$ has *Kleisli exponentials* if for each $X \in |\mathbb{C}|$ the functor $F_M \circ (- \times X) : \mathbb{C} \to \mathbb{C}_M$ has a right adjoint $(-)^X_M : \mathbb{C}_M \to \mathbb{C}$. Alternatively, we formulate this in more elementary terms:

Definition 2.2.9 (Kleisli exponentials [Mog89, Mog91, Sim92, FP94, Fio96a])**.** *The* Kleisli exponential *of Y by X is an object Y^X_M (also written $[X \multimap_M Y]$) together with arrow $eval_M : Y^X_M \otimes X \to Y$ such that for every $f : Z \times X \to M\ Y$ there exists*

a unique arrow $curry_M(f) : Z \to Y_M^X$ *such that the following diagram commutes:*



Note that having Kleisli exponentials is a weaker property than having exponentials, since we can take $M = 1_{\mathbb{C}}$ as a special case, yielding the usual exponential on $\mathbb{C}$. However, in this thesis we postulate axioms (satisfied by several existing concrete categories) such that cartesian closure cannot hold, but the above Kleisli exponentials are available instead (via partiality/lifting monads).

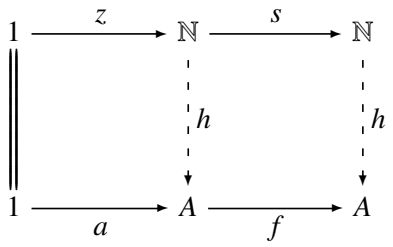
Note that the free functor $\widehat{(-)} : \mathbb{C} \to \mathbb{C}_M$ preserves colimits (including F-algebras if these are given by colimits). In particular $\mathbb{C}_M$ has coproducts, pullbacks, coequalisers etc.

2.3 Recursion and Corecursion

Initiality of an F-algebra gives both *existence* of inductive extensions and *uniqueness* of these. The former is useful for making definitions using induction, whereas the second gives a proof principle. The situation is dual for coinduction/coalgebra. This was realised already by Lawvere, to whom we owe the following notion (and much more):

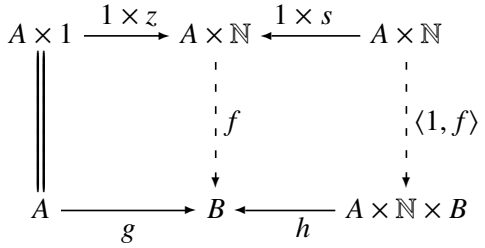
Definition 2.3.1 (Natural number object, nno[Law64])**.** *A* natural number object $\mathbb{N}$ *in a cartesian closed category* $\mathbb{C}$ *consists of two morphisms* $z : 1 \to \mathbb{N}$ *and* $s : \mathbb{N} \to \mathbb{N}$ *such that for each pair of morphisms* $a : 1 \to A$ *and* $f : A \to A$ *with for* $A \in |\mathbb{C}|$*, their exists a unique* $h : \mathbb{N} \to A$ *such that the following diagram*

commutes:



When one merely wishes to make *iterative* definitions in a category, it suffices with a so-called *weak nno* [LS86]. Such objects $\mathbb{N}$ are central in topos theory[2] [LS86, McL92, Joh02], where they automatically generalise to covering definitions by "primitive recursion" by using merely cartesian closure:

Theorem 2.3.1 (Primitive recursion using nno)**.** *Let (N, z, s) be a natural number object in a cartesian closed category. Then, given objects A and B equipped with with morphisms $g : A \to B$ and $h : A \times \mathbb{N} \times B \to B$, there exists a unique $f : A \times \mathbb{N} \to B$ such that the diagram*



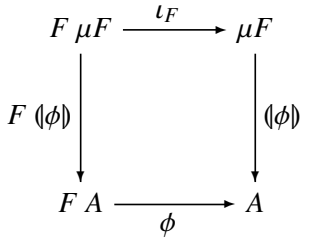
commutes.

Proof. See e.g. [LS86] or [Joh02]. □

Next, we show how the above notions generalise to datatypes other than the natural numbers. For this, consider an arbitrary endofunctor $F : \mathbb{C} \to \mathbb{C}$ which has an associated initial F-algebra $(\mu F, \iota_F)$.

[2] A *topos* has a somewhat richer structure than the categories in this thesis, since it in addition to cartesian closure has a notion of subset (*subobject classifier). The question how topos theory fits with domain theory is, and has been, subject to a lot of research, e.g. synthetic domain theory [Hyl82, Tay91, Hyl92, Ros94] as envisioned by Dana Scott in 1977.*

Definition 2.3.2 (F-iteration)**.** *Let $(\mu F, \iota_F)$ be the initial F-algebra, and $\phi : F\ A \to A$ an arbitrary F-algebra. $(\!|\phi|\!) : \mu F \longrightarrow A$ is defined to be the unique homomorphism in the following commuting diagram:*



In the category **Set**, the object $\mathbb{N}$ is the carrier of the initial algebra for $1 + (\_)$. The structure map $\iota^{1+(\_)}$ is $[zero, succ]$, and many familiar functions can be defined using its universal property:

$$\begin{array}{lcl} add\ (n,m) & = & (\!|[\lambda x.m, succ]|\!)(n) \\ mult\ (n,m) & = & (\!|[\lambda x.\, zero, \lambda x.\, add(m,x)]|\!)(n) \\ pred & = & (\!|[id + [zero, succ]]|\!) \end{array}$$

Several basic properties that are frequently used by functional programmers arise from iteration. Our exposition follows Vene [Ven00] and regards these as important for formal methods:

Corollary 2.3.1 (Basic properties of iteration)**.** *Let $(\mu F, \iota_F)$ be the initial F-algebra.*

- ***Cancellation:*** *For any other F-algebra $\phi : F\ A \to A$ we have*

$$(\!|\phi|\!) \circ \iota_F = \phi \circ F(\!|\phi|\!) \qquad ((|\cdot|)\text{-SELF})$$

- ***Reflection:***

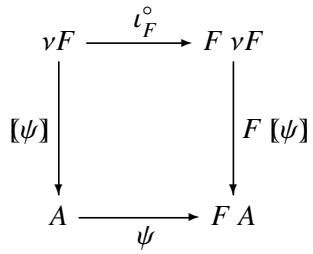
$$id = (\!|\iota_F|\!) \qquad ((|\cdot|)\text{-REFL})$$

- ***Fusion:*** *For any F-algebras $\phi : F\ A \to A$ and $\xi : F\ B \to B$, and arrow $f : A \to B$*

$$f \circ \phi = \xi \circ F\ f \ \Rightarrow\ f \circ (\!|\phi|\!) = (\!|\xi|\!) \qquad ((|\cdot|)\text{-FUSION})$$

It is trivial to dualise F-iteration to the useful notion of F-coiteration simply by reversing the arrows. The "conaturals" arises from such dualisation, namely the ordinal $\omega + 1$ regarded as a cpo. Note that there hence is an extra "infinite" element adjoined.

Definition 2.3.3 (F-coiteration)**.** *Let $(\nu F, \iota_F^\circ)$ be the final F-coalgebra, and $\psi : A \to F\ A$ an arbitrary F-coalgebra. $[\![\psi]\!] : A \longrightarrow \nu F$ is defined to be the unique homomorphism in the following commuting diagram:*



A basic example of a coinductive datatypes, i.e. a final coalgebra, is a stream. A stream of natural numbers is, for example, given by the functor $\mathbb{N} \times (\_)$. Note that the associated initial algebra in **Set** merely contains a list of one element, whereas the final coalgebra for this functor (in **Set**) consists of infinite sequences of natural numbers.

Some standard examples of coiterative functions on streams are the following:

$$\begin{array}{lll} \mathit{nats} & = & [\![\langle \mathit{id}, \mathit{succ} \rangle]\!] \\ \mathit{zip} & = & [\![\langle \pi_1 \times \pi_1, \pi_2 \times \pi_2 \rangle \circ (\iota^\circ \times \iota^\circ)]\!] \\ \mathit{iterate}\quad (f) & = & [\![\langle \mathit{id}, f \rangle]\!] \end{array}$$

The basic properties dualise directly (again, these are often used in functional programming):

Corollary 2.3.2 (Basic properties of coiteration)**.** *Let (A, ι_F°) be the final F-coalgebra.*

- ***Cancellation:*** *For any other F-coalgebra $\psi : B \to F\ B$ we have*

$$\iota_F^\circ \circ [\![\psi]\!] = F[\![\psi]\!] \circ \psi \qquad ([\![\cdot]\!]\text{-SELF})$$

- ***Reflection:***

$$\mathit{id} = [\![\iota_F^\circ]\!] \qquad ([\![\cdot]\!]\text{-REFL})$$

- ***Fusion:*** *For any F-coalgebras $\psi : B \to F\ B$ and $\xi : C \to F\ C$, and arrow $f : B \to C$*

$$\psi \circ f = F\ f \circ \psi \ \Rightarrow\ [\![\psi]\!] \circ f = [\![\xi]\!] \qquad ([\![\cdot]\!]\text{-FUSION})$$

It is sometimes better to use other recursion schemes than iteration and coiteration (with respect to a functor), when defining and reasoning with functional programs. For example, it is well known from recursion theory that the factorial function *fact* is primitive recursive rather than iterative (on the natural numbers):

$$fact \stackrel{def}{=} \pi_1 \circ (\!|[\lambda x.\langle 1, 0\rangle, \lambda\langle f, n\rangle \circ \langle (n+1) * f, n+1\rangle]|\!)$$

Note that *fact* makes use of its recursive argument, which decreases from the provided argument in decrements of 1. Hence, primitive recursion is a generalisation of iteration on the natural numbers, where an extra parameter is maintained during the otherwise iterative computation. Since iteration is defined for arbitrary functors and arbitrary instantiations of the involved carrier of the target algebra, we have the following:

Definition 2.3.4 (F-primitive recursion)**.** *Given $\phi : F(A \times \mu F) \to A$, the paramorphism $(\!|\phi|\!) : \mu F \longrightarrow A$ is defined to be the unique arrow, which makes the following diagram commute:*

$$\begin{array}{ccc} F\mu F & \xrightarrow{\iota_F} & \mu F \\ {\scriptstyle F\,\langle (\!|\phi|\!), id_{\mu F}\rangle}\downarrow & & \downarrow{\scriptstyle (\!|\phi|\!)} \\ F(A \times \mu F) & \xrightarrow[\phi]{} & A \end{array}$$

Although this recursion scheme in a sense is more expressive than F-iteration, it can still be expressed by using the latter up to a post-composed projection:

Lemma 2.3.1 (Meertens [Mee92])**.** $(\!|\phi|\!) = \pi_1 \circ (\!|\langle \phi, \iota \circ F\pi_2\rangle|\!)$

The reader may note that the situation for a general functor is not entirely different to the situation for a natural number object and primitive recursion. (However, work by Uustalu and Vene [UV99] has demonstrated that milder assumptions suffices, namely that cartesian closure is not required.) We collect some useful properties for this principle:

Corollary 2.3.3 (Properties of F-primitive recursion)**.** *Let (A, ι_F) be the final F-algebra.*

- ***Cancellation:*** *For any arrow $\phi : F(A \times \mu F) \to A$ we have*

$$(\!|\phi|\!) \circ \iota = \phi \circ F\langle (\!|\phi|\!), id\rangle \qquad ((\!|\cdot|\!)\text{-Self})$$

- ***Reflection:***

$$id = \langle\!|\iota_F \circ F\pi_1|\!\rangle \qquad (\langle\!|\cdot|\!\rangle\text{-REFL})$$

- ***Fusion:*** *For any arrows* $\phi : F(A \times \mu F) \to A$, $\psi : F(B \times \mu F) \to B$ *and* $f : A \to B$ *we have*

$$f \circ \phi = \psi \circ F(f \times id) \;\Rightarrow\; f \circ \langle\!|\phi|\!\rangle = \langle\!|\psi|\!\rangle \qquad (\langle\!|\cdot|\!\rangle\text{-FUSION})$$

Finally, suppose we want to define a map $\mu F \times \Gamma \to A$ such that the parameter Γ is, intuitively, passed between each recursive call. This is achieved by the following construction of a unique homomorphism h, whose transpose $\bar{h} : \mu F \times \Gamma \to A$ is exactly such a map:

$$\begin{array}{ccc} F(\mu F) & \longrightarrow & \mu F \\ {\scriptstyle F(h)}\downarrow & & \downarrow{\scriptstyle h} \\ F(\Gamma \multimap A) & \xrightarrow[\phi]{} & \Gamma \multimap A \end{array}$$

Several other (co)recursion schemes have been studied in the literature. These range from parametric (co)recursion [Spe93, CF92, CS92, Mos01] to e.g. monadic recursion [Fok94, MJ95, Par01]. Recently, coiteration has been generalised by using distributive laws (based on Lenisa's identification of pointed functors in this context, i.e. a functor for which there exists a natural transformation $\eta : F \Rightarrow 1$) [Len99, Bar03, Bar04] and a related notion of bisimulation up-to-context [San98b]. Paper III will explore a recursion scheme that generalises F-(co)iteration and is based on the more general notion of dialgebra rather than (co)algebra. Since this means tat we move to a domain-theoretic category which is algebraically compact (see below), we must let go of some type structure, and cartesian closure in particular. Moreover, we consider mixed-variant functors (difunctors) rather than endofunctors, in order to cover the recursive types required for interpreting typed object calculus.

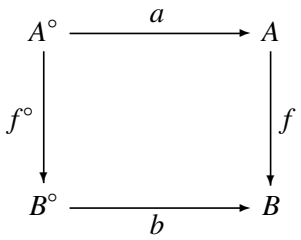
2.3.1 Involutions

The following construction is due to John Power [Fio96b], and formalises the notion of self-dual via an involution. In this thesis, it will be used both implicitly and explicitly:

Definition 2.3.5 (Involutory category)**.** *Let $\mathbb{C}$ be a category. We say that $(\_)^\circ$ is an* involution *on $\mathbb{C}$, when we have*

$$(\_)^\circ \circ (\_)^\circ = Id_{\mathbb{C}}.$$

That is, for any $f \in \mathbb{C}(A, B)$ we have $f^\circ \circ f^\circ = f$, implying that $f^\circ \in \mathbb{C}(B, A)$. Given such an involution, we define the category **Inv**$(\mathbb{C}, (\_)^\circ)$ *to be the category where objects are pairs $(A, a : A^\circ \to A)$ such that $a \circ a^\circ = id_A$. These objects are called* involutory objects*. A morphism $f : (A, a : A^\circ \to A) \to (B, b : B^\circ \to B)$ in* **Inv**$(\mathbb{C}, (\_)^\circ)$ *is a morphism f from $\mathbb{C}$ such that the following diagram commutes in $\mathbb{C}$:*



Note that this says that the involutory category consists of $(\_)^\circ$-algebras, a, which are split epic in $\mathbb{C}$, with the right inverse given as a°.

Before considering an the important example, we note that we have two functors Π_1 and Π_2 given as morphisms in **Cat** (which has products):

$$\begin{array}{lcl} \Pi_1 & : & \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{C}^{op} \\ \Pi_2 & : & \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{C} \end{array}$$

We apply the functor $(\_)^{op}$ on **Cat** to these, giving

$$\begin{array}{lcl} \Pi_1^{op} & : & \mathbb{C} \times \mathbb{C}^{op} \to \mathbb{C} \\ \Pi_2^{op} & : & \mathbb{C} \times \mathbb{C}^{op} \to \mathbb{C}^{op} \end{array}$$

Hence we also have a functor

$$\langle \Pi_2^{op}, \Pi_1^{op} \rangle : \mathbb{C} \times \mathbb{C}^{op} \to \mathbb{C}^{op} \times \mathbb{C}$$

which we will call $(\_)^\S$.

Example 2.3.1. *We have that* **Inv**$(\mathbf{Cat}, (\_)^{op})$ *is an involution. In this category $(\mathbb{A}^{op} \times \mathbb{A}, \langle \Pi_2, \Pi_1 \rangle)$ is an involutory object for any category $\mathbb{A}$. This can be seen by noting that $(\mathbb{A}^{op} \times \mathbb{A})^{op} = \mathbb{A} \times \mathbb{A}^{op}$. Hence, we for example have that the functor*

$\langle \Pi_2, \Pi_1 \rangle : \mathbb{A} \times \mathbb{A}^{op} \to \mathbb{A}^{op} \times \mathbb{A}$ *is an* $(\_)^{op}$*-algebra since* $(\langle \Pi_2, \Pi_1 \rangle)^{op} = \langle \Pi_2, \Pi_1 \rangle$ *is self-dual. The morphisms H are homomorphisms between* $(\_)^{op}$ *algebras:*

$$\begin{array}{ccc} \mathbb{A}^{op} & \xrightarrow{F} & \mathbb{A} \\ {\scriptstyle H^{op}}\downarrow & & \downarrow{\scriptstyle H} \\ \mathbb{B}^{op} & \xrightarrow[G]{} & \mathbb{B} \end{array}$$

Let us denote by $\breve{\mathbb{C}}$ the category $\mathbb{C}^{op} \times \mathbb{C}$ and note that $(\breve{\mathbb{C}})^n = (\breve{\mathbb{C}^n})$. Let us consider some special morphisms arising from the previous example by taking both F and G to be $\langle \Pi_2, \Pi_1 \rangle$ (but for different categories) and $\mathbb{A} = \breve{\mathbb{C}} = \mathbb{C}^{op} \times \mathbb{C}$ and $\mathbb{B} = \breve{\mathbb{D}}$:

$$\begin{array}{ccc} \mathbb{C} \times \mathbb{C}^{op} & \xrightarrow{\langle \Pi_2, \Pi_1 \rangle} & \mathbb{C}^{op}\mathbb{C} \\ {\scriptstyle H^{op}}\downarrow & & \downarrow{\scriptstyle H} \\ \mathbb{D} \times \mathbb{D}^{op} & \xrightarrow[\langle \Pi_2, \Pi_1 \rangle]{} & \mathbb{D}^{op} \times \mathbb{D} \end{array}$$

This diagram requires exactly that $H_1(f, g) = H_2(g, f)$. Motivated by this observation, morphisms in $\mathbf{Inv}(\mathbf{Cat}, (\_)^{op})$, seen to be functors of the form $\breve{\mathbb{C}} \to \breve{\mathbb{C}}$, are hereafter called *symmetric functors*.

Each symmetric functor F induces two functors F_1 and F_2 by post-composition with the projections Π_1 and Π_2 in **Cat**. In fact, each symmetric functor is uniquely determined by F_2, i.e., by the associated difunctor $\breve{\mathbb{C}} \to \mathbb{C}$. This is shown in the following adjunction in **Cat**:

$$\frac{\breve{\mathbb{C}} \xrightarrow{F} \mathbb{C}}{\breve{\mathbb{C}} \xrightarrow{\breve{F}} \breve{\mathbb{C}}}$$

The bijections are explicitly given as follows:

$$\begin{array}{rcl} \breve{F} & \mapsto & \breve{F}_2 \\ F & \mapsto & \langle F^{op} \circ (\_)^{\S}, F \rangle \end{array}$$

That is, a difunctor gives a symmetric functor using a "doubling trick" which is due to Freyd [Fre90], and symmetric functors give difunctors through projections. These translations also apply to a contravariant functor $G : \mathbb{A}^{op} \to \mathbb{A}$ which can be

extended into $G \circ \Pi_1 : \breve{\mathbb{A}} \to \mathbb{A}$ dummy in its positive component. We will write $\breve{G}$ for these $(G \circ \Pi_1)^{\breve{}}$, being slightly flexible with the notation.

2.4 Domain Theory

Domain theory [SHLG94, AJ94, AC98] is a branch of mathematics that aims at explaining computations in terms of processes of approximation of ideal values. This theory can give a fine-grained explanation of programming language concepts and computable functions, since it explicates the intermediate (approximative) stages of computations and views a computation as being the limit point of a generalised sequence of such approximations. It moreover can be used for computing on generalised datatypes such as the real numbers or object types. One goal in domain theory research has been to construct categories of domains that are cartesian closed (thus allowing simply typed lambda calculi, and other type theories, to be interpreted), but also to give solutions to recursive domain equations. The latter is central to the interpretation of many programming languages, including concurrent languages (see e.g. [AC98]), and in particular the untyped lambda calculus [Bar84]. In this thesis, we will use such domain equations also when giving a denotational semantics to typed object calculus. The equations that we will be using generalise those typically used for models of untyped lambda calculus. The reason why such domain equations are natural also for object-based programming languages is that objects are used both as data and as functions (i.e., when a method is invoked, it can be viewed as a function that regards itself as being part of the input). In this respect, models of typed object calculi inherits many characteristics from models of untyped lambda calculus (see e.g. [Wad76]), while types must also be appropriately modelled. This places models of typed object calculus well within the scope of domain theory. For example, method invocation involves a series of approximations which in the limit corresponds to a result (typically another object). From a programming language perspective, we obviously wish that this result arises at a finite stage of the computation. These intuitions are captured by notions such as directed sets (sequences of approximations), compact elements (finite stage) and limit (ideal value). Moreover, we must view methods as continuous functions between such domains, which preserve the sequences and the limit points in a suitable manner. This in particular means that function spaces on domains will themselves be domains.

2.4.1 Predomains

The category **CPO** consists of Scott-continuous functions and directed-complete partial orders (henceforth just called *cpos*). This category is the familiar one [AJ94, SHLG94]:

Definition 2.4.1 (Category of complete partial orders)**.**

- *Objects are quadruples* $A = (A; \sqsubseteq_A, \sqcup_A)$ *where* $(A; \sqsubseteq_A)$ *is a partial order for which every directed subset* $\Delta \subseteq A$ *has* $\sqcup_A \Delta \in A$*. Such partial orders are* directed-complete*. (A* directed set *is a nonempty subset that contains all of its binary suprema, i.e.* $x, y \in \Delta$ *implies* $x \sqcup_A y \in \Delta$*.)*

- *Morphisms* $f : A \to B$ *are monotonic functions such that for any directed set* $\Delta \subseteq D$*,* $f(\Delta)$ *is directed and moreover* $f(\sqcup_A(\Delta)) = \sqcup_B(\Delta)$*, i.e. suprema of directed sets are preserved. The latter property is called (Scott-) continuity.*

From now on, we will drop the subscripts of the supremum operator and ordering when these can be inferred from the context.

Theorem 2.4.1. **CPO** *is bicartesian closed, complete, and cocomplete.*

Proof. We will sketch the proof, which is by now standard (e.g. [AJ94]). Products are cartesian products with the componentwise order, and a function space B^A is the space of continuous functions $A \to B$ under the pointwise order, i.e. $f \sqsubseteq g$ precisely when $\forall a \in A f(a) \sqsubseteq_B g(b)$. Thus it is the codomain of the functions that dictates the order structure. Coproducts are disjoint unions of the underlying sets, equipped with the obvious inherited order. Based on these, we have that $- \times A \dashv -^A$ and so cartesian closure. The limit of a diagram $\Delta : \mathbb{J} \to \mathbf{CPO}$ is given as follows:

$$\varprojlim \Delta = \{(x_J)_{J \in |\mathbb{J}|} \in \prod_{J \in |\mathbb{J}|} \Delta(J) \ : \ \text{for all morphisms } j : A \to B \text{ in } \mathbb{J}\ x_B = \Delta(j)(x_A) \}$$

Note that a limit is an equationally defined subset of a $|\mathbb{J}|$-indexed product. The dual holds for colimits (using coproducts instead). □

The full subcategory of $\mathbf{CPO}_\perp$ of **CPO** consists of all cpos which have a distinguished least element $\perp$ (called *pointed cpos*). This category is merely cartesian closed, since coproducts do not exist (the universal property fails to hold for the coalesced sum, and the disjoint union lacks least element). Despite the rich structure **CPO** is not itself a suitable category for solving recursive domain equations. Neither will $\mathbf{CPO}_\perp$ provide sufficient structure for our purposes, but it is equipped

with an operator fix, which assigns a least fixpoint to any continuous endomap. This category is therefore useful for explaining computation phenomena, as the following theorem shows:

Theorem 2.4.2 (Least fixpoint operator)**.** *Let $f : D \to D$ be a continuous function on a pointed cpo D. Then the following is true:*

1. *f has a least fixpoint given by $\sqcup_{n\in\mathbb{N}} f^n(\bot)$.*
2. *The assignment $\mathrm{fix} : D^D \to D^D$, $f \mapsto \sqcup_{n\in\mathbb{N}} f^n(\bot)$ is continuous.*

For a proof, see [AJ94, SHLG94]. This fixpoint operator enjoys a principle known as uniformity (due to Plotkin):

Lemma 2.4.1 (Uniformity of the fixpoint operator)**.** *Let D and E be pointed cpos. Suppose moreover that the following diagram commutes for continuous functions f, g, h where $h(\bot) = \bot$:*

$$\begin{array}{ccc} D & \xrightarrow{h} & E \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ D & \xrightarrow[h]{} & E \end{array}$$

Then it follows that $\mathrm{fix}(g) = h(\mathrm{fix}(f))$

Proof. The result follows from the following simple calculation:

$$\begin{aligned} h(\mathrm{fix}(f)) &= h(\sqcup_{n\in\mathbb{N}} f^n(\bot)), && \text{by continuity} \\ &= \sqcup_{n\in\mathbb{N}} h \circ f^n(\bot), && \text{by assumption} \\ &= \sqcup_{n\in\mathbb{N}} g^n \circ h(\bot), && \text{by strictness} \\ &= \mathrm{fix}(g) \end{aligned}$$

□

This uniformity principle characterises fix uniquely among all fixpoint operators [GS90], and is moreover used by Pitts [Pit96] in an alternative proof a universal property of recursive types due to Freyd [Fre90] (see below).

By requiring that each continuous map f is strict, i.e. $f(\bot) = \bot$, one arrives at a third subcategory of **CPO**, again with pointed cpos as objects. This category

(denoted $\mathbf{CPO}_{\perp!}$) has (cartesian) products, but not exponentials, since $[X \xrightarrow{\perp!} Y]$ (the set of strict continuous maps) has a universal property expressed using smash products which are not the categorical product of this category. Nevertheless, this category (and its subcategories) has some important structure useful for solving recursive domain equations, and therefore is of central importance to this thesis.

Scott-Ershov Domains

Each of the previous categories has full subcategories, known as (Scott-Ershov) *domains*. We consider the case for **CPO**, since the other subcategories will then be obvious.

Definition 2.4.2 (Compact elements)**.** *The set of* compact elements $D_c \subseteq D$ *of a cpo* D *consists of elements* $c \in D$ *such that for any directed set* $\Delta \subseteq D$*, the following equivalence holds:*

$$c \sqsubseteq_D \sqcup_D \Delta \text{ if and only if } c \in \Delta.$$

That is, the compacts are the elements which cannot be ignored when approximating some other elements. In other words, the compact (or finite) elements are those that carries essential information in the cpo.

Definition 2.4.3 (Scott-Ershov domains)**.** *A (Scott-Ershov)* domain D *is a cpo* $(D; \sqsubseteq_D, \sqcup_D)$ *subject to the following two conditions:*

- Consistent completeness: *every bounded set of elements* $E \subseteq D$ *has a supremum* $\sqcup_D E$ *in* D*.*
- Algebraic: *every element* $d \in D$ *is equal to the supremum of the compact elements below it, i.e.* $d = \sqcup_D approx(d) = \sqcup_D\{c \in D_c \ : \ c \sqsubseteq_D d\}$.

Note that the previous definition also defined the sets $approx(d)$ of compacts below an element, and that algebraicity means that any element d in the domain D is fully determined by those elements.

Lemma 2.4.2. *Let* D *be a domain. Then the following is true:*

(i) For $x, y \in D$*,* $x \sqsubseteq y$ *if and only if* $approx(x) \subseteq approx(y)$*.*

(i) If $\Delta \subseteq D$ *is directed, then* $approx(\sqcup\Delta) = \cup\{approx(x) \ : \ x \in \Delta\}$.

(i) Every monotonic function $f : D_c \to E$ *for a cpo* E *has a unique Scott-continuous extension* $\bar{f} : D \to E$*.*

Proof. E.g. [SHLG94]. □

The subcategories **Dom**, $\mathbf{Dom}_{\perp}$, $\mathbf{Dom}_{\perp!}$ inherit the structure of respectively **CPO**, $\mathbf{CPO}_{\perp}$, and $\mathbf{CPO}_{\perp!}$, but in addition enjoy a natural structure for making approximations. (That is, the domains in $\mathbf{Dom}_{\perp}$ have least elements, while in addition the maps in $\mathbf{Dom}_{\perp!}$ preserve these least elements.)

2.4.2 Recursive Domain Equations

For subcategories of **CPO**, there is a subcategory of embedding-projection pairs denoted $\mathbf{CPO}^{ep}$ (and respectively for subcategories of **CPO**). In this category, limits coincide with colimits, so that for (locally) (co)continuous endofunctor F we simultaneously construct a final F-coalgebra and an initial F-algebra. In this case, we say that the category has *bilimits*, meaning that a particular kind of diagram in $\mathbf{CPO}^{ep}$ has such coincidence for its (co)limit. The idea of using the subcategory of ep-pairs is due to Scott [Sco72]. This technique was first recasted by Wand [Wan79] using order-enriched categories, and then further by Smyth and Plotkin [SP82]. In this section, we will summarise some important results ([AJ94, AC98, Gun92, Plo81] adds many more details).

The restriction to a particular class of functors, was exploited by Smyth and Plotkin (loc. cit.) (and further emphasised by Fiore [Fio96b] in his dissertation), by means of enriched category theory [Kel82]. A main idea behind enriched category theory is that a category can carry additional structure on the homsets, e.g. complete partial orders or metric spaces. This is, as we saw in the previous paragraph, useful extra structure because it allows certain limits to be constructed using functors that preserves the local structure. However, this enriched structure can also be forgotten. This is useful because e.g. completeness of an enriched category $\mathbb{A}$ is a stronger property than completeness of $\mathbb{A}_0$ (generally speaking), and similarly for continuity, adjunctions, etc. In the case of a **CPO**-category, this is witnessed by the fact that not all functions in **Set** are actually Scott-continuous. Similarly, the enriched category may not have all the F-(co)algebras that the underlying one possesses. The former problem is particularly important in this context, since we will often require an initial F-algebra, or similar structures, which are subject to suitable enrichment for their (non-trivial) existence. A branch of domain theory known as *axiomatic domain theory* was initiated in the footsteps of Freyd [Fre90]. Later, Fiore [Fio96b] clarified and expanded much of Freyd's pioneering work by taking it into the framework of enriched category theory, and developing additional results. The two most important notions, both due to Freyd, are *algebraic com-*

pleteness and *algebraic compactness*. Both these notions concerns the existence of (co)algebras for a suitable class of functors, where "suitable" is understood as enrichment. However, we will recast these notions without the framework of enriched category theory and let "suitable" be understood as the locally continuous functors.

Definition 2.4.4 (Algebraic (co)completeness, compactness)**.** *A category $\mathbb{A}$ is said to be* algebraically complete *if every endofunctor F on $\mathbb{A}$ (belonging to a suitable pre-specified class) possesses an initial algebra. We dually say that $\mathbb{A}$ is* algebraically cocomplete *if every such F possesses a final coalgebra. Finally, we say that $\mathbb{A}$ is* algebraically compact *if (i) it is algebraically complete, and (ii) the inverse of the initial algebra of each such functor is the final coalgebra.*

Note that algebraic completeness follows from the property of having ω-colimits (and dually), provided all enriched F are ω-cocontinuous.

