Preface

A general abstract theory for computation involving shared resources is presented. We develop the models of *sharing graphs*, also known as term graphs, in terms of both syntax and semantics.

According to the complexity of the permitted form of sharing, we consider four situations of sharing graphs. The simplest is first-order acyclic sharing graphs represented by let-syntax, and others are extensions with higher-order constructs (lambda calculi) and/or cyclic sharing (recursive letrec binding). For each of four settings, we provide the equational theory for representing the sharing graphs, and identify the class of categorical models which are shown to be sound and complete for the theory. The emphasis is put on the algebraic nature of sharing graphs, which leads us to the semantic account of them.

We describe the models in terms of the notions of symmetric monoidal categories and functors, additionally with symmetric monoidal adjunctions and traced monoidal categories for interpreting higher-order and cyclic features. The models studied here are closely related to structures known as notions of computation, as well as models for intuitionistic linear type theory. As an interesting implication of the latter observation, for the acyclic settings, we show that our calculi conservatively embed into linear type theory. The models for higher-order cyclic sharing are of particular interest as they support a generalized form of recursive computation, and we look at this case in detail, together with the connection with cyclic lambda calculi.

We demonstrate that our framework can accommodate Milner's *action calculi*, a proposed framework for general interactive computation, by showing that our calculi, enriched with suitable constructs for interpreting parameterized constants called controls, are equivalent to the closed fragments of action calculi and their higher-order/reflexive extensions. The dynamics, the computational counterpart of action calculi, is then understood as rewriting systems on our calculi, and interpreted as local preorders on our models.

Preface to the Present Edition

This book contains the author's PhD thesis written under the supervision of Rod Burstall (first supervisor), Philippa Gardner and John Power (second supervisors) at Laboratory for Foundations of Computer Science, University of Edinburgh. The thesis was examined by Martin Hyland (Cambridge) and Alex Simpson (Edinburgh). Except for correcting minor mistakes and updating the bibliographic information, the text agrees with the examined version of the thesis.

Some parts of the book have been published elsewhere in [13, 35, 38]. Since the examination of the thesis, a number of works related to this research have appeared. I

take this opportunity to mention some of them.

- An independent work by Corradini and Gadducci [25] used essentially the same categorical structure described in Chapter 3 for modeling acyclic graph rewriting systems (with Cat-enrichment rather than Preord-enrichment). Miyoshi [70] translated the results in Chapter 6 to their setting and reformulated the cyclic sharing theories as a rewriting logic.

- While the model construction techniques in Chapter 5 show the conservativity of syntactic translations, further techniques for showing the fullness (or full completeness) of the translations have been developed by the author, as reported in [39].

- A direction progressing rapidly is the investigation of traced monoidal categories as a foundation of recursive computation, as claimed in Chapter 7. Some fundamental issues on traced monoidal categories are studied in Abramsky, Blute and Panangaden [4] and Blute, Cockett and Seely [23]; the latter contains a fixpoint theorem related to those in Chapter 7. As an interesting case study, Ryu Hasegawa [40] related the fixpoint operator in a model of (typed and untyped) lambda calculus and the Lagrange-Goodman inversion formula in enumerative combinatorics in terms of trace. The relation to axiomatic domain theory has been studied by Plotkin and Simpson [74].

- In Chapter 9 the possibility of developing the premonoidal variant of the sharing theories and their models was suggested. Related to this, Jeffrey [46] has introduced a semantics of the graphically-presented imperative programs based on premonoidal categories. In that setting, he also modeled recursion using trace.

Acknowledgements

I want to express my heartfelt thanks to my supervisors; I can never thank them enough.

Rod Burstall, my first supervisor, always helped me to think constructively and positively, especially at difficult moments throughout my PhD study in Edinburgh. I will never forget a meeting with Rod when I had a bad hangover – there I got an essential inspiration in deciding my research direction.

Philippa Gardner, my second supervisor during the second year, always gave me enthusiastic encouragement, and I benefited from countless delightful (often overheated) discussions with her.

John Power has always been an important intellectual source and often a mentor for me during these three years, and he became my second supervisor after Philippa moved to Cambridge. Without his generous and much needed support, this thesis would probably never have been written in this form.

At a stimulating place like LFCS, even a brief chat often meant a lot to me. I am grateful to people who influenced me in various forms, especially to Andrew Barber, Ewen Denney, Marcelo Fiore, Alex Mifsud, Robin Milner, Gordon Plotkin and Alex

Simpson. In particular, Chapter 5 and Chapter 8 refer to joint work with Andrew, Philippa and Gordon.

I want to thank Martin Hyland for helpful discussions on traced monoidal categories as well as for his warm encouragement. Thanks are also due to Zena Ariola and Stefan Blom, for e-mail communications on cyclic lambda calculi and related topics.