The fact that $\mathbf{CPO}_{\perp!}$ satisfies this condition is due to Freyd [Fre90] (but see also pioneering work by [SP82] in the more concrete setting of a subcategory of embedding-projection pairs). We survey Freyd's work here, in particular by recalling that, for a difunctor F, an object X is called F-invariant if there is an isomorphism $\alpha : F(X) \cong X$. If $fix\ (\lambda e.\ \alpha \circ F(e) \circ \alpha^{-1}) \in A \to A$ is the identity, X is called special F-invariant (where *fix* is the usual fixpoint operator in $\mathbf{CPO}_{\perp}$). If it is the only idempotent map $A \to A$ for which $e \circ \alpha = \alpha \circ F(e)$, it is called *minimal invariant* [Fre90]. Freyd [Fre90] showed that in $\mathbf{CPO}_{\perp!}$ there exists an F-invariant object for every locally continuous functor that is minimal in this sense.

An initial dialgebra for a bifunctor G is a dialgebra (A, B, ϕ, ψ) such that for any other G-dialgebra (A', B', ϕ', ψ') there is a unique dialgebra map $(h : A \to A', g : B' \to B)$. The existence of initial dialgebras in $\mathbf{CPO}_{\perp!}$ was established by Freyd:

Theorem 2.4.3 (Existence of Initial Dialgebras [Fre90])**.** $\mathbf{CPO}_{\perp!}$ *has initial dialgebras for every locally continuous bifunctor $G : \mathbf{CPO}_{\perp!}^{op} \times \mathbf{CPO}_{\perp!} \to \mathbf{CPO}_{\perp!}$. In addition, initial dialgebras in $\mathbf{CPO}_{\perp!}$ are of the form $(O_G, O_G, \iota_G, \iota_G^\circ)$ where $\iota_G \circ \iota_G^\circ = id$ and $\iota_G^\circ \circ \iota_G = id$.*

Proof. In the category $\mathbf{CPO}_{\perp}$ (which includes also non-strict Scott-continuous maps and therefore the usual least fixed point operator fix), this follows from existence of minimal invariants together with Plotkin's uniformity of the fixpoint operator (given above). For details, see [Pit96]. □

We close this background chapter with a remark on terminology:

Remark 2.4.1. *There is a certain ambiguity in the term* continuous*. We have said that a functor is ($\mathbb{J}$-) (co)continuous if it preserves (co)limits (of shape $\mathbb{J}$). Moreover, we will say that a functor is (locally) continuous when it preserves the cpo-structure on all homsets (something quite different). Finally, we will say that a function is continuous in the usual domain-theoretic sense. It will be clear which of these concepts we mean from any context. Moreover, we can subsume the notion of local continuity using terminology from enriched category theory.*

Chapter 3

Paper I: Difunctorial Semantics

Difunctorial Semantics of Object Calculus*

Johan Glimming
Department of Numerical Analysis and Computer Science
Stockholm University
Sweden
Email: glimming@kth.se

Neil Ghani
Department of Mathematics and Computer Science
University of Leicester
United Kingdom
Email: ng13@mcs.le.ac.uk

August 30, 2004

Abstract

In this paper, we give a denotational model for Abadi and Cardelli's first order object calculus $\mathbf{FOb}_{1+\times\mu}$ (without subtyping) in the category pCpo. The key novelty of our model is its extensive use of recursively defined types, supporting self-application, to model objects. At a technical level, this entails using some sophisticated techniques such as Freyd's *algebraic compactness* to guarantee the existence of the denotations of the object types.

The last sections of the paper demonstrates that the canonical recursion operator inherent in our semantics is potentially useful in object-oriented programming. This is witnessed by giving a straightforward translation of algebraic datatypes into so called wrapper classes.

* This paper was published in *V. Bono, M. Bugliesi, S. Drossopoulou (editors), Workshop on Object-Oriented Developments (WOOD 2004), Electronic Notes in Theoretical Computer Science, 138(2), Elsevier, 2005; this version includes some minor corrections.*

1 Introduction

The semantics of objects is inherently complicated. Firstly, objects are recursive in the sense that they contain methods which operate on the object itself and may return the object as a result. This reference is known as the *self* parameter to the method. Secondly, object types are usually combined with a notion of subtyping, which can introduce anomalies in the operational semantics. Thirdly, method update (including inheritance) is difficult to model, particularly in combination with subtyping and binary methods. Fourthly, objects often come with some notion of class, which leads to the problem of finding an encoding of classes (pre-methods [1], *new* functions [22], etc have been studied) and an associated mechanism for creating new object instances from classes. Arguably, all these listed problems arise from the recursive nature of objects (see e.g. [3]).

There has been much research on finding good approaches to dealing with the recursion inherent in objects. *Self-application semantics [16, 10]* takes the point of view that objects should be modelled as complicated recursive types. There are other approaches involving *higher-order polymorphism [17, 18]* which hide all recursion under an existential quantifier. *Recursive record semantics [5, 6, 22]* can be seen as a compromise where (covariant) recursive types are used for self-returning methods, and where a fix-point operator at the level of terms is needed to handle references to the object's instance variables. Unfortunately, a direct self-application encoding into $F^{\omega}_{<:}$ fails to support subtyping, while the other two approaches do not support method update (note, however, that [4] gives an encoding with both recursion and bounded existentials that, in fact, support method update). Abadi and Cardelli [1] proposed a variety of different object calculi which support method update and gave them a *primitive semantics* based upon reduction rules.

Our starting point is that method update indeed is an important ingredient in the object-oriented paradigm, and we believe that recursive types should be used in modelling objects. For reasons explained in [1], we therefore consider the notion of object fundamental and, hence, prefer a streamlined theoretical approach rather than an encoding via $F^{\omega}_{<:}$. One aim of our work is to provide a mathematical foundation for logics of object calculi, particularly logics of program transformations. Direct denotational models of object calculi are more suited for this purpose.

This paper presents the first steps in this program. We develop a categorical model for object calculus $\mathbf{FOb}_{1+\times\mu}$ presented in [1] (with minor modifications), based on an interpretation of object types as recursive types via mixed variance functors. The encoding of object types has the flavour of a self-application semantics. The main novelty with this model is in its extensive use of recursively defined

types to model object types. Our mathematical setting is the category pCpo where we model objects as solutions of $\mathit{self} \cong \mathit{self} \to \mathsf{F}\ \mathit{self}$, where F is a covariant functor representing the type of the methods. The simplest objects give rise to a constant functor F while the full generality allows us to define what we term *wrapper classes* for algebraic datatypes. We believe that this model can be extended to support subtyping by natural transformations on the underlying functor F and, by regarding F as a *pattern functor*, opens the way to a polytypic style of object oriented programming.

The paper is structured as follows: after setting up the mathematical framework of Freyd's algebraically compact categories in section 2, we present the calculus $\mathbf{FOb}_{1+\times\mu}$ in section 3. In section 4, we give a semantics for the calculus in pCpo. In section 5, we discuss wrapper classes, i.e., link our semantics of objects to the algebraic and coalgebraic style of programming. Section 6 summarises our contribution and compares to related work.

2 Mathematical Preliminaries

We assume familiarity with elementary category theory and, particularly, the basic concepts of category theory such as product, and exponential — see Mac Lane [13] for details. As mentioned in the introduction, we propose a denotational model of typed object calculi whose key novelty is the use of recursively defined types. A simple example, e.g. finding an object D such that $D \cong [D, D]$, indicates that the existence of such recursively defined types is not at all obvious, e.g. there clearly is no set D such that $D \cong [D, D]$. The key feature of this example is that the mapping of an object D to the object $[D, D]$ is not a functor in that the left occurrence of D in the expression $[D, D]$ occurs *contravariantly* while the right occurrence is *covariant*. Such mappings are called *difunctors*.

Definition 2.1 (Difunctor)**.** *If $\mathcal{C}$ is a category, a difunctor is a functor $\mathsf{F} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$. A fixed point of such a difunctor is an object X such that $X \cong \mathsf{F}\ X\ X$*

There has been much research on finding fixed points for difunctors. The classic paper [21] defines a category of embedding and projection pairs where the functor F acts covariantly and from which a fixed point of F can be derived. More recently, [12, 11, 8, 9] have used the more axiomatic setting of *algebraically compact* categories. i.e., categories where (in a suitably qualified sense) all covariant functors have an initial algebra the inverse of whose structure map is the final coalgebra. The related, but weaker, property of *algebraic completeness* merely requires all (again, in a suitably qualified sense) covariant functors to have an initial algebra.

The axiomatic approach is potentially easier to apply to non-domain theoretic models such as realizability models and models containing intensional features. Since we do not wish to over commit ourselves to a specific semantic setting at this stage, we therefore implicitly follow the axiomatic setting of [8, 9] in working in the Kleisli category of a lifting monad. However, for the purpose of concreteness and simplicity of this presentation, we chosen to work with the canonical model of the category pCpo of *w*-complete partial orders and partial continuous functions. We denote by Cpo the subcategory of pCpo consisting of all cpos and total continuous functions. The salient facts about the categories pCpo and Cpo can be found in [19]. Cpo has the standard structure of being cartesian closed with finite coproducts. We give a brief summary of the structure of pCpo:

- **Zero object:** The empty cpo is a zero object in pCpo. That is, it is both an initial object and a terminal object.
- **Coproducts:** If A and B are cpos, their disjoint union is the coproduct of A and B in pCpo.
- **Partial Products:** If A and B are cpos, the cartesian product of the underlying sets is their partial product. It is not a product as the domain of definition of the pairing (f, g) is the intersection of the domains of f and g and hence $fst(f, g) \neq f$ etc. We denote the partial product by $A \otimes B$ to remind ourselves that it is not a product.
- **Kleisli/Partial Exponentials:** If A and B are cpos, the set of partial continuous functions from A to B forms a cpo, as usual. We denote this cpo $[A, B]$ or $A \rightharpoonup B$. As expected, partial exponentials are right adjoint to the partial product. $- \otimes A \dashv [A, -] : \mathsf{Cpo} \longrightarrow \mathsf{pCpo}$. Note the domains and codomains for the functors involved in this adjunction.
- **Compactness:** pCpo is algebraically compact in that all locally continuous functors have coinciding initial algebras and final coalgebras [11].

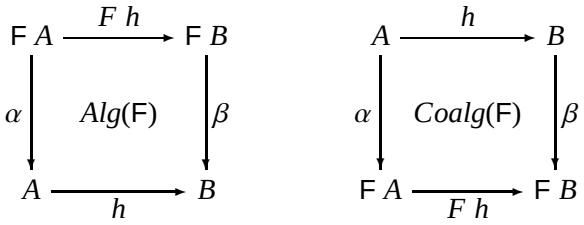
Here, it is worth noting that, apart from compactness, we would have liked our ambient category to be cartesian closed and have finite coproducts so that we could manipulate polynomial functors and their (co-)algebras using the standard techniques. Indeed, settling for partial products and Kleisli exponentials may seem like a poor alternative. However, any compact category has a zero object (induced as the fixed point of the identity functor), and a CCC with a zero object is inconsistent as

$$A \cong A \times 1 \cong A \times 0 \cong 0$$

Hence we cannot get away from working in a non-cartesian closed setting. Nevertheless, the subcategory Cpo (where values take their denotation) is, of course, still cartesian closed.

So, given a category like pCpo, how does one find fixed points for difunctors? Recall that in the simpler case of a covariant endofunctor $\mathsf{F} : \mathcal{C} \longrightarrow \mathcal{C}$, one finds an object $A \cong FA$ as the initial F-algebra or final F-coalgebra.

Definition 2.2 (Algebra, Coalgebra)**.** *Given a functor* F *we say that an arrow* $\alpha : \mathsf{F}\ A \to A$ *is an* F*-algebra with carrier A. Such* F*-algebras are the objects in a category* **Alg**(F) *for every functor* F*. The dual notion is that of* F*-coalgebra, i.e. reversed arrows* $\alpha : A \to \mathsf{F}\ A$*. The arrows between (co)algebras are* F*-homomorphisms, i.e. arrows h such that, for* F*-algebras the left diagram below commutes and, for* F*-coalgebras the right diagram below commutes:*

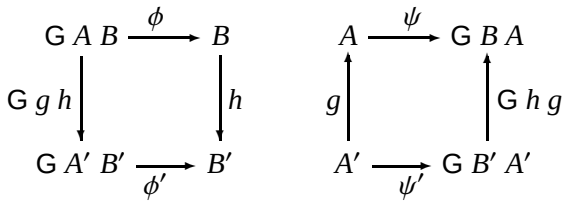


When working with difunctors, algebras and coalgebras generalise to *dialgebras*. Note the presence of both covariance and contravariance in a difunctor means that we have no need for the dual notion of a dialgebra. The term dialgebra has several definitions in the literature. We give ours here:

Definition 2.3 (Dialgebras)**.** *A* G-dialgebra *for difunctor* $\mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ *is a pair of objects A, B together with an associated pair of arrows* $f : \mathsf{G}\ A\ B \to B$ *and* $g : A \to \mathsf{G}\ B\ A$*.*

The category of dialgebras has maps between dialgebras given as follows:

Definition 2.4 (Dialgebra Maps)**.** *Given* G*-dialgebras* (A, B, ϕ, ψ) *and* (A', B', ϕ', ψ')*, a* G-homomorphism *is a pair of arrows* $(g : B \to B', h : A' \to A)$ *such that the following diagrams commute:*



A key idea in axiomatic domain theory is to use algebraic compactness to find fixed points for difunctors. Here is a sketch of the construction:

Lemma 2.5. *Let* $\mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ *be a difunctor on an algebraically compact category* $\mathcal{C}$*. Then* G *has a fixed point.*

Proof. Form the functor $\mathsf{G}' : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}^{op} \times \mathcal{C}$ by following the *doubling trick* proposed by Freyd:

$$\mathsf{G}'\ X\ Y \triangleq (\mathsf{G}(Y, X), \mathsf{G}(X, Y))$$

Since $\mathcal{C}$ is algebraically complete, so is $\mathcal{C}^{op} \times \mathcal{C}$ and thus G' has an initial algebra, say $\mathsf{G}'(X, Y) \to (X, Y)$, which is given by maps $inn_G : X \to \mathsf{G}(Y, X)$ and $out_G : G(X, Y) \to Y$. By Lambek's lemma, inn_G and out_G are isomorphisms. Next, the pair $(out_G, inn_G) : (Y, X) \to \mathsf{G}'(Y, X)$ is easily seen to be the final G'-coalgebra. Since $\mathcal{C}$ is algebraically compact, so is $\mathcal{C}^{op} \times \mathcal{C}$ and hence the initial G'-algebra and final G'-coalgebra coincide. Thus $X = Y$ and we have a G-fixed point as required. □

Of course, while the above proof may seem simple, much of the work is hidden in proving that i) algebraic completeness and compactness are preserved by taking products and opposite categories; ii) formalising exactly the class of difunctors which are to be considered; and iii) proving that certain categories are algebraically complete and compact. Further subtle and technical issues arise, e.g. that these fixed points should be suitably parameterised etc, but for this presentation we have decided to gloss over the details. See [8] for details. Having said this, the modularisation of the construction of fixed points is very elegant. Notice also that more is true than we claimed. In particular we constructed a specific fixed point of a difunctor with a universal property, namely, the initial dialgebra. We shall put this universal property to use later.

3 Object Calculus

We will now give the syntax and operational semantics of a first-order object calculus, henceforth referred to as **FOb**, which is essentially Abadi and Cardelli's $\mathbf{FOb}_{1\mu}$ extended with unit, product and coproducts. **FOb** hence has method updates, but not subtyping or higher-types. There are no real surprises in the calculus and its inclusion is merely for the sake of the completeness of the paper.

We assume countable sets L (method labels), V (type variables), and U (term variables) and will use Greek letters for type variables, lower case letters for term variables, and l_i with index $i \in \mathbb{N}$ for method labels.

Definition 3.1 (**FOb**-types)**.** *The set* $\mathcal{T}_{\mathbf{FOb}}$ *is defined by induction with*

τ	:=	v	*type variables*
		1	*terminal type*
		$\tau_1 \times \tau_2$	*product types*
		$\tau_1 + \tau_2$	*sum types*
		$\tau_1 \to \tau_2$	*function types*
		$\mu v.\tau$	*recursive types*
		$[l_1 : \tau_1, \ldots, l_n : \tau_n]$	*object types*

where $v \in V$*, and for each* i*,* $l_i \in L$*.*

Notice that there is no restriction on the occurrences of type variables in recursive types which means this calculus is expressive in including a variety of sophisticated types such as $\mathsf{Lam} = \mu X.\mathbb{N} + (X \to X)$ where $\mathbb{N} = \mu X.1 + X$. As usual, one next defines the pre-terms of **FOb**.

Definition 3.2 (**FOb**-preterms)**.** *The set* $\mathcal{N}_{\mathbf{FOb}}$ *is defined by induction with* $U \subseteq \mathcal{N}_{\mathbf{FOb}}$*. Inductively, if* $m, m', b \in \mathcal{N}_{\mathbf{FOb}}$*,* $\tau_i \in \mathcal{T}_{\mathbf{FOb}}$*,* $l_i \in L$ *and* $x_i \in U$ *(with* $i \in \mathbb{N}$*) then* $*$*,* $[l_1 = \varsigma(x_1 : \tau_1)b_1, ..., l_n = \varsigma(x_n : \tau_n)b_n]$*,* $m.l$*,* $m\ m'$*,* $\lambda(x : \tau)b$*,* $m.l \Leftarrow \varsigma(x : \tau)m'$*,* $inl\ m, inr\ m$*,* $case(b, x.m, y.m')$*,* $fst\ m$*,* $snd\ m$*,* (m, m')*,* $inn(\tau, m)$*, and* $out(m)$ *are in* $\mathcal{N}_{\mathbf{FOb}}$*.*

We will adopt the following convention for meta-variables denoting terms: the symbol o is a term of the form $[l_1 = \varsigma(x_1 : \tau_1)b_1, ..., l_n = \varsigma(x_n : \tau_n)b_n]$ for some $n \in \mathbb{N}$. Sometimes we also write $[l_i = \varsigma(x_i : \tau_i)b_i]_{i \in I}$ for such terms, for I a finite subset of $\mathbb{N}$. Terms of these two equivalent forms are known as *objects*.

For convenience we will identify any two terms (types) which are equal up to the order of method labels, e.g. $[l_1 = ..., l_2 = ...] \equiv [l_2 = ..., l_1 = ...]$, and we assume that method labels occurring inside the same enclosing brackets [...] are distinct. We use Abadi and Cardelli's definition of substitution, and write $m[a/b]$ meaning a is substituted for all free occurrences of b in m [1]. Further, $m(x)$ means x is free in m (hence possibly not even occurring in m), and then $m(a) \equiv m[a/x]$.

Definition 3.3 (**FOb**-Environments)**.** *An environment* E *is a finite sequence of the form* $x_1 : \tau_1, \ldots, x_n : \tau_n$ *with no variable occurring twice in the sequence.*

The typing judgments are of the form $E \vdash a : \sigma$ where E is an environment, a is a pre-term and σ is a type. We also let $E \vdash \sigma$ abbreviate $\exists a \in \mathcal{N}_{\mathbf{FOb}}$ such that $E \vdash a : \sigma$, i.e. the statement that σ is a well-formed inhabited type.

Definition 3.4 (**FOb**-Typing judgments)**.** *The typing judgments of* **FOb** *are*

$$\frac{E, x_i : \sigma \vdash b_1 : \tau_1, ..., b_n : \tau_n \qquad \sigma = [l_1 : \tau_1, \ldots, l_n : \tau_n]}{E \vdash [l_1 = \varsigma(x_1 : \sigma)b_1, ..., l_n = \varsigma(x_n : \sigma)b_n] : \sigma} \quad \textit{(object intro)}$$

$$\frac{E \vdash a : \sigma, \qquad E, x : \sigma \vdash b : \tau_j \qquad \sigma = [l_1 : \tau_1, ..., l_n : \tau_n]}{E \vdash a.l_j \Leftarrow \varsigma(x : \sigma)b : \sigma} \quad \textit{(object update)}$$

$$\frac{E \vdash a : [l_1 : \tau_1, ..., l_n : \tau_n] \qquad 0 < i < n+1}{E \vdash a.l_i : \tau_i} \quad \textit{(object elim)}$$

$$\frac{E \vdash c : \sigma_1 + \sigma_2 \qquad E, x_i : \sigma_i \vdash m_i : \tau \qquad (i = 1, 2)}{E \vdash case(c, x_1.m_1, x_2.m_2) : \tau} \quad \textit{(case)}$$

$$\frac{E \vdash a : \tau}{E \vdash inl\ a : \tau + \sigma} \quad \textit{(inl)} \qquad \frac{E \vdash m : \tau_1 \to \tau_2, n : \tau_1}{E \vdash m\ n : \tau_2} \quad \textit{(λ-elim)}$$

$$\frac{E \vdash b : \sigma}{E \vdash inr\ b : \tau + \sigma} \quad \textit{(inr)} \qquad \frac{E, x : \tau_1 \vdash b : \tau_2}{E \vdash \lambda(x : \tau_1)b : \tau_1 \to \tau_2} \quad \textit{(λ-intro)}$$

$$\frac{E \vdash a : \tau \times \sigma}{E \vdash fst\ a : \tau} \quad \textit{(fst proj)} \qquad \frac{}{E \vdash * : \mathbb{1}} \quad \textit{(unit-unit)}$$

$$\frac{E \vdash a : \tau \times \sigma}{E \vdash snd\ a : \sigma} \quad \textit{(snd proj)} \qquad \frac{E \vdash b : \sigma[\mu\ \alpha.\sigma(\alpha)]}{E \vdash inn(\tau, b) : \mu\alpha.\sigma(\alpha)} \quad \textit{(μ-in)}$$

$$\frac{E \vdash a : \tau \qquad E \vdash b : \sigma}{E \vdash (a, b) : \tau \times \sigma} \quad \textit{(pair form)} \qquad \frac{E \vdash b : \mu\ \alpha.\sigma(\alpha)}{E \vdash out(b) : \sigma[\mu\ \alpha.\sigma(\alpha)]} \quad \textit{(μ-out)}$$

We say a pre-term $m \in \mathcal{N}_{\mathbf{FOb}}$ is well-typed iff there exists a type $\tau \in \mathcal{T}_{\mathbf{FOb}}$ and an environment E such that $E \vdash m : \tau$. We let $\mathcal{M}_{\mathbf{FOb}}$ denote the set of well-typed terms up to the obvious notion of α-equivalence induced by the term-binders λ, ς and case.

4 Difunctorial Semantics

In this section we give a denotational model of **FOb** using the category pCpo. The key feature of this semantics is that it reflects our intuition that the object types of

FOb are fixed points of recursive type equations. More specifically, the recursion is over the self-parameter which occurs negatively. This intuition is clearly seen in the object-intro typing rule for $\sigma = [l_1 : \tau_1, \ldots, l_n : \tau_n]$ which suggests the i'th method will consume the self-parameter, which has type σ, to produce something of type τ_i. Thus, intuitively, the interpretation of σ should satisfy

$$[\![\sigma]\!] \cong [\![\sigma]\!] \rightharpoonup [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

and hence the denotation of σ should be the fixed point of $\mu X.X \rightharpoonup [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. Crucially, the following lemma shows that such an interpretation supports *self-application* [10] which our semantics both requires and supports. We state the lemma specifically for pCpo to make clear we are not using cartesian closure in the proof.

Lemma 4.1. *Let* $\mathsf{F} : \mathsf{pCpo} \longrightarrow \mathsf{pCpo}$ *be a covariant functor and O satisfy* $O \cong [O, FO]$. *Then there is a self-application map* $sapp : O \rightarrow FO$.

Proof. All isomorphisms are total and hence the isomorphism uncurries to give a map $O \otimes O \rightarrow FO$. Now precompose with the diagonal which partial products posses. □

Notice how this differs with the recursive record semantics [1], where the recursion is in the output or covariant position while the contravariant occurrence of $self$ is replaced by having a separate state type, and a fixed point operator at the level of terms. Our semantics also differs from other encodings such as various encoding with existentials [18, 4] where the contravariant occurrence is present but hidden under the existential quantifier. In our model of **FOb** we instead explicate the contravariant $self$ parameter and interpret all object types into more elaborate recursive types which, as we have seen, support self-application.

If $\mathcal{C}$ is a category we denote by $\hat{\mathcal{C}}$ the category $\mathcal{C}^{op} \times \mathcal{C}$ and note that $(\hat{\mathcal{C}})^n = (\hat{\mathcal{C}^n})$. The doubling trick used to obtain fixed points of difunctors assigns to each difunctor $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{D}$ a functor $\hat{F} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{D}^{op} \times \mathcal{D}$. We call functors that arise in this way *symmetric* - see [8] for a full definition. Each symmetric functor F induces two functors F_1 and F_2 by post-composition with the projections Π_1 and Π_2 arising from the product on Cat. In fact the mapping $F \mapsto \hat{F}$ is a bijection between difunctors and symmetric functors with inverse sending F to F_2. This fact will be used below to define symmetric functors by giving difunctors. Finally let $\mathcal{P}$ be the category $\mathsf{pCpo}^{op} \times \mathsf{pCpo}$.

With this notation we can give a semantics to types as follows. If a type τ has n-free type variables[1], its interpretation is a symmetric functor

$$[\![\tau]\!] : \mathcal{P}^n \longrightarrow \mathcal{P}$$

Using the bijection mentioned above, we define the symmetric functor $[\![\tau]\!]$ by giving $[\![\tau]\!]_2$. The exceptions to this rule are for the interpretations of recursive types and object types.

$$\begin{array}{lcl} [\![1]\!]_2 X & = & 1 \\ [\![\tau_1 + \tau_2]\!]_2 X & = & [\![\tau_1]\!]_2 X + [\![\tau_2]\!]_2 X \\ [\![\tau_1 \times \tau_2]\!]_2 X & = & [\![\tau_1]\!]_2 X \otimes [\![\tau_2]\!]_2 X \\ [\![\tau_1 \to \tau_2]\!]_2 X & = & [\![\tau_1]\!]_1 X \multimap [\![\tau_2]\!]_2 X \\ [\![\mu v.\tau]\!] X & = & ([\![\tau]\!] X)^\dagger \end{array}$$

where $([\![\tau]\!]X)^\dagger$ is the fixed point of $[\![\tau]\!]X : \mathcal{P} \longrightarrow \mathcal{P}$. Finally, for an object type $\sigma = [l_1 : \tau_1, \ldots, l_m : \tau_m]$, we have

$$[\![[l_1 : \tau_1, \ldots, l_m : \tau_m]]\!] = [\![\mu v.\ v \to \tau_1 \times \cdots \times \tau_m]\!]$$

Unwinding the definition, we thus have

$$[\![\sigma]\!]_2 X \cong [\![\sigma]\!]_2 X \multimap [\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$$

and note that, in this situation, lemma 4.1 applies since we can take F to be the constant functor returning $[\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$. Just as we gave an interpretation to types, so we give one to environments. If E is an environment with n-free type variables, then

$$[\![E]\!] : \mathcal{P}^n \longrightarrow \mathcal{P}$$

is the symmetric functor defined by

$$[\![x_1 : \tau_1, \ldots, x_m : \tau_m]\!]_2 X = [\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$$

Finally we come to the interpretation for term judgments. If $E \vdash e{:}\tau$ is a judgment using n-type variables, then its interpretation is an indexed family of morphisms

$$[\![E \vdash e{:}\tau]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\tau]\!]_2 A$$

[1] At this point we play a slight price of informality for not indexing judgments by free type variables. However we previously gained by having less notationally cumbersome judgments. We leave the reader to decide if this was an appropriate choice.

for each symmetric functor $A : \mathcal{P}^n$, i.e. for some $X : \mathsf{pCpo}^n$ the functor A is of the form $A = ((X_1, X_1), \ldots, (X_n, X_n))$. Since the semantic clauses for the term constructs associated with the basic types $1, +, \times, \rightarrow$ are as expected, we leave them as an exercise and focus instead on the judgments for object introduction, update and elimination which we take verbatim from Definition 3.4

- **Object Introduction:** By assumption we are given maps

$$[\![E, x : \sigma \vdash b_i{:}\tau_i]\!]_A : [\![E, x : \sigma]\!]_2 A \longrightarrow [\![\tau_i]\!]_2 A$$

  in pCpo. Using the definition of $[\![E, x : \sigma]\!]_2$ and the the adjunction between partial product and and partial exponentials, these correspond to the following maps *in the category* Cpo:

$$[\![E]\!]_2 A \longrightarrow ([\![\sigma]\!]_2 A \rightharpoonup [\![\tau_i]\!]_2 A)$$

  and hence we get, for each A, one map

$$[\![E]\!]_2 A \longrightarrow ([\![\sigma]\!]_2 A \rightharpoonup [\![\tau_1 \times \cdots \times \tau_n]\!]_2 A)$$

  But, since $[\![\sigma]\!]_2 A \rightharpoonup [\![\tau_1 \times \cdots \times \tau_n]\!]_2 A$ is isomorphic to $[\![\sigma]\!]_2 A$, we are done.

- **Object Elimination:** We are given a family of maps

$$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\sigma]\!]_2 A$$

  and want a map

$$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\tau_j]\!]_2 A$$

  This can be constructed by postcomposing with the self-application map $[\![\sigma]\!]_2 A \longrightarrow [\![\tau_1]\!]_2 A \times \cdots [\![\tau_n]\!]_2 A$ and then the j'th projection.

- **Object Update:** Start with the map

$$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\sigma]\!]_2 A$$

  Unwind the isomorphism defining $[\![\sigma]\!]_2 A$. Replace the j'th component of the tuple with

$$[\![E, x : \sigma \vdash b{:}\tau_j]\!]_A : [\![E, x : \sigma]\!]_2 A \longrightarrow [\![\tau_j]\!]_2 A$$

  and then refold the isomorphism to get the required map.

5 Wrapper Classes

We saw in the previous section how object types and the associated term judgments can be given a semantics by solving recursive equations of the form $O \cong O \to K$ for some constant K representing the types of the fields of the object type. There is thus an asymmetry in that the self parameter can be consumed by the methods but the methods can't produce new self's or objects. More generally one would like methods to be able to both consume and return the self parameter - this would make sense in both functional and imperative object calculi. Doing this means solving equations of the form

$$O \cong [O, \mathsf{F}\ O]$$

where F is some covariant functor. Such generalised objects are clearly supported by the semantics we have already developed. Also note by instantiating F with the identity functor we get the classic equation $D \cong [D, D]$.

We put this idea to use by asking the following question. Given that both the initial algebra and final coalgebra styles of programming have proven to be very popular in the functional world, can we incorporate them into the world of objects? More precisely, if F is a covariant functor with initial algebra $\mu\mathsf{F}$ and final coalgebra $\nu\mathsf{F}$, can we find an object O which supports the kind of programming enjoyed by $\mu\mathsf{F}$ and $\nu\mathsf{F}$. Of course, since we work in an algebraically compact category $\mu\mathsf{F} = \nu\mathsf{F}$.

We provide a partial positive answer to this question by choosing O to be the fixed point of the equation $O \cong [O, \mathsf{F}\ O]$. Note that our analysis is semantic in that we treat all covariant functors rather than retreating into some restricted syntactic class of functors such as polynomials. For the rest of this section, fix a covariant functor F and define the difunctor $G(X, Y) = X \to \mathsf{F}\ Y$. Also we write *inn* and *out* for the structure maps

$$inn : [O, FO] \longrightarrow O \qquad out : O \longrightarrow [O, FO]$$

of the initial G-dialgebra. Our first result is that objects can be "evaluated" into the final coalgebra and hence enjoy a notion of equality induced by bisimulation.