And, above all, many, many thanks go to my cheerful, lovely, friends.

This work was partly supported by an Oversea Research Student award.

Contents

1
Introduction

1.1 Computation Involving Shared Resources

The notion of *sharing* has appeared on various occasions in computer science, either explicitly or implicitly. The idea is simple: instead of giving computational resources (processes, memories etc) to each client, a single resource can be shared by multiple clients.

In general, this kind of replacement may change the nature of the involved computation significantly. For instance, if the resource we are concerned with requires heavy computation or a large memory, sharing becomes an essential technique for saving both time and space needed for the computation. Many implementations of pure functional programming languages are based on this observation – avoiding unnecessary duplication of subcomputation is crucial for achieving efficient functional computation.

However, sharing is not just about the efficiency. If the resource involves some computation with side effects, say non-determinism or imperative states, the sharing of such a resource may change not just the amount of computation but also the result of computation. In such impure cases, the distinction between duplicated resources and shared resources must be made more carefully, and this makes it difficult, or at least non-trivial, to reason about general computation involving shared resources.

Furthermore, sharing can naturally be used for implementing *cyclic* (self-referential) data structures, which have been used for implementing recursive computation efficiently. The expressive power obtained by cyclic sharing is enormous, but dealing with cyclic structures is far more difficult than dealing with just acyclic ones. For instance, there are various practical ways of encoding recursive computation using cyclic sharing, but, to the best of our knowledge, there has been no formal comparison between them.

This thesis is devoted to giving a theory for describing and reasoning about such computation with sharing. The weight is put on the study of the *classes* of models of sharing, rather than individual specific models, in a desire to extract a generic account for sharing.

1.2 Sharing Graphs as Models of Sharing

Sharing for Efficiency

No programmer would be happy to write an expression like

```
... (factorial(100) + 123) * factorial(100) ...
```

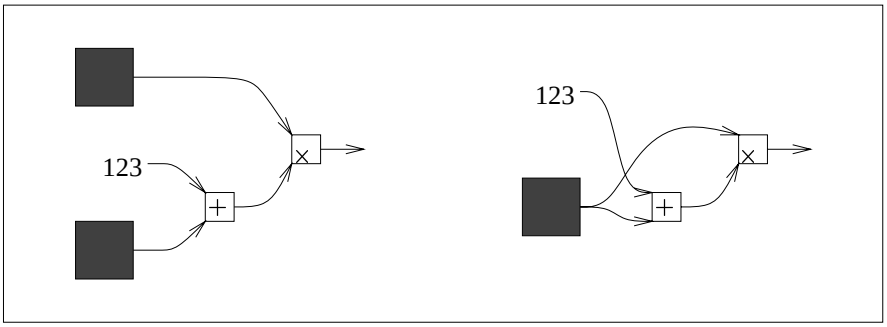
containing two identical subexpressions `factorial(100)` – not just because it makes the program messy but because it does suggest a duplication of very heavy computation (here we suppose that the program `factorial(100)` calculates the factorial of 100, which in many cases results in an overflow). The former reason may be very important from the view of software engineering where readability and reusability of programs are essential, but it is not a matter to be discussed now. Here we shall stick to the second point - efficiency. Many programmers should agree to rewrite the expression above as

```
let x = factorial(100) in ... (x + 123) * x ...
```

The intention is that, we avoid calculating `factorial(100)` twice by *sharing* the result of this computation, without changing the result of computation. The `let` syntax indicates that `factorial(100)` is a shared resource with a name `x` which are later referred (used) at two places in the program.

But actually this is not just a matter for programmers, but more essentially the problem of the implementor of the programming language. Though the two examples above are supposed to return the same result, hence are extensionally equivalent, they are "intensionally" different because the amount of the involved computation is different; implementors must realize some semantic models in which such these two have distinct denotations – they may not be models for programmers (who just care about the results) but are models for implementors (who care about the actual computational steps behind the results).

Graph rewriting theory – the theory of *sharing graphs* (*term graphs*) and rewriting systems on them – has been recognized as a canonical and practically useful instance of such models for implementors [15, 84]. The idea is to use graphs for representing the sharing relations of resources and realize computation on them as rewriting systems. For instance, the situation above can be explained simply by the graphical representation of the expressions, as



The left tree corresponds to the original unshared version, whereas the right graph is for the "refined" version with sharing of a resource. The actual computation is modeled by rewriting, i.e. local replacement of subgraphs. Obviously the left one requires more computation (rewriting steps) because of the duplicated resource (subgraph).