Lemma 5.1. *O is an F-coalgebra and hence there is a F-coalgebra homomorphism $O \longrightarrow \nu\mathsf{F}$.*

Proof. From lemma 4.1, self application gives a coalgebra $O \longrightarrow FO$. □

Not only is there a map from O to the final F-coalgebra, but also there is a map from the initial algebra to O

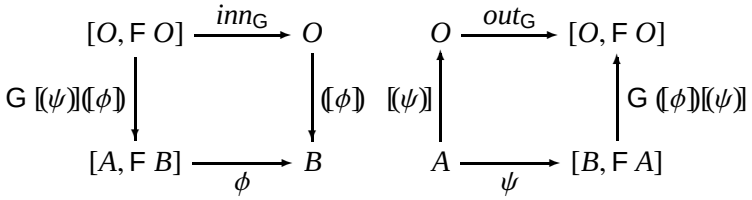
Lemma 5.2. *O is an F-algebra and hence there is a F-algebra homomorphism $\mu\mathsf{F} \longrightarrow O$.*

Proof. We would like constructors for O, that is for O to be an F-algebra. Using the isomorphism defining O, the structure map $\mathsf{F}\ O \longrightarrow O$ can be given by a map $\mathsf{F}\ O \longrightarrow [O, \mathsf{F}\ O]$ which we take to be the first projection after uncurrying. Now that O is an F-algebra, the *fold* operation of the initial algebra defines an F-algebra homomorphism $\mu\mathsf{F} \longrightarrow O$. □

That the composite $\mu\mathsf{F} \longrightarrow O \longrightarrow \nu\mathsf{F}$ is the canonical map induced by the initiality of $\mu\mathsf{F}$ and/or the finality of $\nu\mathsf{F}$ relies on the regularity of O. In this setting O is therefore a retract of $\mu\mathsf{F}$ showing it contains the elements of $\mu\mathsf{F}$ but a whole lot more as well.

Next, we wish to consider recursion principles. Initial algebras come with a canonical recursion operator *fold* which arises as the unique map from the initial algebra to some other algebra. Similarly there is a recursion operator *unfold* which arises as the unique map from some coalgebra to the final coalgebra. As we mentioned earlier, O has the universal property of being the initial dialgebra and hence comes with its own recursion principle for defining maps from O to any other dialgebra. Unwinding the definition of dialgebra etc, this gives the principle of *direcursion*.

Definition 5.3 (Direcursion)**.** *Let (ϕ, ψ) be a dialgebra with types given in the diagram below. Define $(\!|\phi|\!) : O \longrightarrow B$ and $[\!(\psi)\!] : A \longrightarrow O$ to be the unique dialgebra homomorphism such that the following diagram commutes:*



By simply chasing the above diagram, one can extract the direcursion principle as two mutually recursive combinators:

Definition 5.4 (Direcursion - combinators)**.**

$$(\!|\phi, \psi|\!)\ o \triangleq \phi\ \ ((\mathsf{F}\ (\!|\phi, \psi|\!)) \circ (out_{\mathsf{G}}\ o) \circ [\!(\phi, \psi)\!])$$

$$[\!(\phi, \psi)\!]\ o \triangleq inn_{\mathsf{G}}\ \ ((\mathsf{F}\ [\!(\phi, \psi)\!]) \circ (\psi\ o) \circ (\!|\phi, \psi|\!))$$

This recursion scheme has been developed as a programming tool by by [7, 15] and also opens the way for potential optimisations of based upon fusion, deforestation etc and gives laws for object-oriented programs a la *Algebra of Programming*-school. In future work we plan to test whether thesis is practically viable.

Here, we use direcursion to show that O can be used to simulate the unfold operation of the final F-coalgebra. That is given any F-coalgebra $\alpha : A \to \mathsf{F}\ A$, we define a map from A to O. This can be done by instantiating the direcursion principle by taking B to be the one element cpo. The map ϕ must then be the unique total map, while the map $A \to [1, \mathsf{F}\ A]$ sends a to the total function returning $\alpha(a)$.

To summarise, we have defined a translation of some of the key features of initial algebra and final coalgebra programming into the world of objects. That is, we have defined an object type which contains the elements of the initial algebra, has constructors for pattern matching, can be evaluated into the final coalgebra, supports a notion of bisimulation and supports an unfold operator. That these constructions are quite simple suggests to us that these wrapper objects are natural and gives us hope that further concepts can be incorporated into the model without it becoming intractable. But that is of course the subject for future research.

6 Conclusion and Further Work

Our approach in this paper differs from the original denotational semantics given in [1]. Firstly and most fundamentally, they use the ideals/metric approach [14] while our approach is based on Fiore's category of partial maps instantiated for pCpo, thus mimicking the more abstract order-enriched setting of [21, 2]. Secondly, Abadi and Cardelli interpret types as partial equivalence relations (pers) over a universal domain, while we interpret object types by solving recursive type equations in pCpo. As a result, we get a more intuitive model of objects, with an associated principle of recursion. We think the translation of inductive types into wrappers shows the simplicity and naturalness of this model. However, subtyping has known problems in combination with recursive types, and further research is needed in order to model subtyping together with the direcursion principle.

Reus and Streicher [20] have recently treated untyped object calculus in a domain theoretic setting. They use an induction principle to reason with such objects. However, in their work there is one single induction principle, whereas in our typed setting, there is an instantiation for every object type.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi and Gordon D. Plotkin. A per model of polymorphism and recursive types. In J. Mitchell, editor, *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 355–365, Philadelphia, 1990. IEEE Computer Society Press.

[3] Kim B. Bruce. Paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[4] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[5] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[6] Luca Cardelli. Extensible records in a pure calculus of subtyping. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 373–425. The MIT Press, Cambridge, MA, 1994.

[7] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[8] M. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[9] Marcelo P. Fiore and Gordon D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In *CSL*, pages 129–149, 1996.

[10] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[11] P. Freyd. Algebraically complete categories. In *Proc. 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, 1990.

[12] Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90*, pages 498–507. IEEE Computer Society Press, June 1990.

[13] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[14] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 165–174. ACM Press, 1984.

[15] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proc. 7th Conf. of Functional Programing Languages and Computer Architecture*, 1995.

[16] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–124. ACM Press, 1990.

[17] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 299–312. ACM Press, 1993.

[18] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.

[19] Gordon Plotkin. Pisa notes (on domain theory). Available from http://homepages.inf.ed.ac.uk/gdp/publications.

[20] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. In Gordon Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symp. on Logic in Computer Science, LICS 2002*. IEEE Computer Society Press, July 2002.

[21] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[22] Mitchell Wand. Type inference for objects with instance variables and inheritance. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, London, 1994.

Chapter 4

Paper II: Computational Soundness and Adequacy

Computational Soundness and Adequacy for Typed Object Calculus*

Johan Glimming
Department of Numerical Analysis and Computer Science
Stockholm University
Sweden
Email: glimming@kth.se

May 31, 2005 (revised October 15, 2007)

Abstract

By giving a translation from typed object calculus into Plotkin's FPC, we demonstrate that every computationally sound and adequate model of FPC (with lazy operational semantics), is also a sound and adequate model of typed object calculus. This establishes that denotational equality is contained in operational equivalence, i.e. that for any such model of typed object calculus, if two terms have equal denotations, then no program (or rather program context) can distinguish between those two terms. Hence we show that FPC models can be used in the study of program transformations (program algebra) for typed object calculus. Our treatment is based on self-application interpretation and subtyping is not considered. The typed object calculus under consideration is a variation of Abadi and Cardelli's first-order calculus with sum and product types, recursive types, and the usual method update and method invocation in a form where the object types have assimilated the recursive types. As an additional result, we prove subject reduction for this calculus.

*This paper was published as technical report TRITA-CSC-TCS 2007:2 at the School of Computer Science and Communication, Royal Institute of Technology.

1 Introduction

In this paper we will define System $\mathbf{S}^-$, a typed object calculus without subtyping, and interpret this calculus into the metalanguage FPC [19, 20] using a self-application encoding, and prove computational soundness and adequacy of $\mathbf{S}^-$ with respect to this interpretation. As a corollary, every model of lazy FPC that exhibits these two properties, is automatically such a model of typed object calculus. As a consequence, our results make it feasible to use models of FPC as a starting point for the study of semantics of typed object calculus. Given previous research on FPC models, notably by Plotkin and Fiore [19, 6, 8], this formal connection seems to be quite useful (and gives access to a class of computationally adequate models). In particular, program transformations and reasoning principles from FPC models carry over to typed object calculus (notably Freyd's mixed-variant recursion scheme [10]). As an additional result, we establish the subject reduction property of $\mathbf{S}^-$, which unlike any calculus studied by Abadi and Cardelli [1] has sum types (and thus can define many standard datatypes more easily).

The importance of computational adequacy is well-known since Plotkin's pioneering work for PCF [21]. Consider a model $\mathcal{M}$ of typed object calculus. Computational adequacy of $\mathcal{M}$ is the property that if a term has a converging (total, not-bottom) denotation, then that term reduces to a value using the rules of the operational semantics (given by a partial function $\rightsquigarrow$). This property is the reverse implication of the following equivalence, the forward direction being known as computational soundness:

$$\exists v\ t \rightsquigarrow v \quad \Longleftrightarrow \quad [\![t]\!] \text{ total} \tag{1}$$

Let us say that $t \simeq t'$ whenever for each program context C of ground type we have either that (i) $C[t] \rightsquigarrow v$ and $C[t'] \rightsquigarrow v'$ such that v and v' are equal, or (ii) both $C[t] \Uparrow$ and $C[t'] \Uparrow$ (where $\Uparrow$ means divergence, i.e. $\nexists v\ C[t] \rightsquigarrow v$ and similarly for t') and $C[t'] \Uparrow$. This is operational equivalence relation that Plotkin [21] studied for PCF and, after him, many others for other programming languages. The bi-implication (1) implies the following useful property (assuming that we have soundness, i.e. that $t \rightsquigarrow v$ implies $[\![t]\!] = [\![v]\!]$):

$$[\![t]\!] = [\![t']\!] \quad \Longrightarrow \quad t \simeq t' \tag{2}$$

For a proof, suppose $[\![t]\!] = [\![t']\!]$. It follows by soundness that $[\![C[t]]\!] = [\![C[t']]\!]$ holds for any program context C. Since contexts have ground type, computational adequacy ensures that either $C[t] = C[t']$ or else $C[t]$ and $C[t']$ both diverge. Hence

we have $t \simeq t'$. This demonstrates that computational adequacy is a very desirable relationship between operational semantics and denotational semantics, and in particular makes possible to study program transformations (and hence a program algebra) using the model theory of the programming language. This is indeed also the motivation for the present paper.

The idea of using self-application semantics for modelling (or here, interpreting) typed object calculus is not new. It is mentioned by Abadi and Cardelli [1], and appeared already with the work of Kamin [16]. However, recently this approach has been used by Reus, Streicher, and Schwinghammer [23, 22], who have studied it in the context of program logics. We expect our results to be useful for their line of work, although some care must be taken in analysing our computational adequacy result since we are not directly considering any particular model such as theirs.

To the best of our knowledge this is the first full account of these results, and in particular the first proof of computational adequacy of a self-application interpretation with respect to FPC or any of its models. While Viswanathan [25] considers full abstraction for a related typed object calculus, this is not for a self-application interpretation but for a less natural (but interesting, at least from a theoretical viewpoint) interpretation based on a fixed-point operator at the level of terms. Under such an interpretation, it becomes much more complicated to reason with object types, since the universal property associated to recursive types cannot be used directly. Moreover, we demonstrate in this paper that actually a lazy operational semantics must be considered for FPC for the results to be attainable. Hence, our detailed account turned out to provide new insights on the relationship between FPC and typed object calculus.

2 Typed Object Calculus with Recursive Objects

Abadi and Cardelli have developed a family of object calculi, some of which are more powerful than others, e.g. by having subtyping, recursive types, variance annotations, polymorphism, or *Self*-type in addition to the standard first-order fragment [1]. Table 2.1 gives an overview of some typed object calculi, including $\mathbf{S}$ from Abadi and Cardelli's textbook [1] which is particularly expressive, and the simplified calculus $\mathbf{S}^-$ considered in this paper.

The table shows, for example, that variance annotations are not considered at all in this chapter (this is however no fundamental limitation since such annotations can easily be adjoined to an extended system). $\mathbf{S}^-$ will be defined in detail in the

Definition 2.1 (Object Calculi)

	$\mathbf{FOb_1}$	$\mathbf{FOb}_{1\langle:}$	$\mathbf{FOb}_{1\langle:\mu}$	$\mathbf{S}^-$	$\mathbf{S}$
Subtyping		•	•		•
Recursive types			•	•	•
Obj-binder				•	•
Self-type					•
Variance ann.					•
Products				•	
Coproducts				•	
Functions	•	•	•	•	

Note that $\mathbf{S}^-$ is not a subset of $\mathbf{S}$, but contains some extensions such as sum types and functions. These extensions can however also be given to $\mathbf{S}$ (but Abadi and Cardelli's original presentation of $\mathbf{S}$ did not include them).

following sections. It is based on the syntax of $\mathbf{S}$ of [1], but omits the primitive covariant self type and adds some other types instead. A notable omission is subtyping, for which it is currently not known if FPC interpretations (as studied in this paper) can be used. $\mathbf{S}^-$ is essentially a superset of $\mathbf{FOb}_{1\mu}$ of Abadi and Cardelli [1], but with *Obj*-binder instead of the μ-binder, and with the extensions listed in the diagram. Note that we have combined recursive types and object types (using the *Obj* binder) in the sense of $\mathbf{S}$, also without the primitive covariant self type. Since $\mathbf{S}^-$ is endowed with products and coproducts, FPC will contain a subset of the rules of $\mathbf{S}^-$.

We will now define $\mathbf{S}^-$ and give some simple examples. We choose n-ary products and coproducts to simplify these examples. We give an operational semantics with a clear notion of values. Our choice of an operational approach permits us to prove computational soundness and adequacy with respect to a denotational model. These results could not be proven were we to have used the reduction based approach as certain reductions are in fact unsound. The reason for this is that a reduction-based semantics admits a degree of non-determinism in evaluations that invalidates the soundness proof. Notably, an object with some terminating methods and some non-terminating methods, is interpreted as a product of functions, such that even the terminating method may become non-terminating under some

reduction strategies in FPC.

We assume a countable set of method labels L, type variables V, and term variables U. The types of $\mathbf{S}^-$ are given in definition 2.2. Notationally, we write $[\ell_i : \tau_i]^{i\in I}$ for $[\ell_1 : \tau_1, ..., \ell_n : \tau_n]$ with $n \in \mathbb{N}$ and equate object types which are equivalent under permutation of the order of labels or under the obvious notion of α-equivalence induced by the type binder *Obj*. We introduce shorthand $\tau_1 \times \tau_2 = \prod_{i\in\{1,2\}} \tau_i$ and similarly $\tau_1 + \tau_2 = \coprod_{i\in\{1,2\}} \tau_i$ for binary products and coproducts.

Definition 2.2 also gives the preterms of $\mathbf{S}^-$. We identify preterms which are equal up to the order of method labels or are equivalent under the obvious notion of α-equivalence induced by the term-binders λ, ς, and *case* and the type binder *Obj*. We use the standard definition of substitution which can be found in Abadi and Cardelli, and write $m\{a/x\}$ to mean that a is substituted for all free occurrences of x in m [1]. Further, $m(x)$ means x may occur free in m. We use similar notation for the substitution of types for type variables in both types and terms. When clear from the context, we eliminate the type or term variable being substituted for and simply write $m\{a\}$ and $m\{\tau\}$.

Type Theory

A type judgement consists of a sequence of distinct type variables (a type context) together with a type whose free type variables appear in the sequence. The formal definition of type contexts appear in definition 2.3.

Here are a couple of examples:

Example 2.1. *One may consider representing the Java-like interface*

$$\textbf{interface}\ \ Point\ \{\textbf{public void}\ bump(); \textbf{public int}\ val(); \}$$

as the following type in $\mathbf{S}^-$ *(assuming a type Int exists):*

$$Point = Obj(X)[val : Int, bump : X]$$

Example 2.2. *The Java-like interface*

$$\textbf{interface}\ UnLam\ \{\textbf{public void}\ bump(); \}$$

gives rise to an object type of the form

$$UnLam = Obj(X)[bump : X]$$

Definition 2.2 (Syntax of $\mathbf{S}^{-}$)

Syntax for Types
The set $\widehat{\mathcal{T}}$ of pretypes is defined by induction with

τ	::=	X	type
		1	terminal type
		$\prod_{i\in I}\tau_i$	product types
		$\coprod_{i\in I}\tau_i$	coproduct types
		$\tau_1 \to \tau_2$	function types
		$Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$	object types (ℓ_i distinct)

where $X \in V$, and for each i in a finite set I, $\ell_i \in L$ are pairwise distinct.

Syntax for Terms
The set $\widehat{\mathcal{L}}$ of preterms is defined by induction with

m	::=	$\star$	unit
		x_i	term variables
		$\langle m_0, \ldots, m_n\rangle$	tupling
		$\pi_i\, m$	projections
		$case(m_0, x_1.m_1, \ldots, x_n.m_n)$	case
		$\iota_i\, m$	injections
		$m_0\ (m_1)$	λ-application
		$\lambda x : \tau.m$	λ-abstraction
		$Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$	object introduction
		$m_1.\ell \Leftarrow \varsigma(x : \tau)m_2$	method update
		$m.\ell$	method invocation

where for each $i \in \mathbb{N}$, $x_i \in U$, $X \in V$, $\sigma, \tau_i \in \widehat{\mathcal{T}}$, and for each $i \in I \subseteq \mathbb{N}$ $\ell_i \in L$.

Once we have the type judgements, we can define term contexts (definition 2.4) and then term judgements (definition 2.5). As one would expect, terms are closed under substitution. That is, if $\Theta, \langle \Gamma, x : \tau'\rangle \vdash t : \tau$ and $\Theta, \Gamma \vdash t' : \tau'$ are derivable then so is $\Theta, \Gamma \vdash t\{t'/x\} : \tau$.

Convention 2.1. *We say a preterm $m \in \widehat{\mathcal{L}}$ is well-typed if there exists well-formed contexts Θ, Γ and a type judgement $\Theta \vdash \tau$ such that $\Theta, \Gamma \vdash m : \tau$ is derivable. We*

Definition 2.3 (Types and Type Contexts in $\mathbf{S}^-$)

Type Contexts
Type contexts are generated by the following rules

$$\textsc{TyCon Empty}\ \vdash \Diamond \qquad \textsc{TyCon X}\ \frac{\vdash \Theta}{\vdash \langle \Theta, X\rangle}\ \text{where}\ X \in V, X \notin \Theta$$

Well-formed Types
The typing judgments $\Theta \vdash \tau$ are those generated by the following rules:

$$\textsc{Type X}\ \frac{\vdash \Theta}{\Theta \vdash X}\ \text{where}\ X \in \Theta \qquad \textsc{Type Unit}\ \frac{\vdash \Theta}{1} \qquad \textsc{Type Fun}\ \frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1 \to \tau_2}$$

$$\textsc{Type Object}\ \frac{\Theta, X \vdash \tau_i \qquad i \in I}{\Theta \vdash Obj(X)[\ell_i : \tau_i(X)]^{i\in I}} \qquad \frac{\Theta \vdash \tau_i \qquad i \in I}{\Theta \vdash \boxdot_{i\in I}\tau_i}\ \text{where}\ \boxdot \in \{\textstyle\prod, \coprod\}$$

We let $\mathcal{T}$ denote the set of well-formed types.

let $\mathcal{L}$ denote the set of well-typed terms up to α-equivalence and permutations of method labels.

Example 2.3. *A point whose value is* 0 *and whose bump method adds* 1 *to the value can be represented in* $\mathbf{S}^-$ *as*

$$\begin{aligned} p \triangleq Obj(X = Point)[\quad val &= \varsigma(x : X)0, \\ bump &= \varsigma(x : X)x.val \Leftarrow \varsigma(y : X)x.val + 1\ \ ] \end{aligned}$$

Unlike Java, $\mathbf{S}^-$ makes no distinction between objects and classes. Therefore, a class is represented by an object, which can be cloned or copied into new objects which will (initially at least) have the same methods. There are other differences: object calculus allows methods to be updated, which is impossible in Java, and $\mathbf{S}^-$ has no imperative features. Since we have method updates, there is no need to have separate attributes. Attributes, like *val*, are instead identified with method bodies in which the self variable does not occur.

Definition 2.4 (Term contexts for S^-)

Well-formed term contexts are given by the rules

$$\frac{\vdash \Theta}{\Theta \vdash \Diamond}\ \textsc{Con Empty} \qquad \frac{\Theta \vdash \tau, \Gamma}{\Theta \vdash \langle \Gamma, x : \tau \rangle}\ \textsc{Con x} \text{ where } x \in U, x \notin \Gamma$$

where $\Diamond$ is the empty sequence.

We recall some standard substitution lemmas (for a proof, see e.g. Barendregt [3] or Sørensen et al [24]), where we use notation *FV* for free variables of a term and similarly *FTV* for type variables (both for terms and types), and write $\equiv$ for syntactical equality.

Lemma 2.1 (Substitution commutativity)**.**

(i) If $X \not\equiv Y$, $X \notin FTV(\tau_2)$, then the following is true:

$$\tau\{\tau_1/X\}\{\tau_2/Y\} \equiv \tau\{\tau_2/Y\}\{\tau_1\{\tau_2/Y\}/X\}$$

(ii) If $x \not\equiv y$, $x \notin FV(t_2)$, then the following is true:

$$t\{t_1/x\}\{t_2/y\} \equiv t\{t_2/y\}\{t_1\{t_2/y\}/x\}$$

(iii) If $X \not\equiv Y$, $X \notin FTV(\tau_2)$, then the following is true:

$$t\{\tau_1/X\}\{\tau_2/Y\} \equiv t\{\tau_2/Y\}\{\tau_1\{\tau_2/Y\}/X\}$$

Lemma 2.2 (Substitutivity)**.** *The following rules are valid:*

$$\frac{\langle X, \Theta \rangle \vdash \sigma \quad \Theta \vdash \sigma'}{\Theta \vdash \sigma\{\sigma'/X\}}\ \textsc{type subst} \qquad \frac{\Theta, \langle x : \sigma, \Gamma \rangle \vdash t : \sigma' \quad \Theta, \Gamma \vdash s : \sigma}{\Theta, \Gamma \vdash t\{s/x\} : \sigma'}\ \textsc{subst term}$$

$$\frac{\langle X, \Theta \rangle, \Gamma \vdash t : \sigma' \quad \Theta \vdash \sigma''}{\Theta, \Gamma \vdash t\{\sigma''/X\} : \sigma'\{\sigma''/X\}}\ \textsc{subst type}$$

Definition 2.5 (Typing judgments for $\mathbf{S}^-$)

VAL OBJECT
$$\frac{\sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \quad \Theta, \langle \Gamma, x_i : \sigma\rangle \vdash b_i\{\!|\sigma|\!\} : \tau_i\{\!|\sigma|\!\} \quad \forall i \in I}{\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} : \sigma}$$

VAL SELECT
$$\frac{o \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} \quad \Theta, \Gamma \vdash o : \sigma \quad \forall i \in I}{\Theta, \Gamma \vdash o.\ell_i\{\!|\sigma/X|\!\} : \tau_i\{\!|\sigma|\!\}}$$

VAL UPDATE
$$\frac{\sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \quad \Theta, \Gamma \vdash m : \sigma \quad \Theta, \langle \Gamma, x : \sigma\rangle \vdash b\{\!|\sigma|\!\} : \tau_j\{\!|\sigma|\!\} \quad j \in I}{\Theta, \Gamma \vdash m.\ell_j \Leftarrow \varsigma(x : \sigma)b : \sigma}$$

VAL X
$$\frac{\Theta \vdash \langle \Gamma, x_i : \tau_i, ...\rangle}{\Theta, \langle \Gamma, x_i : \tau_i, ...\rangle \vdash x_i : \tau_i}$$

VAL UNIT
$$\Theta, \Gamma \vdash \star : 1$$

VAL PRODUCT
$$\frac{\Theta, \Gamma \vdash a : \prod_{i\in I} \tau_i \quad j \in I}{\Theta, \Gamma \vdash \pi_j\, a : \tau_j}$$

VAL PAIR
$$\frac{\Theta, \Gamma \vdash a_1 : \tau_1 \qquad \ldots \qquad \Theta, \Gamma \vdash a_n : \tau_n}{\Theta, \Gamma \vdash \langle a_1, \ldots, a_n\rangle : \prod_{i\in I} \tau_i}$$

VAL SUM
$$\frac{\Theta, \Gamma \vdash a : \tau_j \quad j \in I}{\Theta, \Gamma \vdash \iota_i\, a : \coprod_{i\in I} \tau_i}$$

VAL CASE
$$\frac{\Theta, \Gamma \vdash m : \coprod_{i\in I} \sigma_i \quad \Theta, \langle \Gamma, x_j : \sigma_j\rangle \vdash m_j : \tau \quad j \in I}{\Theta, \Gamma \vdash case(m, x_1.m_1, ..., x_n.m_n) : \tau}$$

VAL EVAL
$$\frac{\Theta, \Gamma \vdash m_1 : \tau_1 \to \tau_2, m_2 : \tau_1}{\Theta, \Gamma \vdash m_1(m_2) : \tau_2}$$

VAL FUN
$$\frac{\Theta, \langle \Gamma, x : \tau_1\rangle \vdash b : \tau_2}{\Theta, \Gamma \vdash \lambda x : \tau_1.b : \tau_1 \to \tau_2}$$

Proof. By induction on the derivation of a well-formed type $\langle X, \Theta\rangle \vdash \sigma$ and well-typed term $\Theta, \langle x : \sigma, \Gamma\rangle \vdash t : \sigma'$ or $\langle X, \Theta\rangle, \Gamma \vdash t : \sigma'$, respectively. For (TYPE SUBST) and (SUBST TERM) this is similar to the theorems given by Abadi et al [1] including

for their calculi $\mathbf{Ob}_{1<:\mu}$ and $\mathbf{S}$. The sum type cases are covered as in e.g. Pierce [18] or Sørensen et al [24].

The rule (SUBST TYPE) is proved by induction on the derivation of $\langle\Theta, X\rangle, \Gamma \vdash t : \sigma'$ for an arbitrary but fixed substitution $\{\sigma''/X\}$ such that $\Theta \vdash \sigma''$. The basis is vacuous since a type substitution acts on term variables as identity. The only cases where free type variables can occur in terms is in an object type σ occurring in a term of the form $Obj(Y = \sigma)[\ell_i = \varsigma(x_i : Y)b_i]^{i\in I}$. For $t = Obj(Y = \sigma)[\ell_i = \varsigma(x_i : Y)b_i]^{i\in I}$ we have either that X is free in σ and thus $X \not\equiv Y$, or else we are done. Suppose $X \not\equiv Y$. Then by induction hypothesis, we have for each $i \in I$ that the following holds:

$$\frac{X, \Theta, \Gamma \vdash b_i\{\sigma/Y\} : \tau_i\{\sigma/Y\} \quad \Theta \vdash \sigma''}{\Theta, \Gamma \vdash b_i\{\sigma/Y\}\{\sigma''/X\} : \tau_i\{\sigma/Y\}\{\sigma''/X\}}$$

Via lemma 2.1 we have that the following rule is also derivable (the condition on the free type variables is ensured by the variable convention):

$$\frac{X, \Theta, \Gamma \vdash b_i\{\sigma/Y\} : \tau_i\{\sigma/Y\} \quad \Theta \vdash \sigma''}{\Theta, \Gamma \vdash b_i\{\sigma''/X\}\{\sigma\{\sigma''/X\}/Y\} : \tau_i\{\sigma''/X\}\{\sigma\{\sigma''/X\}/Y\}}\ \dagger$$

Note that the premises of these rules must hold by an argument that uses a generation lemma. Moreover, the rule (TYPE SUBST) gives that $\Theta \vdash \sigma\{\sigma''/X\}$ is an object type. It remains to show that

$$\Theta, \Gamma \vdash t\{\sigma''/X\} : \sigma\{\sigma'/X\}$$

i.e. the conclusion of the (SUBST TYPE) rule in this case. For this note that the premises for the rule (VAL OBJECT) are precisely the conclusion of the derived rule (†) above, so we are done. The situation for (VAL UPDATE) is similar, and (VAL SELECT) is trivial. □

Operational Semantics

We have now defined the language of $\mathbf{S}^-$, and will give it an operational semantics. The semantics is call-by-value and, in particular, each component of a product must have a value for a projection of the product to attain a value. The rules for the non-object part of the calculus are standard while we feel that those for the introduction, eliminating, and updating objects are reasonable, e.g. one does not reduce under the binder in object intro terms and hence all object intro terms are

values. This feeling is reinforced by the results we derive later on soundness and adequacy. The values (or normal/canonical forms) are as follows:

$$\begin{array}{rcl} v & ::= & x \mid 1 \mid \iota_i\, v \mid \langle v_1, \ldots, v_n \rangle \mid \lambda x : \tau.m \mid \\ & & Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)m_i]^{i \in I} \end{array}$$

The actual operational rules are given in definition 2.6. Note that the values are precisely the terms v such that $v \rightsquigarrow v$, where $\rightsquigarrow$ means the reduction relation. This is the statement that values are irreducible in a formal sense. A program p is a term such that for some type τ we have $\vdash p : \tau$, i.e. a well-typed term with empty contexts. The key theorem which means that the implementation of the calculus, as given by its operational semantics, respects compile time type information is the preservation of types as shown in the next theorem.

Theorem 2.1 (Subject Reduction)**.** *If t is a well-typed term $\Theta, \Gamma \vdash t : \tau$ such that $t \rightsquigarrow t'$, then $\Theta, \Gamma \vdash t' : \tau$.*

Proof. The proof is by induction on the derivation of $t \rightsquigarrow t'$ and is fairly routine. Suppose $\Theta, \Gamma \vdash t : \tau$ and $t \rightsquigarrow t'$. We have omitted trivial cases:

Case (Red Case): We have $t = case(m, x_1.m_1, \ldots, x_n.m_n)$ and $\Theta, \Gamma \vdash t : \tau$. Since t is well-typed we have $\Theta, \Gamma \vdash m : \coprod_{i \in I} \sigma_i$ and $\Theta, \langle \Gamma, x : \sigma \rangle \vdash m_i : \tau$ for $i \in I$. We have subderivation $m \rightsquigarrow \iota_k v$ and $m_k\{\!|v/x_k|\!\} \rightsquigarrow t'$. But by induction hypothesis this means, by substitutivity, $t' : \tau$.

Case (Red Product): We have $t = \pi_i(m)$ and $\Theta, \Gamma \vdash t : \tau$, which is to say $m = \langle a_1, \ldots, a_n \rangle$ for some $\Theta, \Gamma \vdash a_i : \tau_i$. The result follows by induction hypothesis on the required component.

Case (Red Eval): We have $t = m_1(m_2)$ and $\Theta, \Gamma \vdash t : \tau_2$. Therefore $\Theta, \Gamma \vdash m_1 : \tau_1 \to \tau_2$ and $\Theta, \Gamma \vdash m_2 : \tau_1$. That is to say $m_1 = \lambda x : \tau_2.b$. Now for $m_2 \rightsquigarrow v$ we have $\Theta, \Gamma \vdash b\{\!|v/x|\!\} : \tau_2$ and by induction hypothesis $t' : \tau_2$ as required.

Case (Val Select): we have $t = m.\ell_i$ and $\Theta, \Gamma \vdash t : \tau_i$ for $m = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I}$ and $\Theta, \Gamma \vdash m : \sigma$ with $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i \in I}$. For $m \rightsquigarrow v'$ and $b_i\{\!|v', \sigma|\!\} \rightsquigarrow t'$ we have $\Theta, \Gamma \vdash t' : \tau_i\{\!|\sigma|\!\}$ by induction hypothesis.

Case (Val Object): we have $t = m.\ell_j \Leftarrow \varsigma(x : \sigma).b$ and $t : \sigma$. Since $\Theta, \langle \Gamma, x : \sigma \rangle \vdash b_j\{\!|\sigma|\!\} : \tau_j\{\!|\sigma|\!\}$ we have $\Theta, \Gamma \vdash t' : \sigma$ as required. □

Definition 2.6 (Operational semantics for S^-)

$$\textsc{Red x}\quad x \rightsquigarrow x \qquad \textsc{Red Unit}\quad \star \rightsquigarrow \star$$

$$\textsc{Red Pair}\quad \frac{m_1 \rightsquigarrow v_1 \quad \dots \quad m_n \rightsquigarrow v_n}{\langle m_1, \dots, m_2 \rangle \rightsquigarrow \langle v_1, \dots, v_n \rangle}$$

$$\textsc{Red Proj}\quad \frac{m \rightsquigarrow \langle v_1, \dots, v_n \rangle \qquad 1 \leq i \leq n}{\pi_i(m) \rightsquigarrow v_i} \qquad \textsc{Red Sum}\quad \frac{m \rightsquigarrow v}{\iota_j\, m \rightsquigarrow \iota_j\, v}$$

$$\textsc{Red Case}\quad \frac{m \rightsquigarrow \iota_j(v) \qquad m_j\{\!\{v/x_j\}\!\} \rightsquigarrow v' \qquad j \in [1,n]}{case(m, x_1.m_1, ..., x_n.m_n) \rightsquigarrow v'} \qquad \textsc{Red Fun}\quad \lambda x : \tau.m \rightsquigarrow \lambda x : \tau.m$$

$$\textsc{Red Eval}\quad \frac{m_1 \rightsquigarrow \lambda x : \tau.b \qquad m_2 \rightsquigarrow v \qquad b\{\!\{v/x\}\!\} \rightsquigarrow v'}{m_1(m_2) \rightsquigarrow v'} \qquad \textsc{Red Object}\quad \frac{v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I}}{v \rightsquigarrow v}$$

$$\textsc{Red Select}\quad \frac{v' \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I} \qquad m \rightsquigarrow v' \qquad b_i\{\!\{v', \sigma\}\!\} \rightsquigarrow v}{m.\ell_i \rightsquigarrow v}$$

$$\textsc{Red Update}\quad \frac{v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I} \qquad m \rightsquigarrow v}{m.\ell_j \Leftarrow \varsigma(x : \sigma)b\{\!\{\sigma\}\!\} \rightsquigarrow Obj(X = \sigma)[\ell_i = \varsigma(x : X)b_i, \ell_j = \varsigma(x : X)b]^{i \in I - \{j\}}}$$

where in the last rule we delete the j'th method from v and then add the updated method $\ell_j = \varsigma(x : X)b$.

Definition 3.1 (Eager FPC)

$$\textsc{Val In}\ \frac{X \notin \Theta \quad \Theta, \Gamma \vdash m : \tau\{\mu X.\tau/X\}}{\Theta, \Gamma \vdash in_{\mu X.\tau}(m) : \mu X.\tau} \qquad \textsc{Val Out}\ \frac{\Theta, \Gamma \vdash m : \mu X.\tau}{\Theta, \Gamma \vdash out(m) : \tau\{\mu X.\tau/X\}} \qquad \textsc{Type Rec}\ \frac{\langle \Theta, X\rangle \vdash \tau}{\Theta \vdash \mu X.\tau}$$

$$\textsc{Red Inn}\ \frac{e \rightsquigarrow v}{in(e) \rightsquigarrow in(v)} \qquad \textsc{Red Out}\ \frac{e \rightsquigarrow in(v)}{out(e) \rightsquigarrow v}$$

3 FPC

Our intention is to interpret object types as solutions of certain recursive equations. We do this syntactically by translating the object calculus into the FPC. In previous work [11], we have done it also semantically by giving denotational models for the object calculus using some sophisticated categorical model, in which case the equations become domain equations rather than, like here, type equations in the metalanguage FPC.

The target calculus of recursive types is known in the semantics literature as FPC. This system is originally due to Plotkin [19], but detailed expositions are given e.g. by Gunter [14] and Fiore [8]. FPC intuitively arises from $\mathbf{S}^-$ by deleting the types and terms related to objects and inserting types and terms related to fixed points of mixed variant type constructors. Thus FPC uses the same countable supplies U and V of type and term variables. We summarise the formal rules in definition 3.1.

The notions of substitution, α-congruence, contexts, well-formed types, are all identical, except that we replace object type formation with the following rule for well formed recursive types. The preterms of FPC are exactly those of $\mathbf{S}^-$, omitting all terms derived from the object formation rule, method updates, and method invocation, and adding to the grammar terms of the form $in_{\mu X.\tau}$ for μ-introduction (Val In) and *out* for μ-elimination (Val Out). The term judgments for FPC are similarly obtained from those of $\mathbf{S}^-$, but the Val Object, Val Select and Val Update rules are replaced by two rules for typing recursive types. Finally, the

Definition 3.2 (Lazy FPC)

Operational Semantics
The following rules take the place of Red Proj, Red Case, and Red Eval and all other rules are the same:

$$\textsc{RedL Eval}\quad \frac{m_1 \rightsquigarrow \lambda x:\tau.b \quad b\{\!|m_2/x|\!\} \rightsquigarrow v}{m_1(m_2) \rightsquigarrow v} \qquad \textsc{RedL Proj}\quad \frac{m \rightsquigarrow \langle m_1,\ldots,m_n\rangle \quad m_i \rightsquigarrow v}{\pi_i(m) \rightsquigarrow v}$$

$$\textsc{RedL Case}\quad \frac{m \rightsquigarrow \iota_j(k) \quad m_j\{\!|k/x_j|\!\} \rightsquigarrow v' \quad j \in [1,n]}{case(m, x_1.m_1, .., x_n.m_n) \rightsquigarrow v'}$$

operational semantics of FPC is obtained by deleting Val Object terms as values, removing the operational rules for Red Object, Red Select and Red Update and adding the following values and rules from definition 3.1 to cope with recursive types.

$$v ::= \ldots \mid in_{\mu X.\tau}(v)$$

In addition to this *eager* (call-by-value) version of FPC, we will briefly also recall the *lazy* (call-by-name) operational semantics that can be given to this language.

Under a lazy semantics, we have more values than we had in the eager semantics. If $t_i, \lambda x.m$ are closed terms, the values now also include:

$$v ::= \ldots \quad \iota_j\, t \mid \lambda x.m \mid \langle t_1, ..., t_n\rangle \mid in_{\mu X.\tau}(v)$$

4 Translating Object Calculus into FPC

This section contain a translation of $\mathbf{S}^-$ into (lazy) FPC. This translation is at the level of types, terms and operational semantics and we find an excellent fit whereby the operational semantics for FPC is both sound and complete. This allows us to transport the well-understood theory of FPC, in particular its denotational models (e.g. [8, 26, 14]), to $\mathbf{S}^-$.

The encoding of objects uses recursive types contrary to e.g. the recursive record semantics in the literature, e.g. [5, 1]. Notably, the recursive record semantics would give the following interpretation of the p : *Point* object given in the previous examples:

$$p = Y\ \lambda p.\langle 0, \langle \pi_1\ p + 1, \pi_2\ p\rangle\rangle$$

where $Y : (\tau{\to}\tau){\to}\tau$ is a fixed point combinator (which can be encoded into FPC). The type of p is $\mu X.Int \times X$, but as seen in this example we cannot replace the first component of p without giving a completely new definition of p. We will give p the type $\mu X.(X{\to}Int) \times (X{\to}X)$. This means that p is denoted simply by a product which enjoys the ordinary projections on each component.

Recall the key feature of the encoding chosen for this work is that it reflects our intuition that the object types of $\mathbf{S}^-$ are fixed points of recursive type equations. More specifically, the recursion is over the self-parameter which occurs both covariantly and contravariantly. This intuition is clearly seen in the Val Object typing rule for $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$ which suggests the i'th method will consume the self-parameter, which has type σ, to produce something of type τ_i where σ may occur free, e.g. also be produced. Thus, intuitively, the interpretation of σ should satisfy

$$\lceil\sigma\rceil \cong \lceil\sigma\rceil{\to}\lceil\tau_1\rceil \times \cdots \times \lceil\sigma\rceil{\to}\lceil\tau_n\rceil$$

where, as we mentioned above, each of the τ_i may contain σ. Hence the interpretation of σ should be the fixed point $\mu X.X{\to}\lceil\tau_1\rceil \times \cdots \times X{\to}[\![\tau_n]\!]$ where the τ_i may contain X free. Thus the interpretations of the object types *Point* and *UnLam* are

$$\begin{array}{lcl} \lceil Point\rceil & = & \mu X.(X{\to}X) \times (X{\to}Int) \\ \lceil UnLam\rceil & = & \mu X.X{\to}X \end{array}$$

Note the interpretation of this example shows how the type of untyped lambda terms arises naturally as an object. We do not need to translate type contexts since we have identified the sets of type variables. We thus begin by translating well-formed $\mathbf{S}^-$ types into FPC-types:

$$\begin{array}{lcl} \lceil X\rceil & \triangleq & X \\ \lceil 1\rceil & \triangleq & 1 \\ \lceil A{\to}B\rceil & \triangleq & \lceil A\rceil{\to}\lceil B\rceil \\ \lceil \prod_{i\in I} A_i\rceil & \triangleq & \prod_{i\in I}\lceil A_i\rceil \\ \lceil \coprod_{i\in I} A_i\rceil & \triangleq & \coprod_{i\in I}\lceil A_i\rceil y \end{array}$$

As mentioned above, the translation of object types is into the solution of a mixed variance recursive type equation.

$$\lceil Obj(X)[\ell_i : \tau_i(X)]^{i\in I}\rceil \quad \triangleq \quad \mu X.X{\to}\lceil\tau_1\rceil \times ... \times X{\to}\lceil\tau_n\rceil$$

Notice that the translation of types respects substitutions, that is $\lceil\tau[\sigma/X]\rceil = \lceil\tau\rceil[\lceil\sigma\rceil/X]$. We can now syntactically translate (term) contexts:

$$\begin{array}{lcl}\lceil\Theta \vdash \Diamond\rceil & \triangleq & \Theta \vdash \Diamond \\ \lceil\Theta \vdash \langle\Gamma, x : \tau\rangle\rceil & \triangleq & \Theta \vdash \langle\lceil\Gamma\rceil, x : \lceil\tau\rceil\rangle\end{array}$$

Now we extend our translation to typing judgments. The translations of terms in the intersection of the calculi are just by induction.

$$\begin{array}{lcl}\lceil\Theta,\Gamma \vdash x_i : \tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash x_i : \lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \star : 1\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \star : 1 \\ \lceil\Theta,\Gamma \vdash \langle m_0,\ldots,m_n\rangle : \tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \langle\lceil m_0\rceil,\ldots,\lceil m_n\rceil\rangle : \lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \pi_i\ m : \tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \pi_i\ \lceil m\rceil : \lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \iota_j\ m : \tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \iota_j\ \lceil m\rceil : \lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash m_0\ m_1 : \tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \lceil m_0\rceil\lceil m_1\rceil : \lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \lambda(x:\sigma)m : \tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \lambda(x : \lceil\sigma\rceil)\lceil m\rceil : \lceil\tau\rceil\end{array}$$

$$\begin{array}{l}\lceil\Theta,\Gamma \vdash case(m_0, x_1.m_1,\ldots,x_n.m_n) : \tau\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash case(\lceil m_0\rceil, x_1.\lceil m_1\rceil,\ldots,x_n.\lceil m_n\rceil) : \lceil\tau\rceil\end{array}$$

Based on translation of object types, we can translate object introductions, method update, and object elimination (method invocation) in the obvious way:

$$\begin{array}{l}\lceil\Theta,\Gamma \vdash m : \sigma\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash in(\langle\lambda x : \lceil\sigma\rceil.\lceil b_1\{\!\{\sigma\}\!\}\rceil, ..., \lambda x : \lceil\sigma\rceil.\lceil b_n\{\!\{\sigma\}\!\}\rceil\rangle) : \lceil\sigma\rceil \\ \lceil\Theta,\Gamma \vdash m.\ell_i\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash (\pi_i\ \alpha)(\lceil m\rceil) : \lceil\tau_i\{\!\{\sigma\}\!\}\rceil \\ \lceil\Theta,\Gamma \vdash m.\ell_j \Leftarrow \varsigma(x:\sigma)b\{\!\{\sigma\}\!\}\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash in(\langle\pi_1\alpha,\ldots,\pi_{j-1}\alpha,\lambda x : \lceil\sigma\rceil.\lceil b\rceil\{\!\{\sigma\}\!\},\pi_{j+1}\alpha,\ldots, \\ \quad \pi_n\alpha\rangle) : \lceil\sigma\rceil \\ \text{where} \\ \quad \alpha \equiv out(\lceil m\rceil) \\ \quad m \equiv Obj(X=\sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} \\ \quad \sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I}\end{array}$$

Here π_{l_j} is the projection of a labeled product.

Let F_i be a type expression for each $i \in I$, $I = \{1, ..., n\}$. We have interpreted object types as $\mu X.(X{\to}F_1\ X) \times ... \times (X{\to}F_n\ X)$ (where for method invocation, self is applied *after* projection) rather than $\mu X.(X{\to}F\ X)$ where $F = \prod_{i\in I} F_i$. This is because the latter interpretation would break soundness. Consider, for example the interpretation of method invocation. For soundness, we need to prove that π_{ℓ_j} applied to a term reduces to a value in the case when $m.\ell_j$ reduces to a $\mathbf{S}^-$-value. However, the eager operational semantics of projection in FPC requires that all components of the tuple have a value, and we can easily construct an object for which this would not hold. However, given a lazy operational semantics for FPC (e.g. Winskel [26]) this argument would no longer apply, since partially evaluated terms (in particular products) are included as values.

We will now prove an important lemma which shows that our interpretation function $\lceil - \rceil$ is substitutive on terms (it is trivially substitutive on types):

Lemma 4.1. $\lceil m\{\!|v/x, \sigma/\gamma|\!\}\rceil = \lceil m\rceil\{\!|\lceil v\rceil/x, \lceil\sigma\rceil/\gamma|\!\}$

Proof. The proof is by induction on the image of terms under $\lceil - \rceil$. We need only consider object intro, elim, update under $\lceil - \rceil$:

$$
\begin{array}{ll}
 & \lceil Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}\{\!|v/x, \sigma/\gamma|\!\} : \delta\rceil \\
= & \text{by definition} \\
 & in(\langle \lambda x : \tau.\lceil b_1\{\!|\delta, v/x, \sigma/\gamma|\!\}\rceil, ..., \lambda x : \tau.\lceil b_n\{\!|\delta, v/x, \sigma/\gamma|\!\}\rceil\rangle) \\
= & \text{by induction hypothesis on } b_i \\
 & in(\langle \lambda x : \tau.\lceil b_1\{\!|\delta|\!\}\rceil\{\!|\lceil v\rceil/x, \lceil\sigma\rceil/\gamma|\!\}, ..., \lambda x : \tau.\lceil b_n\{\!|\delta|\!\}\rceil\{\!|\lceil v\rceil/x, \lceil\sigma\rceil/\gamma|\!\}\rangle) \\
= & \text{since } \lceil - \rceil \text{ is substitutive on } in\text{, tupling, and } \lambda \\
 & in(\langle \lambda x : \tau.\lceil b_1\{\!|\delta|\!\}\rceil, ..., \lambda x : \tau.\lceil b_n\{\!|\delta|\!\}\rceil\rangle)\{\!|\lceil v\rceil/x, \lceil\sigma\rceil/\gamma|\!\} \\
= & \text{by definition} \\
 & \lceil Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}\rceil\{\!|\lceil v\rceil/x, \lceil\sigma\rceil/\gamma|\!\}
\end{array}
$$

The situation is similar for method invocation and method update, in that $\lceil - \rceil$ will be substitutive on sub-terms formed according to the rules of FPC. □

Our translation preserves types:

Lemma 4.2. *If* $\Theta, \Gamma \vdash t : \tau$ *then* $\lceil\Theta\rceil, \lceil\Gamma\rceil \vdash \lceil t\rceil : \lceil\tau\rceil$

Proof. The proof is by induction on well-typed terms. We consider only Val Object, Val Select, and Val Update, since the other cases follow by induction. Suppose $\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} : \sigma$ where $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$.

We must show that $\Theta, \lceil\Gamma\rceil \vdash in(\langle \lambda x : \lceil\sigma\rceil.\lceil b_1\{\!|\sigma|\!\}\rceil, ..., \lambda x : \lceil\sigma\rceil.\lceil b_n\{\!|\sigma|\!\}\rceil\rangle) : \lceil\sigma\rceil$ where $\lceil\sigma\rceil = \mu X.X{\to}\lceil\tau_1\rceil \times ... \times X{\to}\lceil\tau_n\rceil$.

This follows from the premises of the (μI) rule holds, i.e. if

$$\begin{aligned}\Theta, \lceil\Gamma\rceil \vdash \quad & in(\langle \lambda x : \lceil\sigma\rceil.\lceil b_1\rceil\{\!|\lceil\sigma\rceil|\!\}, ..., \lambda x : \lceil\sigma\rceil.\lceil b_n\rceil\{\!|\lceil\sigma\rceil|\!\}\rangle) : \\ & X{\to}\lceil\tau_1\rceil \times ... \times X{\to}\lceil\tau_n\rceil \\ & \quad \{\!|\mu(X)X{\to}\lceil\tau_1\rceil \times ... \times X{\to}\lceil\tau_n\rceil/X|\!\}\end{aligned}$$

The premises of Val Object asserts $\Theta, \langle\Gamma, x_i : \sigma\rangle \vdash b_i\{\!|\sigma|\!\} : \tau_i\{\!|\sigma|\!\}$ which by induction hypothesis means $\Theta, \langle\lceil\Gamma\rceil, x_i : \lceil\sigma\rceil\rangle \vdash \lceil b_i\{\!|\sigma|\!\}\rceil : \lceil\tau_i\{\!|\sigma|\!\}\rceil$. We then have a FPC-term of the required type from the bodies $\lceil b_i\{\!|\sigma|\!\}\rceil = \lceil b_i\rceil\{\!|\lceil\sigma\rceil|\!\}$ by the substitution lemma. The Val Fun and Val Proj rules gives us $\langle \lambda x : X.\lceil b_1\rceil\{\!|X|\!\}, ..., \lambda x : X.\lceil b_n\rceil\{\!|X|\!\}\rangle\{\!|\lceil\sigma\rceil/X|\!\}$. Finally Val In gives us the required type.

The case for Val Update is almost identical. For Val Select we assume $\Theta, \Gamma \vdash m : \sigma$ where $m = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$ and $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$ and consider $\Theta, \Gamma \vdash m.\ell_i : \tau_i\{\!|\sigma|\!\}$. We want $\Theta, \lceil\Gamma\rceil \vdash (\pi_{l_i}\ out(\lceil m\rceil))(\lceil m\rceil) : \lceil\tau_i\{\!|\sigma|\!\}\rceil$. By induction hypothesis we have $\Theta, \lceil\Gamma\rceil \vdash \lceil m\rceil : \lceil\sigma\rceil$. Further $\Theta, \lceil\Gamma\rceil \vdash out(\lceil m\rceil) : \lceil\sigma\{\!|\mu X.\sigma/X|\!\}\rceil$ and after projection we have the body b_i of type $\tau_i\{\!|\sigma|\!\}$, and the result follows by applying the induction hypothesis to b_i. □

5 Soundness and Adequacy

We will prove the soundness and adequacy of our translation of $\mathbf{S}^-$ into FPC. This means that the translation of $\mathbf{S}^-$ into FPC is given in such a way that the operational semantics of FPC is strong enough to interpret the operational semantics of $\mathbf{S}^-$ while not being so strong as to give extra computations which were not present in $\mathbf{S}^-$.

We will show that $t \rightsquigarrow v$ implies $\lceil t\rceil \rightsquigarrow \lceil v\rceil$. This establishes that our translation is correct (soundness). We also prove an adequacy result of the operational semantics of $\mathbf{S}^-$. These two results establish that any denotational model of lazy FPC (given along the lines of Plotkin and Fiore [6]) is, via the self-application interpretation, a suitable mathematical setting for object calculus. For example, a category such as **pCPO** immediately gives us a denotational model of object calculus (via e.g. [26]).

In what follows we will let $[\![-]\!]$ denote the interpretation of FPC with respect to some denotational semantics equipped with a notion of totality. Such an interpretation is defined in the usual way by induction on the well-formed types and the well-typed terms, see Fiore and Plotkin [6, 8]. We will speak about a denotation

being "total". As in loc. cit. we require that the interpretation is with respect to a category of partial maps [8], viz. $p\mathcal{K}$. A term $\Theta, \Gamma \vdash t : \tau$ is understood to be interpreted by an indexed family of partial maps $[\![\Gamma]\!]A \rightharpoonup [\![\tau]\!]A$ (where A ranges over objects in $p\mathcal{K}$). The types and type contexts are interpreted by symmetric functors on the ambient category, i.e. self-dual functors of the form $\mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ (henceforth, $\mathcal{C}^{op} \times \mathcal{C}$ is written as $\breve{\mathcal{C}}$). To summarise, we assume that we have indexed partial maps and (symmetric) functors as follows:

$$\begin{array}{lcl} [\![\Theta, \Gamma \vdash t : \tau]\!]_A & : & [\![\Theta \vdash \Gamma]\!](A)_2 \rightharpoonup [\![\Theta \vdash \tau]\!](A)_2 \\ [\![\Theta \vdash \Gamma]\!] & : & p\breve{\mathcal{K}}^{|\Theta|} \to p\breve{\mathcal{K}} \end{array}$$

for $A \in |p\breve{\mathcal{K}}|^{|\Theta|}$ where $|\Theta|$ is the number of type variables appearing in Θ. We wrote $F(X)_2$ for $\Pi_2 \circ F(X)$.

Formally, a partial map is described as a pair $[m, f]$ where m is an admissible mono drawn from a subcategory $\mathcal{D}$ of $\mathcal{K}$. The total maps $[m, f]$ are such partial maps where $m = id$. However, we must identify some representations $[m, f]$ since a single partial map can be written in this form in more than one way. For this, we proceed like Fiore et al [6, 8], and say that partial maps $[m, f]$ and $[n, g]$ describe the same partial map if and only if

$$m = n \circ i \text{ and } f = g \circ i \text{ for some isomorphism } i \, .$$

This immediately gives the formal notion of total maps for $p\mathcal{K}$ as the maps $[id, f]$ up to the equivalence relation induced by such isomorphisms.

We remark that for a domain-theoretic model based on $\mathbf{CPPO}_{\perp!}$ of strict continuous maps and cpos with least elements, totality of a map f is more simply the property that $f(x) = \perp$ implies $x = \perp$ (via the isomorphism between this category and **pCPO** as described e.g. in [8]). Also, for the concrete category **pCPO** of predomains, the usual set-theoretic notion of total partial map can be used, rather than the more general one described above. Moreover, in this case, the notion of partial continuity implies that the domain of definition (admissible monos) must be a Scott-open set [6].

Definition 5.1 (Computational Soundness)**.** *We say that an interpretation* $\lceil - \rceil$ *of* $\mathbf{S}^-$ *into FPC is computationally sound if, for every* $\Theta, \Gamma \vdash o : \tau$ *such that* $o \rightsquigarrow v$ *where* v *is a value, we have that* $[\![\lceil \Theta, \Gamma \vdash o : \tau \rceil]\!]$ *is a total map.*

Definition 5.2 (Computational Adequacy)**.** *We say that an interpretation* $\lceil - \rceil$ *of* $\mathbf{S}^-$ *into FPC is computationally adequate if given any* $\Theta, \Gamma \vdash o : \tau$*, whenever* $[\![\lceil \Theta, \Gamma \vdash o : \tau \rceil]\!]$ *is a total map, we also have that* $o \rightsquigarrow v$ *for some value* v*.*

In order to establish computational soundness and adequacy for the interpretation $\lceil - \rceil$, we require the following theorems. From these, the required result of computational soundness and adequacy follows immediately as a corollary simply by composing the relations appropriately, and by observing that $\lceil v \rceil$ is a value in lazy FPC whenever v is a value in $\mathbf{S}^-$.

Theorem 5.1. *The interpretation $\lceil - \rceil$ has the property that $t \rightsquigarrow v$, then $\lceil t \rceil \rightsquigarrow \lceil v \rceil$*

Proof. We only check the derivation rules Val Object, Val Select, and Val Update, since the result follows from induction for the other derivation rules. The translation of an Val Object term is an FPC value and hence the theorem holds for terms arising as the result of the Val Object rule.

For Val Select, suppose $m \rightsquigarrow v'$ and $b_i\{\!\{v', \sigma\}\!\} \rightsquigarrow v$, where $v' = Obj(X = \sigma)[l_1 = \varsigma(x_i : X).b_i]^{i \in I}$. We want to show that $\lceil m.\ell_i \rceil \rightsquigarrow \lceil v \rceil$. By induction $\lceil m \rceil \rightsquigarrow \lceil v' \rceil$ and hence

$$\pi_i(out\lceil m \rceil) \rightsquigarrow \lambda x : \lceil \sigma \rceil . b_i$$

Again, by induction, $\lceil m \rceil \rightsquigarrow \lceil v' \rceil$ and $\lceil b_i\{\!\{v', \sigma\}\!\} \rceil \rightsquigarrow \lceil v \rceil$. The result then follows by the substitution lemma since $\lceil b_i\{\!\{v', \sigma\}\!\} \rceil = \lceil b_i \rceil\{\!\{\lceil v' \rceil, \sigma\}\!\}$.

For method update, suppose $m \rightsquigarrow v$ where $v = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I}$. In order to prove $\lceil m.\ell_j \Leftarrow \varsigma(x : \sigma).b \rceil \rightsquigarrow \lceil v' \rceil$ where $v' = Obj(X = \sigma)[l_i = \varsigma(x : X).b_i, l_j = \varsigma(x : X).b]^{i \in I}$ we must prove that

$$\begin{array}{l} in(\langle \pi_1\alpha, ..., \lambda x : \lceil \sigma \rceil . \lceil b \rceil, ..., \pi_n\alpha \rangle) \rightsquigarrow \\ in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \rceil \rangle) \end{array}$$

where $\alpha = out(\lceil m \rceil)$. By induction

$$\lceil m \rceil \rightsquigarrow \lceil v \rceil = in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \rceil \rangle)$$

and hence $\pi_i\alpha \rightsquigarrow \lambda x : \lceil \sigma \rceil . \lceil b_i \rceil$ as required. □

Corollary 5.1. *If an FPC model has the property that $t \rightsquigarrow v$ implies $[\![t]\!] = [\![v]\!]$, then $\mathbf{S}^-$ has this property for the same model via the translation $\lceil - \rceil$.*

We proceed with adequacy which shows that the operational semantics of FPC is not too strong with respect to the operational semantics of $\mathbf{S}^-$.

Theorem 5.2. *The interpretation $\lceil - \rceil$ has the property that if $\lceil t \rceil \rightsquigarrow v$, then there is a v' such that $t \rightsquigarrow v'$ and $\lceil v' \rceil = v$*

Proof. The proof is by induction on the derivation tree for $\lceil t \rceil \rightsquigarrow v$. If t is a variable or any of the terms related to the standard type constructors of λ-calculus, then the proof is as expected. If t is a VAL OBJECT term, then both t and $\lceil t \rceil$ are values and hence the theorem trivially holds. There are two more cases:

If t is given by VAL SELECT, say $t \equiv m.\ell_j$, then $\lceil t \rceil \rightsquigarrow v$ must have the following form:

$$\dfrac{\dfrac{\dfrac{\lceil m \rceil \rightsquigarrow in\langle \ldots, \lambda x.b, \ldots \rangle}{out\lceil m \rceil \rightsquigarrow \langle \ldots, \lambda x.b, \ldots \rangle}}{\pi_j out\lceil m \rceil \rightsquigarrow \lambda x.b} \qquad \lceil m \rceil \rightsquigarrow u \qquad \lceil b\{u/x, \sigma\} \rceil \rightsquigarrow v}{(\pi_j out\lceil m \rceil)(\lceil m \rceil) \rightsquigarrow v}$$

We see that the derivation for $\lceil b\{u/x, \sigma\} \rceil \rightsquigarrow v$ is contained in the above derivation. Therefore we can apply the induction hypothesis to it, and also to the term m. The premises of the rule VAL SELECT are now satisfied, and we can conclude that $t \rightsquigarrow \xi$ for value ξ. It remains to be shown that $\lceil \xi \rceil = v$, but this is just the induction hypothesis for $\lceil b\{u/x\} \rceil$.

Finally, let $t = m.\ell \Leftarrow \varsigma(x : \sigma)b$ be a method update term given by VAL UPDATE, and $\lceil t \rceil \rightsquigarrow v$ for some value v. Such a term has the following derivation tree:

$$\dfrac{\dfrac{\lceil m \rceil \rightsquigarrow in\, v}{out\lceil m \rceil \rightsquigarrow v \equiv \langle \ldots, \lambda x.b, \ldots \rangle}}{\begin{array}{c} in\langle \pi_1 out\lceil m \rceil, \ldots, \pi_{j-1} out\lceil m \rceil, \\ \lambda x : \lceil \sigma \rceil.\lceil b \rceil\{\sigma\}, \pi_{j+1} out\lceil m \rceil, \ldots, \pi_n out\lceil m \rceil \rangle \rightsquigarrow v \end{array}}$$

The derivation tree clearly shows that $\lceil t \rceil$ reduces to a value exactly when $\lceil m \rceil$ reduces to a value which means, by induction hypothesis, that we have $m \rightsquigarrow u$ for some value u. In other words, the premise of the VAL UPDATE rule is satisfied, so we have indeed that $t \rightsquigarrow u'$ for some value u'. It remains to be shown that $\lceil u' \rceil = v$. However, v has the form indicated in the derivation tree ($\langle \ldots, \lambda x.b, \ldots \rangle$), which is given as the interpretation of precisely the value $Obj(X = \sigma)[\ell_i = \varsigma(x : X)b_i, l_j = \varsigma(x : X)b]^i \in I$ to which t reduces to by the ($\Leftarrow$) rule. □

The following main result has now been established (it is a direct consequence of the previous theorems):

Corollary 5.2 (Main result)**.** *Every computationally sound and adequate model of lazy FPC is also such a model of* $\mathbf{S}^-$.

It follows from the proof given by Winskel [26] that the categories **pCPO** and $\mathbf{CPPO}_{\perp!}$ both give computationally sound and adequate models of $\mathbf{S}^{-}$.

Although adequacy holds, the stronger property of full abstraction does *not* hold for self-application semantics [25]. In order to discuss full abstraction we first need to define a notion of observation equivalence. Following Morris [17], terms are taken to be equivalent if they are equal, in a suitable sense, in all program contexts of a given kind. This can be understood as determining whether two terms behave in the same way, operationally. A notion of contextual equivalence requires (1) another equivalence relation to be chosen and (2) a class of program contexts to be chosen. For typed languages, we can take program contexts of one or more ground type (when such exist). For example, Plotkin [21] used boolean-valued contexts in his pioneering work on PCF. In our case, the following definition can be used instead, where we take Bool = 1 + 1:

Definition 5.3 (Contextual Equivalence [12, 13])**.** *Say that two closed terms o and o' are* contextually equivalent *(or* operationally congruent*, written $o \cong o'$), if for each closing one-hole contexts $C[-]$ of type* Bool*, we have that $C[o] \rightsquigarrow v$ if and only if $C[o'] \rightsquigarrow v'$. (A context is closing for a term t if $\vdash C[t] : \mathrm{Bool}$.)*

Note that here, we follow Gordon et al and have taken the equivalence (1) to distinguish between terms merely based on their convergence behaviour, and not on whether they reduce to the same values (unlike the equivalence Plotkin studied for PCF).

Gordon et al [12] have demonstrated that this notion of contextual equivalence can be characterised using bisimulations on a canonical labelled transition system (i.e. bisimilarity). The terminology "observational congruence" is justified since Gordon et al proved (using a technique due to Howe [15]) that the equivalence is in fact a congruence relation. Note that this means that coinduction can be used when reasoning with object-based programs, giving an alternative to using a denotational semantics (i.e. to the approach followed in this paper), although without additional structure other than that afforded by a labelled transition system (e.g. fixpoint theorems, Freyd's recursion scheme [10] are not available). At any rate, contextual equivalence makes it possible to consider how closely the operational semantics is connected to a denotational model. Ideally, one would like that programs that behaves the same are exactly those that are denotationally the same, which the following property formalises:

Definition 5.4 (Full Abstraction)**.** *Full abstraction is the property that for any terms o, o', $[\![\ulcorner \Theta, \Gamma \vdash o : \tau \urcorner]\!] = [\![\ulcorner \Theta, \Gamma \vdash o' : \tau \urcorner]\!]$ if and only if $o \cong o'$, i.e. identified denotations correspond to observationally congruent terms.*

Viswanathan showed that self-application models (such as studied in this paper) cannot have this property [25]. His counterexample is based on defining two terms:

$$\begin{array}{rcl} a & = & \mathrm{Obj}(X = \sigma)[\ell = \varsigma(x : X)x.\ell] \\ b & = & \mathrm{Obj}(X = \sigma)[\ell = \varsigma(x : X)case(x.\ell, y.\iota_1 \star, y.\iota_2 \star)] \end{array}$$

Note that both these terms are typeable with type $\mathrm{Obj}(X)[\ell : 1 + 1]$ (where $1 + 1$ represents a boolean type). Although, $a \cong b$, we do not have equal denotations in any model of lazy FPC through our interpretation, since self-application admits application of an object where the l method converges, which gives different function values (see loc. cit.). In retrospect, this is not surprising since there is no way in typed object calculus to observe for a particular method the application of that method to an object where the same method has been updated. Fortunately, the most important direction of the bi-implication is generally regarded to be that denotational equality is included in operational equality, and this is exactly what we have established in the present paper for certain kinds of models.

6 Conclusion and Further Work

In summary, we have developed an interpretation of $\mathbf{S}^-$, a typed object calculus extended with functions, coproducts and products, into FPC, and proved computational adequacy and soundness. We considered both eager and lazy variations of FPC, but we have demonstrated that lazy FPC is required for this result to hold immediately. We have established that models of lazy FPC, such as Winskel's computationally adequate denotational semantics [26], are computationally adequate also for typed object calculus. Since a direct proof of computational adequacy is rather complicated in the presence of recursive object types (compare to the work by Fiore and Plotkin [7]), the use of an interpretation into the metalanguage turned out to simplify matters substantially, while in the end giving the same result.

As is well-known [25], full abstraction does not hold for self-application semantics, although it is more abstract than many other so-called object encodings [4]. However, from a pragmatic point of view, the simplicity of the interpretation is of greater importance than its precise characterisation of objects. The self-application interpretation into FPC that we have studied is simple but comes with a powerful recursion scheme. More precisely, models of FPC studied by Fiore and Plotkin [7, 8] possess a recursion scheme due to Freyd [10]. Using our results, this scheme and results surrounding it carry over to typed object calculus. Hence the current paper has provided a formal connection that can be explored much further.

It is known from work by Fiore and Plotkin [9, 8] that there is a class of enriched categorical model of FPC which are computationally adequate for eager FPC. Notably, these authors gave a precise axiomatisation of a category of partial maps (with order-enrichment), such that a computationally adequate model of FPC arises. Examples included **pCPO** and other **CPO**-categories of partial maps, but also functor categories of **pCPO**, etc. We leave as future work to decide whether such more complex models are needed also for typed object calculus, and if therefore an axiomatic analysis of lazy FPC is called for.

Finally, we would like to remark that subtyping has not been studied in this paper, but is certainly very important and needs to be addressed. To this end, FPC can interpret subtyping using coercion functions, but the details are saved for future work. Here, the work by Abadi and Fiore [2] gives some ideas on how to proceed with such investigations.

Acknowledgments

This work was developed under the stimulating guidance of Neil Ghani, which the author gratefully acknowledges.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi and Marcelo Fiore. Syntactic considerations on recursive types. In *Proceedings 11th Annual IEEE Symp. on Logic in Computer Science, LICS'96, New Brunswick, NJ, USA, 27–30 July 1996*, pages 242–252. IEEE Computer Society Press, 1996.

[3] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[5] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, pages 51–67. Springer-Verlag, 1984. Lecture Notes in Computer Science, volume 173.

[6] Marcelo Fiore. Order-enrichment for categories of partial maps. *Mathematical Structures in Computer Science*, 5:533–562, 1995.

[7] Marcelo Fiore and Gordon Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994*, pages 92–102. IEEE Computer Society Press, July 1994.

[8] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps. Cambridge University Press, Distinguished Dissertations in Computer Science, 1996. Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[9] Marcelo P. Fiore and Gordon D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In *Proceedings of the Computer Science Logic Conf. (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 129–149. Springer-Verlag, 1997.

[10] Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90*, pages 498–507. IEEE Computer Society Press, June 1990.

[11] Johan Glimming and Neil Ghani. Difunctorial semantics of object calculus. In *Electronic Notes in Theoretical Computer Science*, volume 138. Elsevier, November 2005. Proceedings of the Second Workshop on Object Oriented Developments (WOOD 2004).

[12] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 386–395. ACM Press, 1996.

[13] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. Technical Report 386, Computer Laboratory, University of Cambridge, January 1996.

[14] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[15] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[16] Samuel N. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

[17] James Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[18] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[19] G. D. Plotkin. A metalanguage for predomains. In *Workshop on the Semantics of Programming Languages*, pages 93–118. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1983.

[20] G. D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.

[21] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[22] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.

[23] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. *Theorertical Computer Science*, 316(1):191–213, 2004.

[24] Morten H. Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149. Elsevier, 2006.

[25] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS), 1998)*. IEEE Computer Society, 1998.

[26] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

Chapter 5

Paper III: Primitive Direcursion vs. Parametric (Co)Iteration

Parametric (Co)Iteration vs. Primitive Direcursion*

Johan Glimming
Department of Numerical Analysis and Computer Science
Stockholm University
Sweden
Email: glimming@kth.se

May 14, 2007

Abstract

Freyd showed that in certain CPO-categories, locally continuous functors have minimal invariants, which possess a structure that he termed dialgebra. This gives rise to a category of dialgebras and homomorphisms, where the minimal invariants are initial, inducing a powerful recursion scheme (direcursion) on a cpo. In this paper, we identify a problem appearing when translating (co)iterative functions (on a fixed parameterised datatype) to direcursion (on the same datatype), and present a solution to this problem as a recursion scheme (primitive direcursion), generalising and symmetrising primitive (co)recursion for endofunctors. To this end, we give a uniform technique for translating (co)iterative maps into direcursive maps. This immediately gives a plethora of examples of direcursive functions, improving on the situation in the literature where only a few examples have appeared. Moreover, a technical trick proposed in a POPL paper is avoided for the translated maps. We conclude the paper by applying the results to a denotational semantics of Abadi and Cardelli's typed object calculus, and linking them to previous work on higher-order coalgebra and to bisimulations.

*This is a version of a paper that appeared in *T. Mossakowski, U. Montanari, M. Haveraaen (editors), Algebra and Coalgebra in Computer Science, CALCO 2007, Bergen, Norway, August 20-24, 2007, Proceedings. Lecture Notes in Computer Science, 4624, Springer-Verlag, 2007*.

1 Introduction

Solutions to recursive domain equations involving function spaces can be given as initial $\hat{G}$-algebras in suitable categories, where $\hat{G}$ is an endofunctor given by symmetrising G. This was shown by Freyd [11, 12] (based on work by Smyth and Plotkin [29]) and later refined by Fiore [10] in a framework of enriched category theory. Initial $\hat{G}$-algebras (also called *dialgebras*) generalise usual algebras and coalgebras. Moreover, a recursion principle arises for $\hat{G}$-algebras. This principle is hereafter called *direcursion* and it is the topic of this paper. It has previously been investigated both in loc. cit. and as a tool for functional programming (with associated proof principle) (see e.g. [20, 35, 8]). Some notable theoretical results are: the reduction to inductive types as given by Freyd in his seminal paper [11] and the relationship to dinaturality [12], the derivation of an associated proof principle [22, 23], programming examples dealing with higher-order abstract syntax [35], lambda calculus interpreters [20], and circular datatypes [8]. But direcursion remains relatively unexplored as regards termination properties and its relationships to other recursion schemes (and programming examples have so far been rather scarce). In this paper we begin to remedy this situation.

We will here investigate the relationships between (co)iteration and direcursion for a *fixed datatype*. With (co)iteration we mean the unique homomorphisms associated to the initial (final) $\hat{G}(\mu\hat{G}, \_)$-(co)algebras by Bekič's Lemma, i.e. (co)iterative maps on this particular parameterised datatype. Since the carrier of this (co)algebra coincides with the solution $\mathcal{O} = \mu\hat{G}$, we ask how these schemes compare to direcursion for same functor G. Our main result is to show that by generalising direcursion (by precomposing with an injection map or postcomposing with a projection map), we can express all such (co)iterative maps as a canonical direcursive map such that the same computation is carried out at every stage. We call this generalisation *primitive direcursion* since it is primitive recursion for symmetric functors $\hat{G}$, i.e. it is simultaneously primitive recursion and primitive corecursion for recursive types. The latter two principles have been studied in previous work by Meertens [18] and Uustalu and Vene [32]. Primitive direcursion simultaneously gives both schemes in the special case when the bifunctor is constant in its contravariant argument, but also transports those schemes to cover domain equations involving function spaces as well, i.e. to the mixed variant case, for which it has not previously been considered.

The paper is structured as follows: the first few pages survey necessary background material. Next, we develop primitive direcursion from first principles. In the same section, we give the translation of certain iterative/coiterative maps into

primitive direcursive maps, which is our main result. This result can be viewed as an internalised version of Bekič's Lemma. The fourth section exemplifies the results in the setting of program semantics, and the last section gives our conclusions.

2 Mathematical Preliminaries

In this paper we are essentially considering the category $\mathsf{CPPO}_{\perp!}$ of directed complete pointed cpos and *strict* continuous maps (or subcategories with similar properties, e.g. Scott domains and strict maps), as defined in e.g. [3, 30]. Such a category is used when solving domain equations, for instance in [29] based on [28]. We will in this paper assume an ambient category $\mathcal{C}$ which abstracts from $\mathsf{CPPO}_{\perp!}$ exactly the properties that we require here. These are our assumptions on $\mathcal{C}$:

- $\mathcal{C}$ has products $\times$ and coproducts $+$.
- $\mathcal{C}$ is algebraically compact, so that each (suitably qualified) endofunctor has an initial algebra and a final coalgebra, and these are canonically isomorphic [12], i.e. the unique homomorphism from the inverse of the initial algebra to the final coalgebra is an isomorphism. In particular, this family of endofunctors is assumed to include the functors considered in this paper. It follows also that $\mathcal{C}$ has a zero object $0 \cong 1$, also known as a biterminator.
- $\mathcal{C}$ has regular initial dialgebras [11], as will be detailed below. Having regular free dialgebras is in fact a consequence of algebraic compactness [12]. In a weaker axiomatisation where we require merely free dialgebras for a class of functors instead of algebraic compactness, this condition must however remain explicit, see [11, 13] for examples.
- $\mathcal{C}$ is symmetric monoidal closed with tensor $\otimes$ and unit 1. The right adjoint to this tensor is written $\multimap$ (with the natural isomorphisms curry and $\mathsf{uncurry}$, and counit eval).
- $\mathcal{C}$ has a generator I, i.e. for all $f, g \in C(A, B)$ we have $f = g$ iff for all $i : I \to A$ we have that $f \circ i = g \circ i$.

An example of such a category is $\mathsf{CPPO}_{\perp!}$ itself, in which case the tensor $\otimes$ is smash product (with right adjoint strict function space, with curry etc). The product

$\times$, on the other hand, is the cartesian product of cpos (thus we have merely *weak exponentials* in the sense of $\otimes$ being left adjoint to the function space rather than the product), and coproduct is coalesced sum (i.e. the least elements are identified in contrast to e.g. separated sum where a new one is adjoined). The generator for this category is given by the Sierpinski space $I = \{\bot, \top\}$ (so it is in particular not well pointed since $I \not\cong 1$). In $\mathsf{CPPO}_{\bot!}$ the family of endofunctors considered above are the $\mathsf{CPPO}_{\bot!}$-enriched (i.e. locally continuous) functors, i.e. functors $F : \mathcal{C} \to \mathcal{C}$ given by maps $|\mathcal{C}| \to |\mathcal{C}|$ on objects (writing $|\mathcal{C}|$ for the class of objects), and arrow maps given on homsets by a *Scott-continuous* mapping $\mathcal{C}(A, B) \mapsto \mathcal{C}(FA, FB)$ for each $A, B \in |\mathcal{C}|$, such that composition and identities are preserved. Such functors are alternatively called *locally continuous*. Note that for bifunctors this means that each section is locally continuous. In particular, for mixed variant functors $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ we require that $\mathcal{C}(B, A) \times \mathcal{C}(A', B') \mapsto \mathcal{C}(F(A, A'), F(B, B'))$ has said property instead (see [3]).

Definition 2.1 (Algebra, Coalgebra)**.** *Given a covariant functor $F : \mathcal{C} \to \mathcal{C}$ we say that an arrow $\alpha : F(A) \to A$ is an F-algebra with carrier A. The dual notion is that of F-coalgebra, i.e. reversed arrows $\alpha : A \to F(A)$. The arrows between (co)algebras are F-homomorphisms, i.e. arrows h such that the left or right diagram below commute (in the respective case):*

$$\begin{array}{ccc} F(A) & \xrightarrow{F(h)} & F(B) \\ {\scriptstyle\alpha}\downarrow & \mathbf{Alg}(F) & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[h]{} & B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{h} & B \\ {\scriptstyle\alpha}\downarrow & \mathbf{Coalg}(F) & \downarrow{\scriptstyle\beta} \\ F(A) & \xrightarrow[F(h)]{} & F(B) \end{array}$$

We now recall some results concerning solutions to recursive domain equations in $\mathsf{CPPO}_{\bot!}$. These results serve to motivate our axiomatised category $\mathcal{C}$. Given a locally continuous endofunctor F, we construct a diagram by iterating the functor, beginning at the zero object 0. There are then unique morphisms $0 \to F0$ and $F0 \to 0$ so we can construct systems giving both a limit and a colimit. The limit and colimit coincide in this case, and is denoted μF. This motivates the algebraic compactness requirement for $\mathcal{C}$. Further details are given in e.g. [29], [24], [3]. One particular result (not detailed here) is that μF carries an initial algebra and also a final coalgebra (arising from considering respectively cones and cocones, see loc. cit.). This is an important consequence since it together with the following lemma shows that in $\mathsf{CPPO}_{\bot!}$ we can solve domain equations (for locally continuous endofunctors) up to isomorphism:

Proposition 2.1 (Lambek's Lemma)**.** *An initial algebra* $(\mathcal{O}_F, \iota_F)$ *is an isomorphism* $\mathcal{O}_F \cong F(\mathcal{O}_F)$*. Dually for final coalgebra* $(\mathcal{O}_F, \iota_F^\circ)$

The notation ι_F and ι_F° is used for the initial algebra and final coalgebra respectively (we drop suffixes when possible). Our next assumption for $\mathcal{C}$ is that it has regular initial dialgebras. The fact that $\mathsf{CPPO}_{\perp!}$ satisfies this condition is due to Freyd [11] (but see also pioneering work by [29] in the more concrete setting of a subcategory of embedding-projection pairs). We survey Freyd's work here, in particular by recalling that, for a (mixed-variant) functor F, an object X is called F-invariant if there is an isomorphism $\alpha : F(X) \cong X$. If *fix* $e.\ \alpha \circ F(e) \circ \alpha^{-1} \in A \to A$ is the identity, X is called special F-invariant. If it is the only idempotent map $A \to A$ for which $e \circ \alpha = \alpha \circ F(e)$, it is called *minimal invariant* [11]. Freyd [11] showed that in $\mathsf{CPPO}_{\perp!}$ there exists an F-invariant object for every locally continuous functor that is minimal in this sense. A corollary of this result is the recursion principle that we call direcursion, which relies on first generalising the notion of (co)algebra to mixed-variance functors:

Definition 2.2 (Dialgebra)**.** *A* G-dialgebra *for bifunctor* $G : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ *is a quadruple* (A, B, ϕ, ψ) *of objects* A, B *and associated arrows* $\phi : G(B, A) \to A$ *and* $\psi : B \to G(A, B)$*.*

Note that in the case when G is dummy in its contravariant argument, i.e. an endofunctor F on $\mathcal{C}$, this definition gives precisely that (A, ϕ) is an F-algebra and, independently, that (B, ψ) is a F-coalgebra. Dialgebras for a bifunctor G form a category $Dialg(G)$ with the following morphisms:

Definition 2.3 (Dialgebra Map)**.** *Given* G*-dialgebras* (A, B, ϕ, ψ) *and* (A', B', ϕ', ψ')*, a* G-homomorphism *(or* dialgebra map/dimap*) is a pair of arrows* $(h : A \to A', g : B' \to B)$ *such that the following diagrams commute:*

$$\begin{array}{ccc} G(B,A) & \xrightarrow{\phi} & A \\ {\scriptstyle G(g,h)}\downarrow & \equiv & \downarrow{\scriptstyle h} \\ G(B',A') & \xrightarrow[\phi']{} & A' \end{array} \qquad \begin{array}{ccc} B & \xrightarrow{\psi} & G(A,B) \\ {\scriptstyle g}\uparrow & \equiv & \uparrow{\scriptstyle G(h,g)} \\ B' & \xrightarrow[\psi']{} & G(A',B') \end{array}$$

An initial dialgebra for a bifunctor G is a dialgebra (A, B, ϕ, ψ) such that for any other G-dialgebra (A', B', ϕ', ψ') there is a unique dialgebra map $(h : A \to A', g : B' \to B)$. The existence of initial dialgebras in $\mathsf{CPPO}_{\perp!}$ was established by Freyd:

Theorem 2.1 (Existence of Initial Dialgebras [11])**.** $\mathsf{CPPO}_{\perp!}$ *has initial dialgebras for every locally continuous bifunctor* $G : \mathsf{CPPO}_{\perp!}{}^{op} \times \mathsf{CPPO}_{\perp!} \to \mathsf{CPPO}_{\perp!}$. *In addition, initial dialgebras in* $\mathsf{CPPO}_{\perp!}$ *are of the form* $(\mathcal{O}_G, \mathcal{O}_G, \iota_G, \iota_G^\circ)$ *where* $\iota_G \circ \iota_G^\circ = id$ *and* $\iota_G^\circ \circ \iota_G = id$.

Proof. See e.g. [23]. □

The second property in the theorem is what Freyd termed *regular initial dialgebra*. The existence of such dialgebras was one of the conditions we listed for $\mathcal{C}$, and it will be frequently used in this paper. Note also that usual initial algebras and final coalgebras for endofunctors follows from Freyd's condition since the difunctor can be constant in its negative argument, e.g. $G(Y, X) = 1 + X$. (In such cases the intertwined diagrams for direcursion instead become two independent diagrams for iteration and coiteration, respectively.) Moreover, Freyd's condition has the following consequence, which is the recursion principle studied in this paper:

Definition 2.4 (Direcursion [11])**.** *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial* G*-dialgebra and suppose* (A, B, ϕ, ψ) *is some other* G*-dialgebra. Then there exists unique morphisms* $g : \mathcal{O} \longrightarrow A$ *and* $h : B \longrightarrow \mathcal{O}$ *such that the following diagrams commute:*

$$\begin{array}{ccc} G(\mathcal{O},\mathcal{O}) & \xrightarrow{\iota_G} & \mathcal{O} \\ {\scriptstyle G(h,g)}\downarrow & \equiv & \downarrow{\scriptstyle g} \\ G(B,A) & \xrightarrow[\phi]{} & A \end{array} \qquad \begin{array}{ccc} \mathcal{O} & \xrightarrow{\iota_G^\circ} & G(\mathcal{O},\mathcal{O}) \\ {\scriptstyle h}\uparrow & \equiv & \uparrow{\scriptstyle G(g,h)} \\ B & \xrightarrow[\psi]{} & G(A,B) \end{array} \qquad \text{(direc-PROP)}$$

We introduce the notation $(\!|\phi, \psi|\!)_G \stackrel{def}{=} g$ *and* $[\!(\phi, \psi)\!]_G \stackrel{def}{=} h$ *whenever the conditions for* ϕ *and* ψ *are satisfied.*

There are a number of standard properties that can be easily established. We survey them here:

Lemma 2.1 (Basic properties of direcursion [20])**.** *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial* G*-dialgebra and suppose* (A, B, ϕ, ψ) *is some other* G*-dialgebra. Then the follow-*

ing is true:

$$(\!|\phi,\psi|\!) \circ \iota_G = \phi \circ G([\![\phi,\psi]\!], (\!|\phi,\psi|\!)) \qquad \text{(direc-Self)}$$
$$\iota_G^\circ \circ [\![\phi,\psi]\!] = G((\!|\phi,\psi|\!), [\![\phi,\psi]\!]) \circ \psi$$

$$id = (\!|\iota_G, \iota_G^\circ|\!) \qquad \text{(direc-Refl)}$$
$$id = [\![\iota_G, \iota_G^\circ]\!]$$

$$A = B \text{ and } \psi \circ \phi = id_{G(A,A)} \text{ implies } [\![\phi,\psi]\!] \circ (\!|\phi,\psi|\!) = id_{\mathcal{O}} \qquad \text{(direc-Retract)}$$

Proof. The two first properties follow directly. The third one follows by pasting the right square for direcursion below the left one, and vice versa. □

Lemma 2.2 (Direcursion Fusion [20])**.** *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial G-dialgebra. Suppose that* $(A, B, \phi : G(B,A) \to A, \psi : B \to G(A,B))$ *and* $(A', B', \phi' : G(B',A') \to A', \psi' : B' \to G(A',B'))$ *are both G-dialgebras. For every dimap* $g : A \to A', h : B' \to B$ *such that*

$$g \circ \phi = \phi' \circ G(h, g)$$

and

$$\psi \circ h = G(g,h) \circ \psi'$$

we have the following property

$$g \circ (\!|\phi,\psi|\!) = (\!|\phi',\psi'|\!) \text{ and } [\![\phi,\psi]\!] \circ h = [\![\phi',\psi']\!] \qquad \text{(direc-Fusion)}$$

Symmetric Functors, Bekič's Lemma and Parameterised (Co)Algebras

As noted by Freyd [11] (and further detailed in [9]), we can by introducing symmetric endofunctors $\mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}^{op} \times \mathcal{C}$, view a dialgebra as an algebra on the product category. We have in particular the following result:

Lemma 2.3 ([11])**.** *There is a bijective correspondence between functors* $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ *and symmetric functors* $\hat{F} : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}^{op} \times \mathcal{C}$*, where*

$$\begin{aligned} \hat{F}(X,Y) &= (F(Y,X), F(X,Y)) \\ \hat{F}(f,g) &= (F(g,f), F(f,g)) \end{aligned}$$

Proof. This is established e.g. using the notion of involutions (self-dual functors) and a category $\hat{\mathcal{C}}$ of involutory objects (due to John Power, see Fiore [9] for details). □

In particular, this implies that direcursion can alternatively be formulated using $\hat{F}$, in which case the maps $f = (\!|\phi, \psi|\!)$ and $g = [\![\phi, \psi]\!]$ are equivalently and more abstractly defined as follows in the category $\mathcal{C}^{op} \times \mathcal{C}$ (see [9]):

$$\begin{array}{ccc} \hat{G}(\mathcal{O}, \mathcal{O}) & \xrightarrow[(\iota_G, \iota_G^\circ)]{} & (\mathcal{O}, \mathcal{O}) \\ \Big\downarrow{\scriptstyle \hat{G}(f,g)} & \equiv & \Big\downarrow{\scriptstyle (f,g)} \\ \hat{G}(A, B) & \xrightarrow[(\phi, \psi)]{} & (A, B) \end{array}$$

Given a bifunctor $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ and an object $X \in |\mathcal{C}|$ we have an endofunctor $F(X, \_) : \mathcal{C} \to \mathcal{C}$. This means that we can consider two different equation systems as follows (where we already have shown that the initial F-dialgebra gives a solution for the left-hand side (lhs) system).

$$\begin{cases} X \cong F(Y, X) = \pi_1(\hat{F}(X, Y)) \\ Y \cong F(X, Y) = \pi_2(\hat{F}(X, Y)) \end{cases} \qquad \begin{cases} X \cong F(\mu F(X, \_), X) \\ Y \cong \mu F(X, \_) \end{cases}$$

The solution to the rhs system is said to be parameterised, and it is not immediately clear if it has the same solutions as the lhs one. However, we have:

Lemma 2.4 (Bekič's Lemma [4])**.** *Let $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ be a locally continuous endofunctor with initial dialgebra $(\mathcal{O}, \mathcal{O}, \iota, \iota^\circ)$. Then we have that $\mathcal{O} \cong \mu F(\mathcal{O}, \_)$.*

Proof. E.g. [3, 9]. □

The endofunctors of the form $F(A, \_)$ in this way have initial $F(A, \_)$-algebras where A is the parameter.

Primitive (Co)Recursion and (Co)Iteration

It is well known since Lawvere, that iteration and primitive recursion (on natural numbers) can be expressed in the form of universal properties (e.g. a so-called natural number object, nno) in cartesian closed categories. Subsequent research on functional programming has built on this idea, and considered other datatypes.

A datatype within functional programming is typically modelled by an initial algebra for suitable functor F, and this work has treated also the functor as a parameter of programs. Since each functor F on $\mathcal{C}$ has an initial algebra $\mathcal{O}$, there is a unique homomorphism into $\mathcal{O}$ from any other object A for which there is a structure $\phi : FA \to A$. We call these *F-iterative maps*, written $(\!|\phi|\!)_F$ for easy reference. The dually constructed maps are called *F-coiterative maps*, and are written $[\!(\psi)\!]_F$. From these, Meertens [18] constructed F-primitive recursion:

Theorem 2.2 (F-primitive recursion)**.** *Suppose $(\mu F, \iota_F)$ is the initial F-algebra. For every morphism $\phi : F(\mu F \times A) \to A$ there exists a unique morphism h (called* F-primitive recursion*) such that the following diagram commutes:*

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\iota_F} & \mu F \\ {\scriptstyle \langle id, h\rangle}\downarrow & \equiv & \downarrow{\scriptstyle h} \\ F(\mu F \times A) & \xrightarrow[\phi]{} & A \end{array}$$

Proof. For existence define a map $\phi' : F(\mu F \times A) \to \mu F \times A$ by $\phi' = \langle \iota \circ F(\pi_1), \phi\rangle$. Hence there is a coiterative map $(\!|\phi'|\!)_F$ which satisfies the diagram when post-composed with π_2. For uniqueness suppose $h = \phi \circ \langle id, h\rangle \circ \iota_F^\circ$. But then $\langle \pi_1 \circ (\!|\phi'|\!)_F, h\rangle$ is a homomorphism into $\mu F \times A$, and hence, by properties of pairing in $\mathcal{C}$ together with initiality of $(\mu F, \iota_F)$, we have $h = \pi_2 \circ (\!|\phi'|\!)_F$. □

This result dualises into a scheme useful for coalgebraic datatypes, as was shown by Uustalu et al [32, 33]. The definition of such *F-primitive corecursion* is dual to the above construction, and so is the associated proof.

3 Primitive Direcursion

We will in this section consider a symmetrised version of primitive (co)recursion, and prove a number of basic results for this recursion principle, followed by our main result. We provide a quite detailed exposition here, and develop primitive direcursion from first principles. Many proofs (but not our main result) can alternatively be viewed as a special case of primitive recursion with the ambient category $\mathcal{C}^{op} \times \mathcal{C}$, i.e. $\hat{F}$-primitive recursion, and are therefore omitted. Note that direcursion itself by a similar argument is a special case of iteration, if one moves

to the symmetrised category $\mathcal{C}^{op} \times \mathcal{C}$, as we mentioned in a previous section. However, we include some such proofs to highlight which properties of $\mathcal{C}$ they rely on (including the regularity assumption).

Theorem 3.1 (Primitive Direcursion)**.** *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial G-dialgebra. Let A and B be two objects and* $\phi : G(\mathcal{O}+B, O\times A) \to A$ *and* $\psi : B \to G(\mathcal{O}\times A, \mathcal{O}+B)$ *two morphisms. Then there exist* $g : \mathcal{O} \longrightarrow A$ *and* $h : B \longrightarrow \mathcal{O}$ *such that the following diagrams commute:*

$$\begin{array}{ccc} G(\mathcal{O},\mathcal{O}) & \xrightarrow{\iota_G} & \mathcal{O} \\ {\scriptstyle G([id,h],\langle id,g\rangle)}\downarrow & \equiv & \downarrow{\scriptstyle g} \\ G(\mathcal{O}+B, \mathcal{O}\times A) & \xrightarrow[\phi]{} & A \end{array} \qquad \begin{array}{ccc} \mathcal{O} & \xrightarrow{\iota_G^\circ} & G(\mathcal{O},\mathcal{O}) \\ {\scriptstyle h}\uparrow & \equiv & \uparrow{\scriptstyle G(\langle id,g\rangle,[id,h])} \\ B & \xrightarrow[\psi]{} & G(\mathcal{O}\times A, \mathcal{O}+B) \end{array}$$

(prim-Prop)

Proof. Let ϕ and ψ be given as in the antecedent of the theorem. We instantiate direcursion with $A' = \mathcal{O}\times A$ and $B' = \mathcal{O}+B$ and define $\phi' : G(\mathcal{O}+B, \mathcal{O}\times A) \to \mathcal{O}\times A$ and $\psi' : \mathcal{O} + B \to G(\mathcal{O} \times A, \mathcal{O} + B)$ by $\phi' = \langle \iota \circ G(\mathsf{inl}, \pi_1), \phi\rangle$ and $\psi' = [G(\pi_1, \mathsf{inl}) \circ \iota^\circ, \psi]$. From these two maps, we define $g = \pi_2 \circ (\!|\phi', \psi'|\!)$ and $h = [\!(\phi', \psi')\!] \circ \mathsf{inr}$. Finally, we verify that each square commutes (omitting the reasoning for the right square as the following dualises):

$$\begin{array}{lll} & g \circ \iota & \\ = & \pi_2 \circ (\!|\phi', \psi'|\!) \circ \iota & \text{by assumption} \\ = & \pi_2 \circ \langle \iota \circ G(\mathsf{inl}, \pi_1), \phi\rangle \circ G([\!(\phi', \psi')\!], (\!|\phi', \psi'|\!)) & \text{by (direc-Self)} \\ = & \phi \circ G([\!(\phi', \psi')\!], (\!|\phi', \psi'|\!)) & \text{by (co)pairing} \\ = & \phi \circ G([[\!(\phi', \psi')\!] \circ \mathsf{inl}, [\!(\phi', \psi')\!] \circ \mathsf{inr}], \langle \pi_1 \circ (\!|\phi', \psi'|\!), & \\ & \pi_2 \circ (\!|\phi', \psi'|\!)\rangle) & \text{by surjective pairing} \\ = & \phi \circ G([[\!(\iota, \iota^\circ)\!], h], \langle (\!|\iota, \iota^\circ|\!), g\rangle) & \text{by (direc-Fusion)} \\ = & \phi \circ G([id, h], \langle id, g\rangle) & \text{by (direc-Refl)} \end{array}$$

□

The previous theorem in fact defines a unique pair of morphisms:

Theorem 3.2 (Primitive Direcursion Characterisation)**.** *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial G-dialgebra. Suppose*

$$\begin{array}{llll} (a) & \phi' & = & \langle \iota \circ G(inl, \pi_1), \phi\rangle, \textit{ and} \\ (b) & \psi' & = & [G(\pi_1, inl) \circ \iota^\circ, \psi]. \end{array}$$

Then the following two statements are equivalent:

$$\begin{aligned} &(i)\ g \circ \iota_G = \phi \circ G([id, h], \langle id, g\rangle) \\ &\quad\ \iota_G^\circ \circ h = G(\langle id, g\rangle, [id, h]) \circ \psi, \text{ and} \\ &(ii)\ g = \pi_2 \circ (\!|\phi', \psi'|\!) \text{ and } h = [\!(\phi', \psi')\!] \circ \mathit{inr}. \end{aligned} \qquad \text{(prim-CHARN)}$$

We introduce the notation $\langle\!|\phi, \psi|\!\rangle_G \stackrel{def}{=} g$ *and* $|\!\rangle\phi, \psi|\!\langle_G \stackrel{def}{=} h$ *for any given pair of (well-typed) maps* ϕ *and* ψ.

Proof. Supposing (*i*) holds, we establish (ii).

$$\begin{aligned} & g \\ =\ & \pi_2 \circ \langle id, g\rangle && \text{by pairing} \\ =\ & \pi_2 \circ (\!|\langle \iota \circ G(\pi_1, \mathsf{inl}), \phi\rangle, [G(\mathsf{inl}, \pi_1) \circ \iota^\circ, \psi]|\!) && \text{by } (\star) \text{ below} \end{aligned}$$

We now complete the proof by spelling out the step ($\star$), thus giving the first half of the characterisation:

$$\begin{aligned} & \langle id, g\rangle \circ \iota \\ =\ & \langle id \circ \iota, g \circ \iota\rangle && \text{by pairing} \\ =\ & \langle \iota \circ G(id, id), g \circ \iota\rangle && \text{by functor law} \\ =\ & \langle \iota \circ G([id, h] \circ \mathsf{inl}, \pi_1 \circ \langle id, g\rangle), g \circ \iota\rangle && \text{by pairing} \\ =\ & \langle \iota \circ G(\mathsf{inl}, \pi_1) \circ G([id, h], \langle id, g\rangle), \phi \circ G([id, h], \langle id, g\rangle)\rangle && \text{by assumption (i)} \\ =\ & \langle \iota \circ G(\mathsf{inl}, \pi_1), \phi\rangle \circ G([id, h]\langle id, g\rangle) && \text{by pairing} \end{aligned}$$

The remaining case is dual. Using prim-PROP we are done. □

The following basic property follows directly:

Corollary 3.1. *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial G-dialgebra. Every function given by G-primitive direcursion can also be given by G-direcursion up to a certain pre/post-composed map:*

$$\begin{aligned} \langle\!|\phi, \psi|\!\rangle &= \pi_2 \circ (\!|\langle \iota \circ G(\mathit{inl}, \pi_1), \phi\rangle, [G(\pi_1, \mathit{inl}) \circ \iota^\circ, \psi]|\!) \\ |\!\rangle\phi, \psi|\!\langle &= [\!(\langle \iota \circ G(\mathit{inl}, \pi_1), \phi\rangle, [G(\pi_1, \mathit{inl}) \circ \iota^\circ, \psi])\!] \circ \mathit{inr} \end{aligned} \qquad \text{(prim-DIREC)}$$

Proof. Straightforward. □

Note here that primitive direcursive functions can generally not be defined by purely direcursive functions, although the previous corollary establishes a close relationship between the two notions.

Lemma 3.1. *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial G-dialgebra. Primitive direcursion satisfies the following cancellation and reflection laws:*

$$\langle\!|\phi, \psi|\!\rangle \circ \iota_G = \phi \circ G([id, |\!\rangle\phi, \psi|\!\langle], \langle id, \langle\!|\phi, \psi|\!\rangle\rangle) \qquad \text{(prim-Self)}$$

$$\iota_G^\circ \circ |\!\rangle\phi, \psi|\!\langle = G(\langle id, \langle\!|\phi, \psi|\!\rangle\rangle, [id, |\!\rangle\phi, \psi|\!\langle]) \circ \psi$$

$$\forall\psi \quad id = \langle\!|\iota \circ G(\mathsf{inl}, \pi_1), \psi|\!\rangle \qquad \text{(prim-Refl)}$$

$$\forall\phi \quad id = |\!\rangle\phi, G(\pi_1, \mathsf{inl}) \circ \iota^\circ|\!\langle$$

Proof. The cancellation law follows immediately from (prim-Charn) by just chasing the diagram given in (prim-Prop). We show the first reflection law (the other is dual):

$$\begin{aligned}
& \langle\!|\iota \circ G(\mathsf{inl}, \pi_1), \psi|\!\rangle \\
= \; & \langle\!|\iota \circ G(\mathsf{inl}, \pi_1), \psi|\!\rangle \circ \iota \circ \iota^\circ && \text{by regularity} \\
= \; & \iota \circ G(\mathsf{inl}, \pi_1) \circ G([id, |\!\rangle\phi, \psi|\!\langle], \langle id, \langle\!|\phi, \psi|\!\rangle\rangle) \circ \iota^\circ && \text{by (prim-Self)} \\
= \; & \iota \circ G([id, |\!\rangle\phi, \psi|\!\langle] \circ \mathsf{inl}, \pi_1 \circ \langle id, \langle\!|\phi, \psi|\!\rangle\rangle) \circ \iota^\circ && \text{by composition} \\
= \; & \iota \circ G(id, id) \circ \iota^\circ && \text{by (co)pairing} \\
= \; & \iota \circ \iota^\circ && \text{by functor property} \\
= \; & id && \text{by regularity}
\end{aligned}$$

□

The previous lemma, albeit a direct consequence of the construction, is of importance since it gives a more efficient implementation of primitive direcursion in a lazy functional programming language. The final basic property, the fusion law, takes a particular form for primitive direcursion:

Corollary 3.2 (Primitive Direcursion Fusion)**.** *Let* $(\mathcal{O}, \mathcal{O}, \iota_G, \iota_G^\circ)$ *be the initial G-dialgebra. Suppose that* $(A, B, \phi : G(\mathcal{O} + B, \mathcal{O} \times A) \to A, \psi : B \to G(\mathcal{O} \times A, \mathcal{O} + B))$ *and* $(A', B', \phi' : G(\mathcal{O} + B', \mathcal{O} \times A') \to A', \psi' : B' \to G(\mathcal{O} \times A', \mathcal{O} + B'))$ *are both G-dialgebras. For every dimap* $g : A \to A', h : B' \to B$ *such that*

$$g \circ \phi = \phi' \circ G(id + h, id \times g)$$

and

$$\psi \circ h = G(id \times g, id + h) \circ \psi'$$

we have the following property

$$g \circ \langle\!|\phi, \psi|\!\rangle = \langle\!|\phi', \psi'|\!\rangle \text{ and } |\!\rangle\phi, \psi|\!\langle \circ h = |\!\rangle\phi', \psi'|\!\langle \qquad \text{(prim-Fusion)}$$

We establish another relationship between direcursion and primitive direcursion, showing that primitive direcursion generalises direcursion in the following sense:

Lemma 3.2. *Let $(\mathcal{O},\mathcal{O},\iota,\iota^\circ)$ be the initial G-dialgebra. For any $\phi : G(B,A) \to A$ and $\psi : B \to G(A,B)$, the following equalities hold:*

$$\begin{array}{lcl} (\!|\phi, \psi|\!) & = & \langle\!|\phi \circ G(inr, \pi_2), G(\pi_2, inr) \circ \psi|\!\rangle \\ [\!(\phi, \psi)\!] & = & |\!\rangle\phi \circ G(inr, \pi_2), G(\pi_2, inr) \circ \psi|\!\langle \end{array} \qquad \text{(direc-Prim)}$$

That is, every direcursive function is also a primitive direcursive function.

Proof. We first define two maps $\phi' : G(\mathcal{O}+B, \mathcal{O}\times A) \to (\mathcal{O}\times A)$ and $\psi' : (\mathcal{O}+B) \to G(\mathcal{O}\times A, \mathcal{O}+B)$:

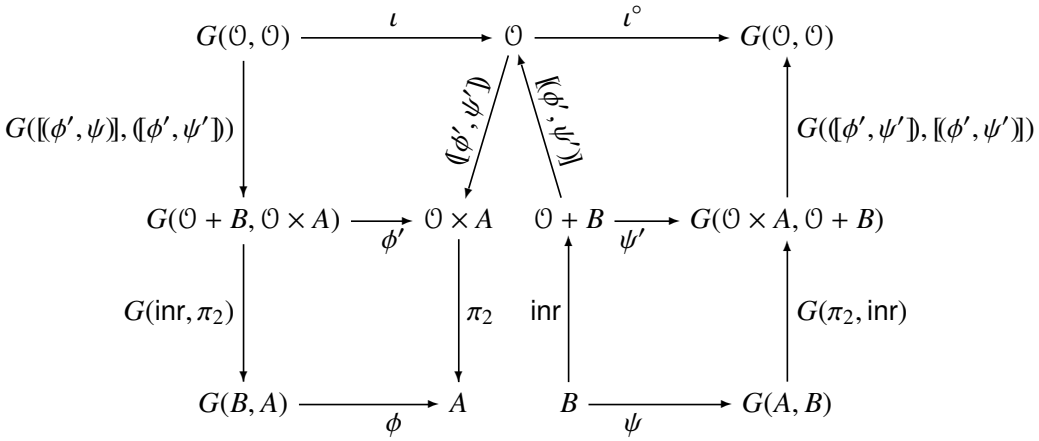
$$\begin{aligned} \phi' &= \langle \iota \circ G(\mathsf{inl}, \pi_1), \phi \circ G(\mathsf{inr}, \pi_2) \rangle \\ \psi' &= [G(\pi_1, \mathsf{inl}) \circ \iota^\circ, G(\pi_2, \mathsf{inr}) \circ \psi] \end{aligned}$$

By applying (prim-Direc) to the right hand side of (direc-Prim) and by definition of ϕ' and ψ' it is sufficient to prove the following:

$$\begin{aligned} (\!|\phi, \psi|\!) &= \pi_2 \circ (\!|\phi', \psi'|\!) \\ [\!(\phi, \psi)\!] &= [\!(\phi', \psi')\!] \circ \mathsf{inr} \end{aligned}$$

We will now use (direc-Fusion), and take a dimap (π_2, inr) as the witness for the

equality, as shown in the following diagrams:



The upper two squares commute simply by the universal property of direcursion, i.e. by (direc-Prop). The lower two squares commute (i.e. we have the dimap condition) since we have the following:

$$\pi_2 \circ \phi' \;=\; \pi_2 \circ \langle \iota \circ G(\mathsf{inl},\pi_1), \phi \circ G(\mathsf{inr},\pi_2))\rangle \;=\; \phi \circ G(\mathsf{inr},\pi_2)$$

For the second square we dually reason as follows:

$$\psi' \circ \mathsf{inr} \;=\; [G(\pi_1,\mathsf{inl}) \circ \iota^\circ, G(\pi_2,\mathsf{inr}) \circ \psi] \circ \mathsf{inr} \;=\; G(\pi_2,\mathsf{inr}) \circ \psi$$

□

The (prim-Self) law generalises into a statement that primitive direcursion is universal in the sense of e.g. Proposition 4.3 in [6] (i.e. any map in the category can be defined using the scheme):

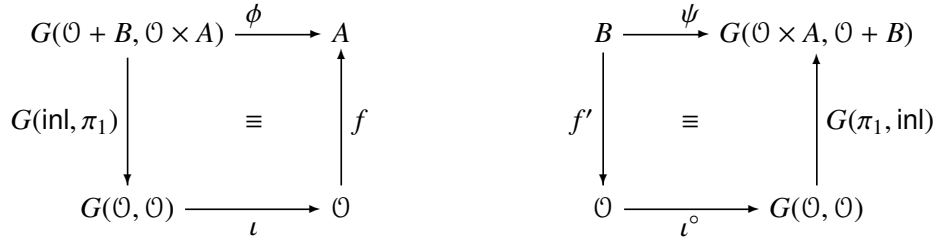
Lemma 3.3. *Let $(\mathcal{O},\mathcal{O},\iota,\iota^\circ)$ be the initial G-dialgebra. Any morphism $f : \mathcal{O} \to A$ satisfies the following identity, for any $\psi : B \to G(\mathcal{O}\times A, \mathcal{O}+B)$:*

$$f = \langle\!|f \circ \iota \circ G(\mathsf{inl},\pi_1),\psi|\!\rangle$$

Furthermore, any morphism $f' : B \to \mathcal{O}$ satisfies the following identity, for any $\phi : G(\mathcal{O}+B, A\times\mathcal{O}) \to A$:

$$f' = |\!\rangle\phi, G(\pi_1,\mathsf{inl}) \circ \iota^\circ \circ f'|\!\langle$$

Proof. We first convince the reader that the equalities are well-typed:



Let $\phi_0 = f \circ \iota \circ G(\mathsf{inl}, \pi_1)$. For the first equality we reason as follows:

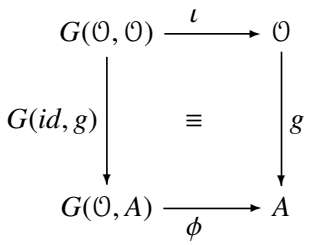
$$
\begin{aligned}
& \langle\!|f \circ \iota \circ G(\mathsf{inl}, \pi_1), \psi|\!\rangle \\
=\ & \iota \circ \iota^\circ \circ \langle\!|f \circ \iota \circ G(\mathsf{inl}, \pi_1), \psi|\!\rangle && \text{by regularity} \\
=\ & f \circ \iota \circ G(\mathsf{inl}, \pi_1) \circ G([id, |\!\rangle \phi_0, \psi \langle\!|], \langle id, \langle\!|\phi_0, \psi|\!\rangle\rangle) \circ \iota^\circ && \text{by (prim-SELF)} \\
=\ & f \circ \iota \circ G([id, |\!\rangle \phi_0, \psi \langle\!|] \circ \mathsf{inl}, \pi_1 \circ \langle id, \langle\!|\phi_0, \psi|\!\rangle\rangle) \circ \iota^\circ && \text{by composition} \\
=\ & f \circ \iota \circ G(id, id) \circ \iota^\circ && \text{by (co)pairing} \\
=\ & f && \text{by regularity}
\end{aligned}
$$

Let $\psi_0 = G(\pi_1, \mathsf{inl}) \circ \iota^\circ \circ f'$. For the second equality dually reason as follows:

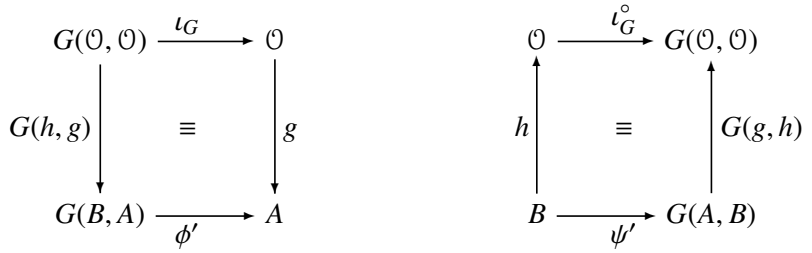
$$
\begin{aligned}
& |\!\rangle \phi, f \circ \iota \circ G(\mathsf{inl}, \pi_1) \langle\!| \\
=\ & |\!\rangle \phi, f \circ \iota \circ G(\mathsf{inl}, \pi_1) \langle\!| \circ \iota \circ \iota^\circ && \text{by regularity} \\
=\ & \iota \circ G(\langle id, \langle\!|\phi, \psi_0|\!\rangle\rangle, [id, |\!\rangle \phi, \psi_0 \langle\!|]) \circ G(\pi_1, \mathsf{inl}) \circ \iota^\circ \circ f' && \text{by (prim-SELF)} \\
=\ & \iota \circ G(\pi_1 \circ \langle id, \langle\!|\phi, \psi_0|\!\rangle\rangle, [id, |\!\rangle \phi, \psi_0 \langle\!|] \circ \mathsf{inl}) \circ \iota^\circ \circ f' && \text{by composition} \\
=\ & \iota \circ G(id, id) \circ \iota^\circ \circ f' && \text{by (co)pairing} \\
=\ & f' && \text{by regularity}
\end{aligned}
$$

□

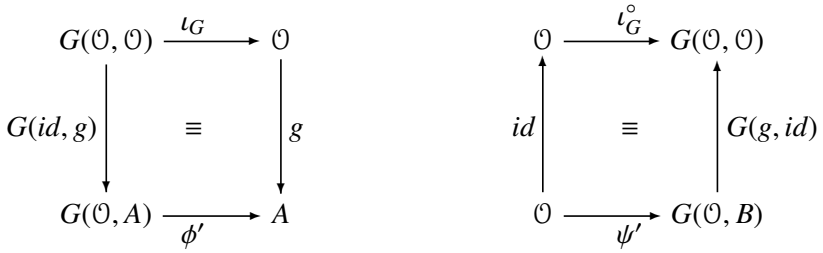
We now turn to our main result. Suppose g is an iterative map as follows:



We ask: is g definable using direcursion? "Definable" here means that it belongs to a class of functions defined inductively, closed under composition, and including elementary functions such as (co)pairing, projections/injections, constants, id, as well as any function defined by direcursion with parameters ϕ, ψ in this class. Can we, for example, choose A, B, ϕ', ψ' such that the following diagrams commute with this given g as a solution?



To simulate iteration, we wish to force $h = id$ in this definition. In other words, we must specialise the above definition as follows:



We conclude that for $h = id$ we require the following condition:

$$G(g, id) \circ \psi' = \iota^\circ$$

If g has a right inverse (g^{-1}), then we can solve this equation:

$$\psi' = G(g^{-1}, id) \circ G(g, id) \circ \psi' = G(g^{-1}, id) \circ \iota^\circ$$

We have therefore arrived at a problem: a sufficient condition is that the iterative map g has a right inverse g^{-1}, but this does not hold in many cases, for example not for all those iterative maps that fail to be surjective. For instance, if A is a standard lazy list datatype (as a cpo) then we are forced to exclude maps without e.g. the empty lists (or infinite lists etc) in their image. We therefore would like to have a more generally applicable solution. Since we have merely identified a sufficient condition, we are not in a hopeless situation. It turns out that primitive direcursion provides a solution to this problem:

Theorem 3.3 (Iteration as direcursion)**.** *Let G be a locally continuous functor $\mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ and suppose that $\mathcal{O}$ is the carrier of the initial G-dialgebra. Suppose that $h = (\!|\phi|\!)_{G(\mathcal{O},\_)}$. Then $h = \pi_2 \circ (\!|\phi', \psi'|\!)$ with $\phi' : G(\mathcal{O}, \mathcal{O} \times A) \to \mathcal{O} \times A$ and $\psi' : \mathcal{O} \to G(\mathcal{O} \times A, \mathcal{O})$ given by*

$$\phi' = (\iota \times \phi) \circ \langle G(id, \pi_1), G(id, \pi_2) \rangle, \text{ and} \tag{1}$$

$$\psi' = G(\pi_1, id) \circ \iota^\circ. \tag{2}$$

Proof. We begin by asking when the following diagrams will commute where we force $g = \langle id, h \rangle$.

$$\begin{array}{ccc} G(\mathcal{O}, \mathcal{O}) & \xrightarrow{\iota_G} & \mathcal{O} \\ {\scriptstyle G(id, \langle id, h \rangle)}\downarrow & \equiv & \downarrow{\scriptstyle \langle id, h \rangle} \\ G(\mathcal{O}, \mathcal{O} \times A) & \xrightarrow[\phi']{} & \mathcal{O} \times A \end{array} \qquad \begin{array}{ccc} \mathcal{O} & \xrightarrow{\iota_G^\circ} & G(\mathcal{O}, \mathcal{O}) \\ {\scriptstyle id}\uparrow & \equiv & \uparrow{\scriptstyle G(h, id)} \\ \mathcal{O} & \xrightarrow[\psi']{} & G(\mathcal{O} \times A, \mathcal{O}) \end{array}$$

For the right hand square we must find a canonical ψ' satisfying the following property:

$$G(\langle id, h \rangle, id) \circ \psi' = \iota^\circ$$

But we can take $\psi' = G(\pi_1, id) \circ \iota^\circ$ and show that it satisfies this property:

$$G(\langle id, h \rangle, id) \circ G(\pi_1, id) \circ \iota^\circ = G(\pi_1 \circ \langle id, h \rangle, id) \circ \iota^\circ = G(id, id) \circ \iota^\circ = \iota^\circ$$

It remains to show that we can always define also ϕ' such that the left hand square also commutes. To define ϕ' we use that h is $G(\mathcal{O}, \_)$-iterative, i.e.

$$h \circ \iota = \phi \circ G(id, h).$$

From this property we will then prove that $\langle id, h \rangle \circ \iota = \phi' \circ G(id, \langle id, h \rangle)$, by defining a canonical ϕ' from ϕ as follows:

$$\phi' = (\iota \times \phi) \circ \langle G(id, \pi_1), G(id, \pi_2) \rangle$$

We can now establish the result for parametric iterative maps:

$$\begin{aligned}
\langle id, h\rangle \circ \iota &= (\iota \times \phi') \circ \langle G(id, \pi_1), G(id, \pi_2)\rangle \circ G(id, \langle id, h\rangle)) && (3)\\
&= (\iota \times \phi') \circ \langle G(id, \pi_1) \circ G(id, \langle id, h\rangle), G(id, \pi_2)\rangle \circ G(id, \langle id, h\rangle)) && (4)\\
&= (\iota \times \phi') \circ \langle G(id, id), G(id, h)\rangle && (5)\\
&= (\iota \times \phi') \circ \langle id, G(id, h)\rangle && (6)\\
&= \langle \iota, \phi' \circ G(id, h)\rangle && (7)\\
&= \langle \iota, h \circ \iota\rangle = \langle id, h\rangle \circ \iota && (8)
\end{aligned}$$

Note how we in (7) used that h is iterative. □

This result immediately dualises:

Theorem 3.4 (Coiteration as direcursion)**.** *Let G be a locally continuous functor $\mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ and suppose that $\mathcal{O}$ is the carrier of the initial G-dialgebra. Suppose that $h = [\![\psi]\!]_{G(\mathcal{O},\_)}$. Then $h = [\![\phi', \psi']\!] \circ \mathsf{inr}$ with $\phi' : G(\mathcal{O} + B, \mathcal{O}) \to \mathcal{O}$ and $\psi' : \mathcal{O} + B \to G(\mathcal{O}, \mathcal{O} + B)$ given by*

$$\phi' = \iota \circ G(\mathsf{inl}, id), \text{ and} \quad (9)$$

$$\psi' = [G(id, \mathsf{inl}), G(id, \mathsf{inr})] \circ (\iota^\circ + \psi). \quad (10)$$

We consider the first theorem again. Can we eliminate also the postcomposed projection in the construction? For this we seek a dialgebra homomorphism $(\pi_2, id) : (\mathcal{O}\times A, \mathcal{O}, \phi'_\phi, \psi'_\phi) \to (A, \mathcal{O}, \phi'', \psi'')$ (in order to use fusion), i.e. we require (for ϕ' and ψ' as in the theorem):

$$\begin{array}{ccc}
G(\mathcal{O}, \mathcal{O}\times A) & \xrightarrow{\phi'} & \mathcal{O}\times A \\
\Big\downarrow{\scriptstyle G(id,\pi_2)} & \equiv & \Big\downarrow{\scriptstyle \pi_2} \\
G(\mathcal{O}, A) & \xrightarrow[\phi'']{} & A
\end{array}
\qquad\qquad
\begin{array}{ccc}
\mathcal{O} & \xrightarrow{\psi'} & G(\mathcal{O}\times A, \mathcal{O}) \\
\Big\uparrow{\scriptstyle id} & \equiv & \Big\uparrow{\scriptstyle G(\pi_2,id)} \\
\mathcal{O} & \xrightarrow[\psi'']{} & G(A, \mathcal{O})
\end{array}$$

That is, we wish to find another dialgebra (ϕ'', ψ'') such that the unique homomorphism from the initial G-dialgebra into that dialgebra factors through the maps given in this diagram. The right-hand diagram forces the following property (using the definition of ψ'):

$$G(\pi_2, id) \circ \psi'' = \psi' = G(\pi_1, id) \circ \iota^\circ_G$$

For the left-hand side to commute we require:

$$\phi'' \circ G(id, \pi_2) = \pi_2 \circ \phi' = \phi \circ G(id, \pi_2)$$

We conclude that in general it will not be possible to eliminate the postcomposed projection, and dually not the precomposed injection. However, a sufficient condition is that the iterative map g is split epic with right inverse g^{-1} as initially remarked above. We close the section by summarising the development:

Corollary 3.3 (Main Result)**.** *Suppose $G : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ is a locally continuous bifunctor and that $(\mathcal{O}, \mathcal{O}, \iota, \iota^\circ)$ is the initial G-dialgebra. Then the following is true for arbitrary maps $\phi : G(\mathcal{O}, A) \to A$ and $\psi : B \to G(\mathcal{O}, B)$ and α, β:*

$$(\!|\phi|\!)_{G(\mathcal{O},-)} = \langle\!|\phi \circ G(id, \pi_2), \alpha|\!\rangle_G, \text{ and} \tag{11}$$

$$[\!(\psi)\!]_{G(\mathcal{O},-)} = |\!\rangle \beta, G(id, \mathsf{inr}) \circ \psi |\!\langle_G. \tag{12}$$

In particular, we can take α and β to be the unique maps that factors through the zero object for a suitably chosen object, e.g. zero $0 \cong 1$ itself:

$$\begin{aligned} \alpha &= \bot_{1, G(\mathcal{O}\times A, \mathcal{O})} : 1 \to G(\mathcal{O} \times A, \mathcal{O}) \\ \beta &= \bot_{G(\mathcal{O}+B, \mathcal{O}), 1} : G(\mathcal{O} + B, \mathcal{O}) \to 1 \end{aligned}$$

4 Example: Application to Object Calculus Semantics

Direcursion arises naturally in self-application semantics [17] of typed object calculus based on recursive types, which has recently been subject to some research [25, 27, 16, 15]. In this section, we will consider the interpretation in Glimming et al [16], but for concreteness we work in $\mathsf{CPPO}_{\bot!}$ rather than a category of partial maps. For example, $\mathbb{B}_\bot$ is a flat cpo with underlying set $\{\mathtt{tt}, \mathtt{ff}, \bot\}$, and all maps are strict. We conclude the paper by giving some examples. Note that all of these examples (and others) have been implemented in a lazy functional programming language:

Example 4.1 (Object-Based Natural Numbers)**.** *Abadi and Cardelli [1] model object-based "natural numbers" essentially by defining a type $\sigma = Obj(X)$*[pred : X, zero : Bool]*, and then giving suitable terms to represent numbers (see loc. cit.). Under self-application semantics, this object type is modelled as a solution to the domain equation $\mathcal{O} \cong \mathcal{O} \multimap (\mathcal{O}_\bot \times \mathbb{B}_\bot)$. Writing $(\mathcal{O}, \mathcal{O}, \iota, \iota^\circ)$ for the initial dialgebra arising from the induced mixed variant functor G, we can define a map*

sapp : $\mathcal{O} \to (\mathcal{O}_\perp \times \mathbb{B}_\perp)$ *by* *sapp* = *eval* $\circ \langle \iota^\circ, id \rangle$. *Now we have for example* $\iota(\lambda x.(\perp, \perp)), \iota(\lambda x.(x, \perp)) \in \mathcal{O}$ *and also zero* $= \lceil 0 \rceil = \iota(\lambda x.(x, \mathtt{tt})) \in \mathcal{O}$. *Moreover, define* $\lceil n+1 \rceil = \iota(\lambda x.(\lceil n \rceil, \mathtt{ff})) \in \mathcal{O}$ *for* $n \in \mathbb{N}$. *Note that in general the "methods" need not be constant functions, but can depend on the current "self" in a non-trivial way. Now the extraction of a natural number in* $\mathbb{N}_\perp$ *(given by a flat cpo) from an "object" in* $\mathcal{O}$ *can be defined as a* $G(\mathcal{O}, \_)$*-iterative function as follows:*

$$\begin{array}{ccc} G(\mathcal{O},\mathcal{O}) & \xrightarrow{\iota} & \mathcal{O} \\ \Big\downarrow{\scriptstyle G(id,h)} & \equiv & \Big\downarrow{\scriptstyle h} \\ G(\mathcal{O},\mathcal{O}\times\mathbb{N}_\perp) & \xrightarrow[\phi]{} & \mathcal{O}\times\mathbb{N}_\perp \end{array}$$

In the diagram, ϕ *is the following algebra map for taking one step during the extraction:*

$$\phi = f \circ \langle \pi_2, eval \rangle \circ \langle G(1, \pi_{1\perp} \circ \pi_1), \iota \circ G(1, \langle \pi_{1\perp} \circ \pi_1, \pi_2 \rangle) \rangle$$

where $f : \mathcal{O} \times (\mathbb{N}_\perp)_\perp \to \mathcal{O} \times \mathbb{N}_\perp$ *is defined as follows (for usual strict addition +):*

$$f(o,n) = \begin{cases} (o, 0), & \text{if } \pi_2 \circ \mathit{sapp}(o) = \mathtt{tt} \\ (\pi_1 \circ \mathit{sapp}(o), m+1), & \text{if } n = m_\perp \\ (o, \perp), & \text{otherwise} \end{cases}$$

Note that the first case applies when the zero *method evaluates to* `tt`*, and that the second gives back the predecessor in the first component. It can now be inferred that parametric iterative maps can be useful for defining functions on objects, since more involved examples can be constructed similar to this simplified one for "natural numbers". We have shown in this paper that the map h can equivalently be defined as* $h = \pi_1 \circ k$ *where* $k = (\!|\phi', \psi'|\!)_G$ *for a suitable dialgebra* (ϕ', ψ') *as detailed in previous sections. The resulting definition is shown in the following diagrams:*

$$\begin{array}{ccc} G(\mathcal{O},\mathcal{O}) & \xrightarrow{\iota} & \mathcal{O} \\ \Big\downarrow{\scriptstyle G(id,k)} & \equiv & \Big\downarrow{\scriptstyle k} \\ G(\mathcal{O},\mathcal{O}\times(\mathcal{O}\times\mathbb{N}_\perp)) & \xrightarrow[\phi']{} & \mathcal{O}\times(\mathcal{O}\times\mathbb{N}_\perp) \end{array} \qquad \begin{array}{ccc} \mathcal{O} & \xrightarrow{\iota^\circ} & G(\mathcal{O},\mathcal{O}) \\ \Big\uparrow{\scriptstyle g=id} & \equiv & \Big\uparrow{\scriptstyle G(k,id)} \\ \mathcal{O} & \xrightarrow[\psi']{} & G(\mathcal{O}\times(\mathcal{O}\times\mathbb{N}_\perp),\mathcal{O}) \end{array}$$

Note that we need two different $\mathcal{O}$ here, since we cannot invert sapp*. Our main result states that we have $\phi' = (\iota \times \phi) \circ \langle G(id, \pi_1), G(id, \pi_2)\rangle$ and $\psi' = G(\pi_1, id) \circ \iota^\circ$. (We can alternatively and equivalently define $h = \langle\!|\phi \circ G(id, \pi_2), \bot|\!\rangle_G$ according to Corollary 3.3.)*

Example 4.2 (Constructors)**.** *We next define a direcursive function δ which creates an "object" when given a natural number, i.e. a "constructor". In this case, we consider the type $\sigma' = Obj(X)[\text{pred} : X, \text{zero} : \text{Bool}, \text{succ} : X]$, which allows a method for computing the successor of the number stored in an object as well. A domain $\mathcal{O}'$ arises from the equation $\mathcal{O}' \cong \mathcal{O}' \multimap (\mathcal{O}'_\bot \times \mathbb{B}_\bot \times \mathcal{O}'_\bot)$. We dub the induced mixed variant functor H, and define δ first using full direcursion:*

$$\begin{array}{ccc} H(\mathcal{O}', \mathcal{O}') & \xrightarrow{\iota_H} & \mathcal{O}' \\ {\scriptstyle H(\delta,\gamma)}\downarrow & \equiv & \downarrow{\scriptstyle \gamma} \\ H(\mathbb{N}, 1) & \xrightarrow[\bot]{} & 1 \end{array} \qquad \begin{array}{ccc} \mathcal{O}' & \xrightarrow{\iota_H^\circ} & H(\mathcal{O}', \mathcal{O}') \\ {\scriptstyle \delta}\uparrow & \equiv & \uparrow{\scriptstyle H(\gamma,\delta)} \\ \mathbb{N} & \xrightarrow[\delta^-]{} & H(1, \mathbb{N}) \end{array}$$

The unique solution arises by providing δ^- as follows:

$$\delta^-(n) = \lambda x.(n - 1, n \equiv 0, n + 1)$$

Note that the positive part of the diagram trivialises in this definition. In fact, there is a more natural parametric coiterative definition:

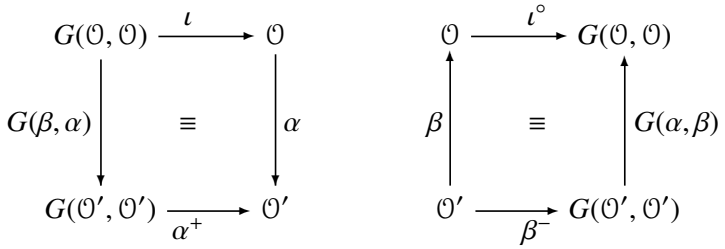
$$\begin{array}{ccc} \mathcal{O}' & \xrightarrow{\iota_H} & H(\mathcal{O}', \mathcal{O}') \\ {\scriptstyle \delta'}\uparrow & \equiv & \uparrow{\scriptstyle H(id,\delta')} \\ \mathcal{O}' \times \mathbb{N}_\bot & \xrightarrow[\delta'^-]{} & H(\mathcal{O}', \mathcal{O}' \times \mathbb{N}_\bot) \end{array}$$

Here, we require the following $H(\mathcal{O}, \_)$-coalgebra:

$$\delta'^-(n) = \lambda o.((o, n - 1), n \equiv 0, (o, n + 1))$$

Using Corollary 3.3 we have $\delta' = \flat\bot, H(id, \mathsf{inr}) \circ \delta'^- \sharp_H$. The general problem of proving that two maps such as δ' and δ compute the same "objects", requires suitable bisimulations (see example 5 and 6 below).

Example 4.3 (Subtyping)**.** *Using direcursion, we will define an embedding-projection (ep-) pair* $(\alpha, \beta) : \mathcal{O} \to \mathcal{O}'$*. Here* β *serves as a* coercion function *for subtyping, whereas* α *gives an* approximation function *from a type into the given supertype. The pair* (α, β) *is the unique solution to the following diagrams:*



In these diagrams, we have chosen α^+ *and* β^- *as follows:*

$$\alpha^+ = \iota \circ \langle \pi_1, \pi_2, \bot \rangle^{id}$$
$$\beta^- = \langle \pi_1, \pi_2 \rangle^{id} \circ \iota^\circ$$

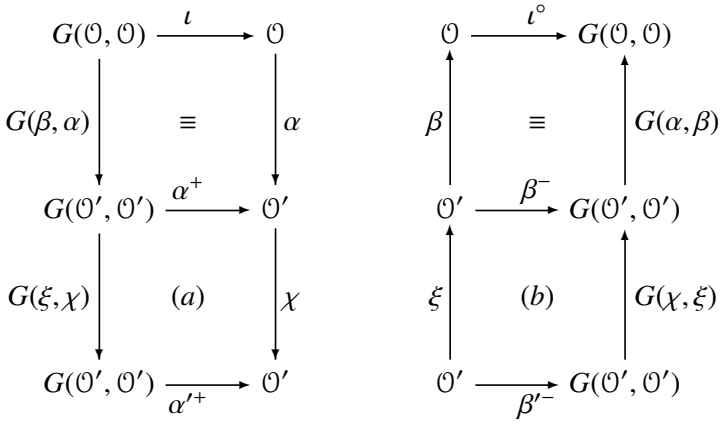
Here $\bot : \mathcal{O}'_\bot \times \mathbb{B}_\bot \to \mathcal{O}'_\bot$ *is the constantly undefined map. It is straightforward to show that* $\beta^- \circ \alpha^+ = id$ *and moreover that* $\alpha^+ \circ \beta^- \sqsubseteq_{\mathcal{O}'} id$*. (Recall that* $\sqsubseteq_{\mathcal{O}'}$ *is the coordinatewise order inherited from the infinitary product which determines* $\mathcal{O}'$*.) It follows that* (α, β) *is an ep-pair since G is locally continuous. Note that since subtyping uses full direcursion, we can use a constructor that works with both* $\mathcal{O}'$ *and* $\mathcal{O}$*, since the (*prim-Fusion*) rule can be used, once both maps are given in direcursive form using our main result.*

Example 4.4 (Inheritance)**.** *We continue with the previous example by also considering the following function:*

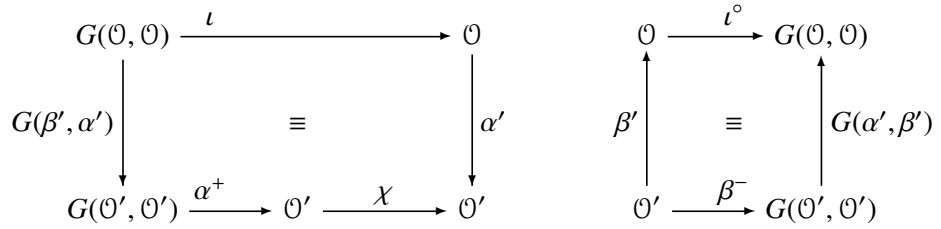
$$\chi(o) = \iota(\lambda p.(\pi_1 \circ \iota^\circ(o)(p), \pi_2 \circ \iota^\circ(o)(p), \iota(\lambda q.(p, \texttt{ff}, \iota^\circ(p)(q)))))$$

Note how χ *serves to interpret the successor method from Abadi et al [1]. It takes an element in* $\mathcal{O}'$ *and equips it with the successor method. The definition involves both using method updates to copy the previous predecessor and zero state, but also the third component which adds "*succ*" such that it becomes constantly defined in*

the recursive structure. Now consider χ together with (α,β):

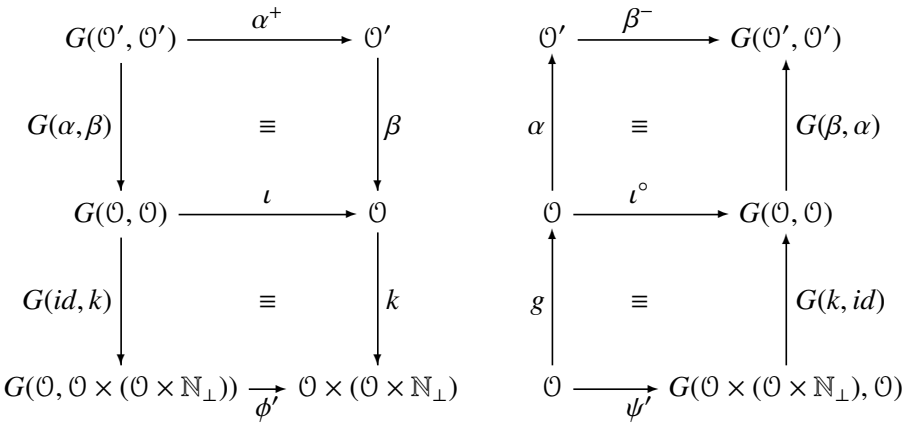


To unveil inheritance *as a direcursive map, all we need to do is to find a dialgebra (α'^+,β'^-) and a map ξ making (χ,ξ) a dimap into this new G-dialgebra. Since this is a "standard" update (i.e. of the form Abadi and Cardelli considers in typed object calculi), it is constant in the recursive structure. This means that the fusion law gives us a unique dimap (α',β') for inheritance with $\xi = id$ and $\beta'^- = \beta^-$:*



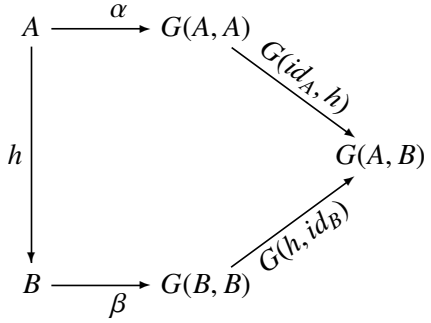
*(This gives again an ep-pair.) We would like to use the parametric (co)iterative operations from example 1 and 2 also after we have inherited some of $\mathcal{O}$ into the new $\mathcal{O}'$. To use the fusion rule we must first apply Corollary 3.3 to the parametric coiterative map. We "rotate" the diagrams for the ep-pair and paste them to the definition of extraction provided in example 2, after which the (*prim-FUSION*) rule can readily be used once again, illustrating the usefulness of our results. However, if we attempt this for example 1 instead (without moving to H-dialgebras), we*

arrive at these seemingly awkward diagrams:



The diagrams that arise generalise a so-called "hylomorphism" (well-studied in purely functional program algebra, see e.g. [21, 19]), i.e. that of a coiterative map followed by a iterative map. Here, both maps are instead direcursive, but the diagrams are combined such that the negative part of direcursion is followed by the positive part of another direcursive map. Ongoing work investigates if e.g. the so-called shift law [19] generalises to this setting using dinatural transformations.

Example 4.5 (Generalised Bisimulations)**.** *Tews [31] in his PhD thesis based a notion of higher-order bisimulation on a certain kind of morphism between what he termed generalised coalgebras. These maps, together with an associated notion of bisimulation, were in his thesis demonstrated as useful in a number of practical examples. However, Tews found that almost all important closure properties failed to hold in the category of sets unless strong restrictions were imposed. Via Corollary 3.3, we have now linked Tews' work to Freyd's direcursion principle. To see this, let h be a generalised coalgebra map in Tews' sense, i.e. let the following diagram commute:*



One readily identifies h as a parameterised $G(A, \_)$-coalgebra map from (A, α) to $(B, G(h, id_B) \circ \beta)$ so our results apply to these maps. Existence of final generalised coalgebras appears to be an open problem [31], but note that for us they are special cases of a final parameterised coalgebras (with a carrier isomorphic to that of the initial G-dialgebra). As Tews' remarks, a coiteration scheme (as well as a proof principle) of potential practical utility [31] would follow, if we can find a class of coalgebras which guarantees the existence of such coalgebraic extensions. We have in this paper shown that these extensions (when they exist) are primitive direcursive maps, and hence unqiue. However, the results presented in this paper do not close this problem but rather reopen it, and set the stage for further investigations (existence may be subject to strong side conditions, so a class of coalgebras would have to be identified, e.g. by verifying closure properties à la Birkhoff). Moreover, the ambient category that we have used circumvents Tews' counterexample.

Example 4.6 (Self-Applicative Bisimulations)**.** *Note that* sapp *can be made to carry a $G(\mathcal{O}, \_)$-coalgebra. This is achieved by defining $\mathcal{O} \to G(\mathcal{O}, \mathcal{O})$ by $K \circ$* sapp *where K is the usual combinator (we abuse notation slightly and call this map* sapp *as well). Hence there is a* parametric coiterative *function $[\![\mathsf{sapp}]\!]_{G(\mathcal{O},\_)}$ from $\mathcal{O}$ into the final $G(\mathcal{O}, \_)$-coalgebra. The kernel $\sim_{\mathsf{sapp}}$ of this coalgebra map is a bisimulation equivalence [26]. Thus $\mathcal{O}/\sim_{\mathsf{sapp}}$ is a well-defined quotient (on the underlying set). This bisimulation collapses object-based programs into class-based programs. The results of the present paper allow us to view this map as a dialgebra homomorphism $\langle\!|\mathsf{sapp} \circ G(id, \pi_2), \bot|\!\rangle_G$ (by Corollary 3.3), so that it can be combined with the above examples (or more complex future ones). Note that $\sim_{\mathsf{sapp}}$, although giving the link to class-based languages, is not the most suitable bisimulation for identifying object-based programs (and a type-indexed set of similar domains $\mathcal{O}_\sigma$ do not give a fully abstract model of object calculus, since Viswanathan's counterexample [34] can readily be applied). For this we require instead a finer notion, which we term OBP-bisimulation. It requires a structure $\beta : \mathcal{O} \to G(\mathcal{O} \times \mathcal{L}^+, \mathcal{O})$, which combines both* mup *and* sapp *behaviour, where* mup *: $\mathcal{O} \times \mathcal{L} \times \mathcal{O} \to \mathcal{O}$ copies given method $\ell \in \mathcal{L}$ from the first argument to the third. Here $\beta =$* sapp $\circ$ mup$^+$ *where* mup$^+$ *is a generalisation of method update which copies all the labels listed in $\mathcal{L}^+$ before the update. ($\mathcal{L}^+$ is a finite sequence of labels to be updated.) Note that β is not a coalgebra map. We therefore regard β as the negative part of a dialgebra map, i.e. require full direcursion. This outlines our current direction for future work. We emphasise that our main result already shows how $\sim_{\mathsf{sapp}}$ (and any other bisimulation arising in this way) can be combined*

with Freyd's direcursion so that a previous gap has been closed.

5 Conclusions and Further Work

In the 1990s, Freyd demonstrated in two well-known papers [11, 12] a principle for defining functions on recursive types. However, he did not discuss how parametric (co)iterative maps can be written using his scheme. We have shown here that all such maps *can* in fact easily be defined, after we found and developed a certain variation of Freyd's principle that we termed primitive direcursion (generalising primitive recursion for endofunctors). Moreover, we have established some elementary algebraic properties of primitive direcursion. Taken together, our result can be viewed as a set of corollaries of Bekič's Lemma, giving a link between direcursion and parameterised datatypes via a recursion scheme.

A consequence of our result is that we demonstrate that many direcursive maps (previously known as difficult to exemplify [20]) arise by translating (more common) (co)iterative maps into Freyd's principle. As an additional consequence, functions defined on the parameterised initial algebra (and dually final coalgebra) can be combined (as per the usual "fusion laws") with more involved functions (e.g. the interpreters from [20]) which require full direcursion. In such situations, we can now use the reasoning principles of direcursion, even if the (co)iterative function is not surjective (not injective), and we in such cases avoid having to provide an "inverse-like" function which was previously identified as a problem [8]. We have given some examples from the semantics of object calculi, which demonstrates a situation where "fusion laws" can be used, showing that our results could be practically useful. Current work aims to take this further and establish an algebra of object-based programs in the spirit of Bird et al [5].

The author is presently developing bisimulations in the setting of denotational models of object calculus, in the vein of the work of Abramsky [2], with a goal to combine these with "fusion laws". Indeed the developments of Fiore [10] show that applicative bisimulations can be internalised into $\mathbb{C}$. Notably, such bisimulations use merely the parameterised final coalgebra since the contravariant argument remains fixed.

Another topic for further investigation is the expressivity of (primitive) direcursion. One would like to know if it is possible to naturally capture known classes of functions given by circular definitions (in the spirit of e.g. [6], but see also [14, 7]). The author is particularly interested in characterising the total direcursive maps.

Finally, taking the two kinds of maps given in Corollary 3.3 together (with two suitable instantiations), we have the following (as in one of our examples):

$$\pi_2 \circ \langle\!|\phi, G(id, \pi_2), \alpha|\!\rangle \circ |\!\rangle\beta, G(id, \mathsf{inr}) \circ \psi|\!\langle \circ \mathsf{inl} : A \to B$$

For future work, we ask: when can direcursive maps be split into a pair of parametric (co)iterative maps of this form essentially computing results in two separate stages (like e.g. *fix* or quicksort)? Note that we can allow two different (di)naturally related mixed-variant functors, and choose A, B as well as an intermediate $\mathcal{O}$ into which all partial results can be embedded. We hope that such an analysis could further our understanding of direcursion.

Acknowledgements

I wish to express my gratitude in particular to John Power, Bernhard Reus, and to Viggo Stoltenberg-Hansen, for encouragement and helpful discussions on the research reported here, and to the anonymous referees for their comments on an earlier version of this paper.

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[3] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.

[4] H. Bekič. Definable operation in general algebras, and the theory of automata and flowcharts. In *Programming Languages and Their Definition - Hans Bekic (1936-1982)*, pages 30–55, London, UK, 1984. Springer-Verlag.

[5] R. S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[6] D. Cancila, F. Honsell, and M. Lenisa. Generalized coiteration schemata. *Electronical Notes in Computer Science.*, 82(1), 2003.

[7] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Symposium on Principles of Programming Languages*, pages 206–217. ACM Press, 2006.

[8] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[9] M. P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[10] M. P. Fiore. A coinduction principle for recursive data types based on bisimulation. *Information and Computation*, 127(2):186–198, 1996.

[11] P. J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90*, pages 498–507. IEEE Computer Society Press, June 1990.

[12] P. J. Freyd. Algebraically complete categories. In *Proceedings 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104, 1991.

[13] P. J. Freyd. Remarks on algebraically compact categories. In Fourman, Johnstone, and Pitts, editors, *Workshop on Applications of Categories in Computer Science, Proceedings of the London Mathematical Society Symposium*, number 177 in London Mathematical Society Lecture Note Series, pages 95–106. Cambridge University Press, 1992.

[14] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? In *Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science*, volume 44 of *Electronic Notes in Theoretical Computer Science*, April 2001.

[15] J. Glimming. Dialgebraic semantics of typed object calculi. Licentiate Thesis, May 2005. ISBN 91-7178-031-9, TRITA-NA-0511.

[16] J. Glimming and N. Ghani. Difunctorial semantics of object calculus. In *Electronic Notes in Theoretical Computer Science*, volume 138. Elsevier, Novem-

ber 2005. Proceedings of the Second Workshop on Object Oriented Developments (WOOD 2004).

[17] S. N. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

[18] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[19] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[20] E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In S. Peyton-Jones, editor, *Functional Programming Languages and Computer Architecture*, pages 324–333. Association for Computing Machinery, 1995.

[21] R. Paterson. Operators. In *Lecture Notes for the International Summerschool on Constructive Algorithmics, Ameland, Netherlands*. CWI Amsterdam, Utrecht University, University of Nijmegen, September 1990.

[22] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[23] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996.

[24] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[25] B. Reus and T. Streicher. Semantics and logic of object calculi. *Theorertical Computer Science*, 316(1):191–213, 2004.

[26] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[27] J. Schwinghammer. *Reasoning about Denotations of Recursive Objects*. Ph.D. thesis, University of Sussex, 2005.

[28] D. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 97–136. Springer-Verlag, 1972.

[29] M. Smyth and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[30] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, 1994.

[31] H. Tews. *Coalgebraic Methods for Object-Oriented Specification*. PhD thesis, Technical University of Dresden, 2002.

[32] T. Uustalu and V. Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10(1):5–26, 1999.

[33] V. Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, 2000.

[34] R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS), 1998)*. IEEE Computer Society, 1998.

[35] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 249–262, Uppsala, Sweden, August 2003. ACM SIGPLAN.

Chapter 6

Concluding Remarks

In this thesis, we have studied problems that arise in the denotational semantics of typed object calculus (including a new recursion scheme which is of more general interest). We asked three main questions in the introductory chapter.

The first question was: *what is an object?* In the preceding chapters, we have modelled an object as a particular form of recursive type, equipped with maps for method invocation and method update. Our computational adequacy result shows that this answers the question as far as pragmatics are concerned (with the exception of subtyping, which is an open problem). (But see our discussion on the problem of full abstration below.)

The second question we stated was: *what is a model of object calculus?* To this question, we provided a first answer in Paper I, by saying that a model is a category of partial maps which is partial cartesian closed, algebraically compact and has coproducts (when sum types are included in the calculus). These axioms are not sufficient to ensure that we have a computationally adequate model, however, and we therefore proceeded in Paper II by giving a formal connection to models of FPC. We show there that a model of the object calculus under consideration in Paper II is obtainable from any model of lazy FPC, such that computational adequacy is preserved.

The third question we formulated was: *how can object-based programming be improved?* We have shown that every object type is equipped with a recursion principle (direcursion) and, further, that this recursion principle can be used for defining and reasoning with objects (in the case study of Paper III and via encoded algebraic datatypes in Paper I). We have shown that any program transformation valid (on object denotations) in a model of lazy FPC induces a sound program law

(Paper II). This is a fundamental result needed for the development of a program algebra for object-based programs. In this way, we have also provided a foundation for the development of polytypic object-based programming languages (in the vein of Jansson [Jan00] or Weirich [Wei02]; the former is based on the theory of algebraic datatypes, while we have studied recursive/dialgebraic datatypes here).

In addition, Paper III provides a theoretical result of relevance to axiomatic domain theory and not only to models of typed object calculus. There, we demonstrate a correspondence between direcursive maps on recursive datatypes and certain (co)recursive maps on associated parametric datatypes.

Paper I

In the first paper, we developed a denotational semantics different from the one given in [AC96]. While Abadi et al use the ideals/metric approach [MPS84], we used an approach based on Fiore's category of partial maps instantiated for pCpo, thus mimicking the more abstract order-enriched setting of [SP82, AP90]. Secondly, Abadi and Cardelli interpret types as partial equivalence relations (pers) over a universal domain, while we interpret object types by solving recursive type equations in pCpo. As a result, we get a more intuitive model of objects, with an associated principle of recursion. We think the translation of inductive types into wrappers shows the simplicity and naturalness of this model. We also showed that every object carries a coalgebra structure, and so is equipped with a notion of bisimilarity. This appears to be the first category-theoretic semantics of typed object calculus.

The author has recently discovered a communication operator $\odot$ which allows two objects to exchange information (interact over a joint interface). The operational semantics of this operator interleaves the execution of two objects such that the left argument initiates a communication with the right argument (over the interface given by a suitable joint supertype). After each such step, the roles are reversed, as the following preliminary operational semantics shows:

$$\frac{m \Downarrow \overbrace{(Obj(X=A)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}}^{o} \odot p) \qquad b_i\{x \mapsto p \odot o\} \Downarrow q}{m.\ell_i \Downarrow q}$$

$$\frac{o \Downarrow o' \qquad p \Downarrow p'}{o \odot p \Downarrow o' \odot p'} \qquad \frac{o \odot p \Downarrow q}{(o \odot p).\ell \Leftarrow \varsigma(x : X)b \Downarrow (q.\ell \Leftarrow \varsigma(x : X)b)}$$

where o, p, q are values.

Ongoing work aims to give a denotational semantics for object calculus extended with this communication operator, while also studying if there exists any relationships between it and communication operators in process calculi. Interestingly, it seems possible to define $[\![a \odot b]\!]$ using direcursion.

Paper II

In this paper, we developed an interpretation of a variation of Abadi and Cardelli's first-order typed object calculus (with functions, coproducts and products) into Plotkin's FPC. We proved computational adequacy and soundness in the sense that each model of FPC with lazy operational semantics with these properties is also such a model of typed object calculus. Beyond the theorems that establish this, a notable result is that while we considered both eager and lazy variations of FPC, it turned out that lazy FPC is needed for this result to hold. As an immediate result, we have that Winskel's computationally adequate denotational semantics of lazy FPC [Win93] is a computationally adequate model of the variation of typed object calculus under consideration. Since a direct proof of computational adequacy is rather complicated in the presence of recursive object types (compare to the work by Fiore and Plotkin [FP94]), the use of an interpretation into the metalanguage turned out to simplify matters substantially, while in the end giving the same result.

One important result established in Paper II is that if we have that $[\![\ulcorner t \urcorner]\!] = [\![\ulcorner s \urcorner]\!]$ holds in a denotational model of lazy FPC which is computationally adequate, then it follows that $t \simeq s$ in the typed object calculus under consideration (i.e. the terms t and s are observationally congruent). This establishes that such a model of lazy

FPC can be used in the study of program algebras for object-based programming languages.

Paper III

In two well-known papers from the 1990s, Freyd demonstrated a principle for defining functions on recursive types [Fre90, Fre91]. However, he did not discuss how parametric (co)iterative maps can be written using his scheme. We have shown here that all such maps *can* in fact easily be defined, after we found and developed a certain variation of Freyd's principle that we termed primitive direcursion (generalising primitive recursion for endofunctors). Moreover, we have established some elementary algebraic properties of primitive direcursion. Taken together, our result can be viewed as a set of corollaries of Bekič's Lemma, giving a link between direcursion and parameterised datatypes via a recursion scheme.

A consequence of our result is that we demonstrate that many direcursive maps (previously known as difficult to exemplify [MH95]) arise by translating (more common) (co)iterative maps into Freyd's principle. As an additional consequence, functions defined on the parameterised initial algebra (and dually final coalgebra) can be combined (as per the usual "fusion laws") with more involved functions (e.g. the interpreters from [MH95]) which require full direcursion. In such situations, we can now use the reasoning principles of direcursion, even if the (co)iterative function is not surjective (not injective), and we avoid having to provide an "inverse-like" function which was previously identified as a problem [FS96]. We have given some examples from the semantics of object calculi, which demonstrates a situation where "fusion laws" can be used, showing that our results could be practically useful. Current work aims to take this further and establish an algebra of object-based programs in the spirit of Bird et al [BdM97]. The author has recently also studied a generalisation of this result called parametric direcursion. This would allow not only the carrier of the free dialgebra to be a parameter, but also other datatypes such as the set of method labels (for maintaining a label trace during computation with objects).

The relationship between parametric (co)iterative maps and Freyd's scheme (via primitive direcursion) has relevance for bisimilarity, since applicative bisimilarity is given on the parameteric final coalgebra for untyped lambda calculus (lazy or otherwise). This suggests that our results may be useful for further work on bisimilarity particularly when one wishes to move from coinductive types to mixed-variant/recursive types, i.e. to a higher order.

Further Investigations

In this final section, we will sketch how we hope to proceed with this research. Although Paper III provides some novel insights on Freyd's mixed-variant recursion scheme (notably, the relationship to parametric (co)algebras and thus bisimilarity), several open problems (identified below, most of which have been approached to some extent already) remain to be investigated before Freyd's scheme can be claimed to be fully analysed. Other topics for further work appear within the area of denotational semantics of typed object calculus, where at least two substantial open problems exist: fully abstract denotational model and subtyping. Abadi [Aba07] agees with the author that it remains open whether subtyping can, in fact, be combined with the self-application approach followed in this thesis (apparently revising the state of the art a decade earlier, c.f. [AC96]). The author believes that this problem is to a large extent interconnected with further investigations of direcursion.

- **Set-theoretic direcursion.** Non-well-founded set theory with AFA (as studied by Forti et al [FH83] and Aczel [Acz88]) ensures via the so-called Special Final Coalgebra Theorem [Acz88] that, subject to a naturality condition (see e.g. Turi and Rutten [TR98]), functors give rise to final coalgebras. It is interesting to note that this theorem does not say anything directly about mixed-variant functors, and the author would like to investigate in detail the relationship between direcursion (or other reasoning principles on higher-order datatypes) and final semantics, and also non-well-founded set theory. It is interesting to note that algebraic compactness would require the least and greatest fixpoint of the set-operator induced by a functor to coincide, which seems to be a rather counter-intuitive requirement for sets. Moreover, there are intriguing relationships between domain theory and non-well-founded set theory, as discussed by e.g. Hallnäs [Hal85, Hal90]. One might therefore hope that an investigation of direcursion in non-well-founded set theory leads to new insights about theories of approximation, given that dialgebraic datatypes are especially challenging for domain theory (and was one of its first applications, i.e. Scott's D_∞ model of untyped lambda calculus). Note that Hallnäs considered non-well-founded sets to be (projective) limits of well-founded sets, such that a bisimilarity relation is defined in stages (much like in Abramsky's work on lazy lambda calculus [Abr90], i.e. following Moschovakis [Mos74]). It would presumably be interesting to study this construction but to admit mixed-variant functors as well, thus understanding exactly what set theory arises in this case. (Hallnäs construction lead him to

consider an axiom that turned out to be equivalent to AFA [Acz88].) Note that work by Smyth and Plotkin [SP82] on limit-colimit coincidence depends heavily on the symmetry of a subcategory of embeddings, and that one might wish to investigate some notion of embedding-projection pairs also for sets. Furthermore, as Lenisa [Len98] points out, it remains to be discovered how mixed-variant functors (à la Freyd) can be incorporated in the final semantics approach.

- **Effective direcursion.** It also seems worthwhile to try to transport work on effective algebra [SHT95], effective category theory [Kan81]) and on effective solutions to recursive domain equations [Kan79] to Freyd's notion of dialgebra (and indeed also to coalgebra). In this way, it would be interesting to try to find appropriate notions of computability on object-based programs.

- **Theory of "universal dialgebra".** It is possible to study F-bisimulations abstractly using a category-theoretic generalisation of relations as (equivalence classes of) pairs of morphisms that are jointly monic (see e.g. [FS90, McL92]). With such a generalisation, results on F-bisimulations for metric space, domain-theoretic and set-theoretic categories can all be established in a single category-theoretic framework [Tur96]. It would be interesting to develop a richer theory of dialgebra, starting from Freyd's seminal work, by following this approach. At least initially, one can work in a domain-theoretic category and try to generalise as many as possible of the standard results of universal coalgebra, e.g. the isomorphism theorems, based on a suitable notion of mixed bisimilarity/congruence relations. For algebras, it is minimality that gives the induction principle (i.e. not having subalgebras), whereas for coalgebras it is simplicty (i.e. not having proper quotients). One might expect to find an intertwined principle in the case of dialgebra, such that these two notions arises as special cases when the involved functor is dummy in its contravariant argument. Much of this seems to be closely related to previous work on relational proof principles for domains [Pit94, Pit96], though one might hope to arrive at a coherent treatment in the vein of universal (co)algebra, as developed by Birkhoff et al and Aczel et al (for which see e.g. [MT92, Rut96, Rut00]).

- **Total direcursion and total FPC.** Previous work by Berger [Ber04] proved strong normalisation for extensions of Gödel T with various forms of recursion by using an inductively defined notion of totality in domain-theoretic models. (The same author previously also studied totality more abstractly

[Ber93].) It would be interesting to investigate the extent in which similar techniques can be used also for recursive types, and if it is possible to axiomatise one or more system of recursive types similar to FPC, where all functions are strongly normalising. Berger considers in ibid. several notions of recursion, but not direcursion.

- **Extensional characterisation of direcursive maps.** Previous work by Gibbons et al [GHA01] has given alternative characterisations of those functions that are definable using iteration or coiteration (*fold* and *unfold* in the established functional programming jargon) in a set-theoretic category. The following property was identified in the case of iteration:

$$\exists \phi\ h = (\!|\phi|\!) \quad \Leftrightarrow \quad \mathrm{Ker}(Fh) \subseteq \mathrm{Ker}(h \circ \iota_F),$$

  i.e. that the kernel of the morphism h is a congruence under the initial F-algebra. A dual result follows for coiterative maps (with images instead of kernels, and the rest of the condition also dualised). It would be interesting to see if their work can be generalised to a domain-theoretic category (without arbitrary subobjects and cartesian closure) and whether this leads to combining their two criteria to a simultaneous/mixed-variant extensional characterisation. This would further our understanding of direcursion since it could give a more intuitive explanation of what maps can be defined using the scheme. The author is currently investigating this problem, based on the approach of relational spans (since a set-theoretic category is not applicable).

- **Parametric direcursion.** The author has for some time been developing a further generalisation of direcursion that supports parameters. This makes use of the adjunction $\Gamma \otimes (\cdot) \dashv (\cdot)^\Gamma$ between the tensor and the weak exponential (i.e. in $\mathbf{CPO}_{\perp!}$ smash product and strict function space), but we conjecture that generalisation to allow also other adjunctions is possible. When traversing a graph (which like an object denotation can contain cycles), it is important to keep track of the nodes that has been visited, in order to avoid revisiting them indefinitely (see e.g. [Par01] for graph traversal). For object-based programs, parametric direcursion similarly makes possible to maintain (accumulate, as it were) a trace of method labels, and thus avoids revisiting spurious paths indefinitely. This suggests to us that parametric direcursion can be a useful computation scheme for object-based programming.

- **Communication.** Previous work has extended typed object calculus with notions from π-calculus [Gor98]. Abramsky, on the other hand, have ex-

tended his lazy lambda calculus [Abr90] with simpler (or at least less complex) notions such as testers for parallel convergence. It would be interesting to investigate if object calculus extended with facilities for communication (as sketched in a previous section above) can subsume the transition to a name-passing calculus, and to evaluate the merits of either approach. In particular, it is interesting to study how far object-based programs can be used to capture various programming languages notions such as while-loops, conditionals, etc, and to what extent the communication operator can be used to combine the "local state" of such constructions incrementally (c.f. [Red96]). Object-oriented languages based on games/game semantics has recently been studied [LW06], and the communication operator we propose would amount essentially to a game being played by two objects, so a comparison to this work seems appropriate.

- **Extensionality.** The equational theory of typed object calculus as presented by Abadi and Cardelli [AC96], does not include an extensionality principle in the spirit of the η-rule of lambda calculus. It would be interesting to study such a rule, and prove that the theory extended with this rule remains consistent, which is of course guaranteed immediately if this principle is valid in a model (which is reasonable to expect since, after all, objects are in this thesis modelled as certain self-applicative functions subject to an extensionality principle in the underlying set theory). The situation should however be carefully analysed, particularly in relation to the full abstraction problem.

- **Bisimilarity.** Gordon et al [GR96a, GR96b, Gor98] have studied operational equivalences for several of Abadi and Cardelli's object calculi, by introducing a labelled transition system induced by the operational semantics. The resulting notion of bisimilarity was termed experimental equivalence and shown to coincide with contextual equivalence using Howe's syntactical proof technique [How96]. More recently, the present author has investigated notions of bisimilarity also for the denotational semantics approach, and this means the usual identification of a coalgebra structure. However, the candidate for such a morphism has the form of a map $O \to G_\sigma(O \otimes L, O)$ for the difunctor G_σ and the carrier O of the associated free dialgebra (as induced by self-application semantics). For this reason, a parametric coalgebra has not been identified, so previous work by e.g. Fiore [Fio96c] does, interestingly, not seem to apply (i.e. a more general notion of bisimilarity for dialgebras could be called for).

- **Coalgebraic models of object-based programming languages.** When the behaviour functor induced by the operational semantics has been identified, a coalgebraic model using hypersets arises (i.e. by using one of several available non-well-founded set theories). (This is known from work by e.g. Lenisa [Len96, Len98].) This could nevertheless be a useful exercise, particularly since the method update may provide some additional obstacles because of the contravariant argument that must be "frozen".

- **Fully abstract models of object calculus.** While coalgebraic techniques can give fully abstract models, it is also of primary interest to develop syntax-independent models which have this property, since this provides additional insights into the mathematical structure of objects, while also giving a framework for reasoning with programs and extending the programming language. Hence, the development of a fully abstract self-application semantics appears to be an interesting topic for further investigation; in combination with our results, research by Viswanathan [Vis98] suggests to us that this is possible also for denotational semantics. To date, the author's investigations has been based on using the notion of bisimilarity that arises from inducing a suitable extra coalgebra/dialgebra structure on the carrier O of the free dialgebra associated to an object type. To this end, the target is to use a denotational characterisation of bisimilarity (as described above) and quotient the carrier O with this relation, thus arriving at a fully abstract model up to potential parallelism (for which an expansive approach might be required, i.e. addition of a communication operator or a tester for parallel convergence in the vein of Abramsky [Abr90]). Technically, direcursion can be used for defining a suitable embedding-projection pair, thus ensuring that the "quotient" again is a domain (i.e. enriches, as it were).

- **Subtyping as homomorphisms.** With subtyping we mean a preorder $<:$ on the set of types, together with the following subsumption rule:

$$\frac{\Theta \vdash \tau <: \sigma \quad \Theta, \Gamma \vdash t : \tau}{\Theta, \Gamma \vdash t : \sigma}$$

  This rule shows that subtyping is a substitution property, which allows a term to be used at any type above its currently assigned type. The most intuitive explanation of subtyping is therefore that it means set inclusion. For example, we may say that $[\![\tau]\!] \subseteq [\![\sigma]\!]$ holds whenever τ and σ are types such that $\tau <: \sigma$, thus making the subsumption rule valid. Now, consider the

subtyping rule that is (usually) used for function types:

$$\frac{\Theta \vdash \tau' <: \tau \quad \Theta \vdash \sigma <: \sigma'}{\Theta \vdash \tau \to \sigma <: \tau' \to \sigma'}$$

Suppose now that we have two types *Float* and *Int* of some kind of floating point numbers and integers, such that $Int <: Float$ following our intuition about these "sets". We would expect a function $f : Int \to Int$ also to be typable as $f : Int \to Float$, and, in general, for any type σ such that $Int <: \sigma$, as $f : Int \to \sigma$. For a set-theoretic interpretation, we therefore require a set of all functions whose codomain contains that interpretation of *Int*, but this is not a set, but a proper class. However, if we start from a given set of untyped values, this problem can be circumvented. In particular, we can assume a set $\mathcal{D}$ whose elements correspond to computable values in a suitable sense (e.g. the solution to $D \cong D^D$ à la Scott, where continuity is taken as an essential feature of a computable function). On this set $\mathcal{D}$, we can give a natural interpretation of functions $\sigma \to \tau$ as follows:

$$[\![\sigma \to \tau]\!] = \{\, f \in \mathcal{D} \;:\; \forall x \in [\![\sigma]\!]\; f(x) \in [\![\tau]\!] \,\}$$

However, even this approach has short-comings, particularly when we model typed lambda calculus (extended with subtyping), where the type of a subterm depends on the type of its surrounding program context [Mit96]. An alternative approach is to define partial equivalence relations, i.e. equivalence relations on a subset of the given untyped universe. This is the approach used by e.g. Abadi et al for their denotational semantics [AC96]. In this thesis, we have emphasised the principle of direcursion as an important concept in models of typed object calculus based on self-application. We would therefore like to define coercion maps using direcursion, i.e. rather model subtyping as conversion functions following work by Breazu-Tannen, Coquand, Gunter, and Scedrov [BTCGS94] and Reynolds [Rey80]. In our context, this seems to boil down to defining dialgebras using the natural transformations associated to each pair of difunctors associated to a provable subtype judgement (usually projection functors). These dialgebras give rise to a direcursive map as we have showed in this thesis, but as coercion maps these are required to respect the operational semantics, which amounts to them being homomorphisms in two respects: they must commute with method invocations (i.e. be coalgebra homomorphisms), but also with method updates

(i.e. be algebra homomorphisms). The challenge is, then, to find a suitable generalisation of direcursion which is capable of defining maps with those properties (for all object types). More generally, one may wish to study coercion functions also for extended subsets of FPC, along the lines of Abadi and Fiore [AF96]. It also seems worthwhile to make comparisons with Viswanathan's investigations [Vis98] and also to Freyd's reduction of recursive types to inductive types [Fre90].

- **Program algebras for object-based programs.** There is a substantial body of work on program algebra for functional programming languages (see e.g. [BdM97]). It seems worthwhile to study classes of algorithms and object-oriented (or even object-based) design patterns [GHJV95] using similar techniques, for example by parameterising calculations with respect to (di)functors (i.e. object types) and use direcursion (and its variants) when reasoning with specifications/implementations of an object-based program. (Note that computational adequacy (Paper II) ensures that such reasoning will be meaningful at the level of operational semantics.) Rypacek et al [RBN06] has made some progress on formalising design patterns, and it would be interesting to extend the coverage of their work to the object-based paradigm.

- **Equational logic for object-based programs.** Abadi and Cardelli included an equational theory for object calculus in their treatise [AC96]. This theory arises by taking the compatible, symmetric, reflexive and transitive closure of the reduction relation, and also includes the usual rules for method update, method invocation, and object introduction. It would be very interesting to extend this equational theory with alternative introduction/elimination rules, giving a mixed induction/coinduction principle tailored for objects.

- **Polytypic object calculus.** Generic programming extensions have been developed for functional programming languages (for example [Jan00]), based on the idea of defining programs by induction on a predetermined class of datatype shapes/patterns (corresponding e.g. to polynomial functors). It would be interesting to also develop such polytypic programming languages starting from typed object calculus. Since direcursive maps involve two parameters ϕ and ψ, language support for inductive definitions of these with respect to the shape of difunctors is required. Rather than considering all possible difunctors in such inductive definitons, one could consider all difunctors that arise from the denotations of subtypes of a predetermined type,

thus guaranteeing a certain minimal interface. One would perhaps start by identifying some classes of object-based programs (or design patterns) that can be expressed generically by induction on the structure they use beyond that provided by a shared supertype. Some elementary candidates may be programs that compute over (single/double) linked lists.

- **Axiomatic models of object-based programming langauges.** Fiore and Plotkin [FP94] studied axiomatically a class of computationally adequate models of FPC. It seems interesting to carry out a similar development also for typed object calculus, while also identifying universal properties of object types rather than Freyd's property for recursive types. For this, it would presumably be worthwhile to identify the natural transformations induced by the equational theory following e.g. Pitts [Pit01].

Bibliography

[Aba07] Martín Abadi. Personal communication, August 2007.

[Abr90] Samson Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[AC94a] Martín Abadi and Luca Cardelli. A semantics of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science, Paris, France*, pages 332–341, Los Alamitos, CA, July 1994. IEEE.

[AC94b] Martín Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AC98] Roberto Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Number 46 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[Acz88] Peter Aczel. *Non-Well-Founded Sets*. CSLI Publications, 1988.

[AdB91] Pierre H. M. America and Frank S. de Boer. A proof theory for a sequential version of POOL. Technical Report CS-R9118, Centrum voor Wiskunde en Informatica (CWI), 1991.

[AdBKR89] P. America, J. de Bakker, J. N. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Inf. Comput.*, 83(2):152–205, 1989. Reprinted in [dBR92].

[AF96] Martín Abadi and Marcelo Fiore. Syntactic considerations on recursive types. In *Proceedings 11th Annual IEEE Symp. on Logic in Computer Science, LICS'96, New Brunswick, NJ, USA, 27–30 July 1996*, pages 242–252. IEEE Computer Society Press, 1996.

[AHIK00] Luca Aceto, Hans Hüttel, Anna Ingólfsdóttir, and Josva Kleist. Relating semantic models for the object calculus. *Theoretical Computer Science*, 230(1-2):258, 2000.

[AJ94] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994. Revised edition available from the authors.

[AL91] A. Asperti and G. Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*. Foundations of Computing Series. MIT Press, 1991.

[AL97] Martín Abadi and Rustan Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214, pages 682–696. Springer-Verlag, 1997.

[AP90] Martín Abadi and Gordon Plotkin. A per model of polymorphism and recursive types. In J. Mitchell, editor, *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 355–365, Philadelphia, 1990. IEEE Computer Society Press.

[AR88] Pierre H. M. America and Jan J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. In *Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pages 254–288. Springer-Verlag, 1988.

[AR89] Pierre America and Jan J.M.M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, 39(3):343–375, 1989. Journal version of [AR88], technical report CS-R8709 at CWI, February 1987, and reprinted in [dBR92].

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. Revised edition.

[Bar03] Falk Bartels. Generalised coinduction. *Matematical Structures in Computer Science*, 13(2):321–348, 2003.

[Bar04] Falk Bartels. *On Generalised Coindiction and Probabilistic Specification Formats: Distributive laws in coalgebraic modelling*. PhD thesis, Center for Mathematics and Computer Science (CWI), Amsterdam, 2004.

[BBL96] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS, International Symposium of Mathematical Foundation of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 218–229. Springer Verlag, 1996.

[BBS04] Andrej Bauer, Lars Birkedahl, and Dana S. Scott. Equilogical spaces. *Theoretical Computer Science*, 315:35–59, 2004.

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[BdM97] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[Bec69] John Beck. Distributive laws. In B. Eckman, editor, *Seminar on Triples and Categorical Homology Theory*, volume 80 of *Lecture Notes in Mathematics*, pages 119–140. Springer-Verlag, 1969.

[Ber93] Ulrich Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60:91–117, 1993.

[Ber04] Ulrich Berger. An abstract strong normalization theorem. In *Computer Science Logic: 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005. Proceedings*, volume 3634 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, 1987.

[Bir89] Richard S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI Series F*, pages 151–216. Springer-Verlag, 1989.

[BJJM99] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *Lecture Notes in Computer Science*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[BL95] Viviana Bono and Luigi Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL, International Conference of Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer Verlag, 1995.

[BM96] John Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*, volume 60 of *CSLI Lecture Notes*. University of Chicago Press, 1996.

[Bor94] Francis Borceux. *Handbook of Categorical Algebra*. Number 50 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994. Three volumes.

[Bou04] Gérard Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14(3):263–315, 2004.

[BTCGS94] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. In *[GM94]*, pages 197–245. MIT Press, 1994.

[BW85] Michael Barr and Charles Wells. *Toposes, Triples, and Theories*. Number 278 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985. Revised September 2000, and also republished in Reprints in Theory and Applications of Categories, No. 12 (2005) pp. 1-287.

[BW99] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Les publications Centre de recherches mathématiques, 1999. Third edition (previous editions published by Prentice Hall).

[Car84] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, pages 51–67. Springer-Verlag, 1984. Lecture Notes in Computer Science, volume 173.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. Revised version of [Car84].

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, University of Calgary, June 1992.

[CHL03] Daniela Cancila, Furio Honsell, and Marina Lenisa. Generalized coiteration schemata. *Electronical Notes in Computer Science.*, 82(1), 2003.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cie97] Krzysztof Ciesielski. *Set Theory for the Working Mathematician*. Cambridge University Press, 1997.

[Coo87] William R. Cook. A self-ish model of inheritance. Unpublished manuscript., 1987.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, Department of computer Science, Providence, Rhode Island, May 1989.

[CP89] William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Conference proceedings on Object-*

oriented programming systems, languages and applications, pages 433–443. ACM Press, 1989.

[Cro93] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.

[CS92] Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.

[dB91] F. S. de Boer. A proof system for the language POOL. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 124–150. Springer-Verlag, Berlin, Heidelberg, 1991.

[dBR92] J. W. de Bakker and J. J. M. M. Rutten, editors. *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*. World Scientific, Singapore, 1992.

[DD96] Jaco De Bakker and Erik De Vink. *Control Flow Semantics*. MIT Press, 1996.

[DHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proc. of ACM-SIGPLAN OOPSLA, International Symposium on Object Oriented, Programming, System, Languages and Applications*, pages 166–178. The ACM Press, 1998.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[EHR92] L. Egidi, F. Honsell, and S. Ronchi Della Rocca. Operational, denotational and logical descriptions: A case study. *Fundamenta Informaticae*, 16(2):149–169, February 1992.

[FH83] Marco Forti and Furio Honsell. Set theory with free construction principles. *Annali della Scuola Normale Superiore di Pisa - Classe di Scienze (IV)*, 10(3):493–522, 1983.

[FHM94] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[Fio96a] Marcelo P. Fiore. Axiomatic domain theory. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 59, 1996.

[Fio96b] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[Fio96c] Marcelo P. Fiore. A coinduction principle for recursive data types based on bisimulation. *Information and Computation*, 127(2):186–198, 1996.

[Fis96] Kathleen Fisher. *Type Systems for object-oriented programming languages*. PhD thesis, Stanford University, Stanford, CA, USA, August 1996. STAN-CS-TR-98-1602.

[FJM+96] M. P. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of EATCS*, 59:227–256, 1996.

[Fok94] Maarten M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[FP94] Marcelo Fiore and Gordon Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994*, pages 92–102. IEEE Computer Society Press, July 1994.

[Fre90] Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90*, pages 498–507. IEEE Computer Society Press, June 1990.

[Fre91] Peter J. Freyd. Algebraically complete categories. In *Proceedings 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104, 1991.

[Fre92] Peter J. Freyd. Remarks on algebraically compact categories. In Fourman, Johnstone, and Pitts, editors, *Workshop on Applications of*

Categories in Computer Science, Proceedings of the London Mathematical Society Symposium, number 177 in London Mathematical Society Lecture Note Series, pages 95–106. Cambridge University Press, 1992.

[FS90] Peter J. Freyd and Andre Scedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[GG05] Johan Glimming and Neil Ghani. Difunctorial semantics of object calculus. In *Electronic Notes in Theoretical Computer Science*, volume 138. Elsevier, November 2005. Proceedings of the Second Workshop on Object Oriented Developments (WOOD 2004).

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *3rd International Workshop on High-Level Concurrent Languages (HLCL '98)*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[GH05] Jeremy Gibbons and Graham Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, April-May 2005.

[GHA01] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? In *Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science*, volume 44 of *Electronic Notes in Theoretical Computer Science*, April 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gib93] Jeremy Gibbons. Upwards and downwards accumulations on trees. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1993.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[Gli07a] Johan Glimming. Computational soundness and adequacy of typed object calculi. TRITA-CSC-TCS 2007:2, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, October 2007.

[Gli07b] Johan Glimming. Parametric (co)iteration vs. primitive direcursion. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007, Bergen, Norway, August 20-24, 2007, Proceedings*, volume 4624 of *Lecture Notes in Computer Science*, 2007.

[GM94] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, Cambridge, MA, 1994.

[Gol96] Derek Goldrei. *Classic Set Theory for Guided Independent Study*. Chapman and Hall, 1996.

[Gor94] Andrew D. Gordon. A tutorial on co-induction and functional programming. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Functional Programming*, pages 78–95. Springer, Berlin, Heidelberg, 1994.

[Gor98] Andrew D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 9–54. Cambridge University Press, 1998.

[GP97] Andrew D. Gordon and Andrew M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997.

[GR96a] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 386–395. ACM Press, 1996.

[GR96b] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. Technical Report 386, Computer Laboratory, University of Cambridge, January 1996.

[GS90] Carl A. Gunter and Dana S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. North Holland, 1990.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[GVY04] Neil Ghani, Bjorn Victor, and Kidane Yemane. Relationally staged computation in the π-calculus. In *Procedings of CMCS 2004*, volume 106 of *ENTCS*, pages 105–120. 2004.

[Hag93] Tatsuya Hagino. A categorical programming language. In M. Takeichi, editor, *Advances in Software Science and Technology*, volume 4, pages 111–135. Academic Press, 1993.

[Hal85] Lars Hallnäs. Approximations and descriptions of non-well-founed sets. Preprint, Department of Philosophy, University of Stockholm, 1985.

[Hal90] Lars Hallnäs. On the syntax of infinite objects: an extension of Martin-Löf's theory of expressions. In *COLOG-88: Proceedings of the International Conference on Computer Logic, Tallinn, USSR, December 12–16, 1988*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Chris Hankin, editor, *European Symposium on Programming*, volume 1381 of *Lecture Notes of Computer Science*, pages 105–121. Springer-Verlag, 1998.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[HL99] Furio Honsell and Marina Lenisa. Coinductive characterizations of applicative structures. *Mathematical Structures in Computer Science*, 9(4):403–435, 1999.

[HL00] Furio Honsell and Marina Lenisa. Coalgebraic coinduction in (hyper)set-theoretic categories. *Electronic Notes in Theoretical Computer Science*, 33, 2000.

[Hoo96] Paul F. Hoogendijk. *A generic theory of datatypes*. Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1996.

[How96] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[Hyl82] J.M.E. Hyland. The effective topos. In A.S. Troelstra and D. Van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 165–216. North Holland Publishing Company, 1982.

[Hyl92] J.M.E. Hyland. First steps in synthetic domain theory. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory '90*, volume 1144 of *Lectures Notes in Mathematics*, pages 131–156, Como, 1992. Springer-Verlag.

[Jac98] Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. *Electronical Notes in Computer Science*, 11, 1998. Special issue on the workshop Coalgebraic Methods in Computer Science (CMCS 1998).

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology, 2000.

[JC94] C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems – ESOP '94*, Lecture Notes in Computer Science, pages 302–316. Springer-Verlag, 1994.

[Joh02] Peter T. Johnstone. *Sketches of an Elefant: A Topos Theory Compendium*, volume I,II. Oxford University Press, 2002.

[Kam88] Samuel N. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

[Kan79] Akira Kanda. Fully effective solutions of recursive domain equations. In *Mathematical Foundations of Computer Science 1979*, volume 74 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Kan81] Akira Kanda. Constructive category theory. In *Proceedings on Mathematical Foundations of Computer Science*, volume 118, pages 563–577. Springer-Verlag, 1981.

[Kel82] Max Kelly. *Basic Concepts of Enriched Category Theory*. Number 64 in London Mathematical Society Lecture Notes. Cambridge University Press, 1982. Republished in Reprints in Theory and Applications of Categories, No. 10, 2005.

[Kle00] Josva Kleist. *Reasoning about objects using process calculus techniques*. PhD thesis, Aalborg University, 2000.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[Lam80] Joachim Lambek. From lambda calculus to cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 376–402. Academic Press, 1980.

[Law64] Bill Lawvere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences of the United States of America*, 52:1506–1511, 1964.

[Law66] Bill Lawvere. The category of categories as a foundation for mathematics. In *Proceedings of the Conference on Categorical Algebra*, pages 1–20. Springer-Verlag, 1966.

[Law69] Bill Lawvere. Adjointness in foundations. *Dialectica*, 23:281–296, 1969.

[Len96] Marina Lenisa. Final semantics for a higher order concurrent language. In Hélène Kirchner, editor, *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linköping, Sweden, April, 22-24, 1996, Proceedings*, volume 1059 of *Lecture Notes in Computer Science*, pages 102–118. Springer-Verlag, 1996.

[Len98] Marina Lenisa. *Themes in Final Semantics*. PhD thesis, Unversità di Pisa-Udine, 1998.

[Len99] Marina Lenisa. From set-theoretic coinduction to algebraic coinduction: Some results, some problems. In B. Jacobs and J. Rutten, editors, *Proceedings 2nd Int. Workshop on Coalgebraic Methods in Computer Science, CMCS'99, Amsterdam, The Netherlands, 20–21 March 1999*, volume 19. Elsevier, North-Holland, 1999.

[Lin89] Ingrid Lindström. A construction of non-well-founded sets within Martin-Löf type theory. *Journal of Symbolic Logic*, 54(1):57–64, 1989.

[Liq97] Luigi Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proc. of ECOOP, European Conference on Object Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, 1997.

[Liq98] Luigi Liquori. On object extension. In *Proceedings of ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 498–522, 1998.

[LM84] G. Longo and E. Moggi. Cartesian cloased categories of enumerations for effective type-structures. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[LR02] F. William Lawvere and Robert Rosebrugh. *Sets for Mathematics*. Cambridge University Press, 2002.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[LS97] Bill Lawvere and Stephen Schanuel. *Conceptual Mathematics: a First Introduction to Categories*. Cambridge University Press, 1997.

[LW06] John Longley and Nicholas Wolverson. Game semantics for object-oriented languages: a progress report. Presented at the Workshop on Games for Logic and Programming Languages II, 2006.

[Mac97] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1997. First edition appeared in 1971.

[Mal90] Grant R. Malcolm. *Algebraic data types and program transformation*. Ph.D. thesis, Department of Computing Science, Groningen University, The Netherlands, 1990.

[McL92] Colin McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides*. Clarendon Press, 1992.

[Mee86] Lambert Meertens. Algorithmics – towards programming as a mathematical activity. In J. W. De Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings CWI Symposium on Mathematics and Computer Science*, number 1 in CWI Monographs, pages 289–334. North-Holland, 1986.

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Mee96] Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Langages: Implementations Logics, and Programs, Proceedings of PLILP'96*, number 1140 in Lecture Notes of Computer Science, pages 1–16. Springer-Verlag, 1996.

[Mey82] Albert R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1982.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In S. Peyton-Jones, editor, *Functional Programming Languages and Computer Architecture*, pages 324–333. Association for Computing Machinery, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil99] Robin Milner. *Communication and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[Mit90] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–124. ACM Press, 1990.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[MJ95] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, 1995.

[Mø06] Rasmus Møgelberg. From parametric polymorphism to models of polymorphic FPC. Submitted for publication (LFCS, Univ. of Edinburgh), November 2006.

[Mog88] Eugenio Moggi. *The Partial Lambda-Calculus*. PhD thesis, University of Edinburgh, 1988.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Mos74] Yiannis N. Moschovakis. *Elementary induction on abstract structures*. North-Holland, 1974.

[Mos01] Lawrence S. Moss. Parametric corecursion. *Theoretical Computer Science*, 260(1–2):139–163, 2001.

[MPS84] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 165–174. ACM Press, 1984.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes: parts i and ii. *Information and Computation*, 100(1):1–77, 1992. Also appeared as Technical Report ECS-LFCS-89-85 and -86, University of Edinburgh, 1989.

[MT92] Karl Meinke and John V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science: Background - Mathematical Structures (Volume 1)*, pages 189–368. Oxford University Press, Inc., 1992.

[Oos95] Jaap Van Oosten. Basic category theory. Technical Report LS-95-1, BRICS, 1995.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Proceedings of the 5th GI-Conference onTheoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Par01] Alberto Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260(1–2):165–207, 2001.

[Par02] Alberto Pardo. Generic accumulations. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, volume 243 of *IFIP Conference Proceedings*, pages 49–78. Kluwer, 2002.

[Pie91] Benjamin C. Pierce. *Basic category theory for computer scientists*. Foundations of Computing Series. MIT Press, 1991.

[Pit94] Andrew M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[Pit96] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996.

[Pit01] Andrew M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5: Algebraic and Logical Structures*. Clarendon Press, Oxford, 2001.

[Plo72] Gordon D. Plotkin. A set-theoretical definition of application. Memorandum MIP-R-95, School of Artificial Intelligence, University of Edinburgh, 1972.

[Plo81] Gordon D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[Plo85] Gordon D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.

[Poi92] Axel Poigné. Basic category theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science: Background - Mathematical Structures (Volume 1)*, pages 413–640. Clarendon Press, Oxford, 1992.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.

[RB86] David E Rydeheard and Rod M Burstall. Monads and theories: a survey for computation. In Maurice Nivat and John C. Reynolds, editors, *Algebraic methods in semantics*, pages 575–605. Cambridge University Press, New York, NY, USA, 1986.

[RBN06] Ondrej Rypacek, Roland Backhouse, and Henrik Nilsson. Type-theoretic design patterns. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 13–22, New York, NY, USA, 2006. ACM Press.

[Red96] Uday Reddy. Global state considered unnecessary: An introduction to object-based semantics. *LISP and Symbolic Computation*, 9(1):7–76, 1996.

[Rei95] Horst Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.

[Rey80] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Gener-*

ation, Proceedings of a Workshop, pages 211–258. Springer-Verlag, 1980. Lecture Notes in Computer Science volume 94.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer Verlag, 1984.

[Ros94] Giuseppe Rosolini. Notes on synthetic domain theory. Available from the authors home page at DISI, Università di Genova, 1994.

[RS04] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. *Theorertical Computer Science*, 316(1):191–213, 2004.

[RS06] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.

[Rut90] Jan J. M. M. Rutten. Semantic correctness for a parallel object-oriented language. *SIAM Journal of Computing*, 19(2):341–383, 1990. Reprinted in [dBR92].

[Rut96] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. Technical Report CS-R9652, CWI, 1996.

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[San98a] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998. Earlier version published as Rapport de Recherche RR-3000, INRIA Sophia-Antipolis, August 1996.

[San98b] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[Sch05] Jan Schwinghammer. *Reasoning about Denotations of Recursive Objects*. Ph.D. thesis, University of Sussex, 2005.

[Sco60] Dana Scott. A different kind of model for set theory. Unpublished paper given at the 1960 Stanford Congress of Logic, Methodology and Philosophy of Science, 1960.

[Sco69] Dana Scott. Lattice-theoretic models of the λ–calculus. Princeton University, Princeton, N.J., 1969.

[Sco72] Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 97–136. Springer-Verlag, 1972.

[Sco73] Dana Scott. Lattice-theoretic models for various type-free calculi. In *Proceedings of the 4th International Congress for Logic Methodology, and the Philosophy of Science (Bucharest)*. North-Holland, 1973.

[SHLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical theory of domains*. Cambridge University Press, 1994.

[SHT95] V. Stoltenberg-Hansen and J. V. Tucker. Effective algebras. In *Handbook of logic in computer science (vol. 4): semantic modelling*, pages 357–526. Oxford University Press, Oxford, UK, 1995.

[Sim92] Alex K. Simpson. Recursive types in Kleisli categories. Unpublished paper (University of Edinburgh), August 1992.

[Sim93] Alex K. Simpson. A characterisation of the least-fixed-point operator by dinaturality. *Theoretical Computer Science*, 118(2):301–314, 1993.

[SP82] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[Spe93] Dwight L. Spencer. *Categorical Programming with Functorial Strength*. PhD thesis, The Oregon Graduate Institute of Science and Technology, January 1993.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab, 1971.

[Str02] Thomas Streicher. Message to the TYPES mailing list, October 2002.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[Tan02] Francis Hin-Lun Tang. *Towards feasible, machine-assisted verification of object-oriented programs*. PhD thesis, LFCS, University of Edinburgh, 2002.

[Tar55] Alfred Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tay91] Paul Taylor. The fixed point property in synthetic domain theory. In *6th Symp. on Logic in Computer Science*, pages 152–160. IEEE Computer Society Press, 1991.

[Tay99] Paul Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999.

[TP97] Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proceedings 12th Ann. IEEE Symp. on Logic in Computer Science, LICS'97, Warsaw, Poland, 29 June – 2 July 1997*, pages 280–291. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[TR98] Daniele Turi and Jan Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8(5):481–540, 1998.

[Tur96] Daniele Turi. *Functorial operational semantics and its denotational dual*. PhD thesis, CWI, 1996.

[UV99] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10(1):5–26, 1999.

[UVP01] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–390, Fall 2001.

[Ven00] Varmo Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, 2000.

[Vis98] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS), 1998)*. IEEE Computer Society, 1998.

[Wad76] Christopher P. Wadsworth. The relation between computational and denotational properties for scott's d_∞-models of the lambda calculus. *SIAM Journal of Computing*, 5(3), 1976.

[Wal92] R. F. C. Walters. An imperative language based on distributive categories. *Mathematical Structures in Computer Science*, 2(3):249–256, 1992.

[Wal95] David Walker. Objects in the pi calculus. *Information and Computation*, 116(2):253–271, 1995.

[Wan79] Mitchell Wand. Fixed point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.

[Wan94] Mitchell Wand. Type inference for objects with instance variables and inheritance. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, London, 1994. Originally appeared as Northeastern University College of Computer Science Technical Report NU-CCS-89-2, February, 1989.

[Wei02] Stephanie Weirich. *Programming With Types*. PhD thesis, Cornell University, August 2002.

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

[WW03] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 249–262, Uppsala, Sweden, August 2003. ACM SIGPLAN.

universitetsservice
Printed in Sweden 2007, www.us-ab.com