Impure Cases: Sharing as a Programming Technique

Consider a language with a non-deterministic construct `zero_or_one` which returns 0 or 1 at random. As before, we shall use the let-syntax for representing sharing. Then the following two programs obviously have different meanings.

```
zero_or_one + zero_or_one

let x = zero_or_one in x + x
```

The former returns 0, 1 or 2, whereas the latter 0 or 2 (see Figure 1.1). In this case the shared resource is not *pure*; it contains a side-effect, thus should be better understood as a process in a concurrent language or an object in an object-oriented language. Similar things happen if we consider imperative languages with states. In such "impure" settings, introducing sharing may change the result of computation, hence changing the extensional (programmers') semantics of the language. Therefore sharing becomes an important feature of the programming language which programmers have to recognize as a programming technique; and actually most programmers of impure languages do, often explicitly when manipulating states, objects and memories.

Cyclic Sharing and Recursion

Circular phenomena have been a rich source of a wide range of intellectual investigations for long time – in science, technology, and even philosophy; see [16] for a survey and lots of examples. Computer science is not an exception. Sharing graph-based models have a natural advantage in representing cyclic data structures, and the most interesting and practical usage of such cyclic sharing is, of course, as the means of realizing recursive computation, which is one of the most important subjects in computer science. As already shown by Turner [87] in 70s, recursive computation can be efficiently implemented using self-referential (i.e. cyclic) terms. We come back this point later and explain in some detail – the analysis of recursive computation created from cyclic sharing is one of the central implications of this thesis.

1.3 Sharing Graphs and Their Presentation

As motivated above, we regard sharing graphs, or term graphs, as abstract representations of the sharing relation of resources. They can be seen as a special sort of directed graph in which nodes represent resources and links show the sharing, but perhaps are better understood as a generalization of the tree notations for terms – the name "term graphs" means the direct generalization of "term trees".

If there is no notion of sharing, it suffices to talk about just trees (terms) where subtrees (subterms) correspond to subcomputations. However, if we want to talk about sharing, trees are not sufficient, and we are naturally led to replace trees by a class of directed graphs. Now a subgraph may be referred from various places in the graph,

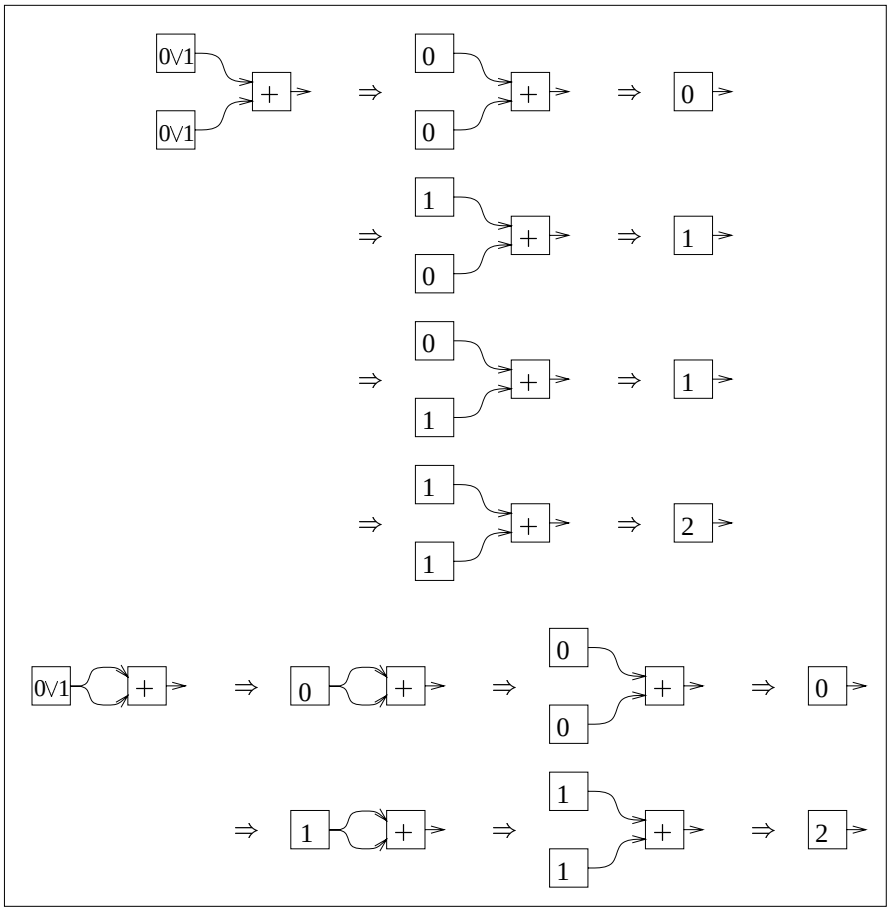


Figure 1.1: Sharing non-deterministic computation

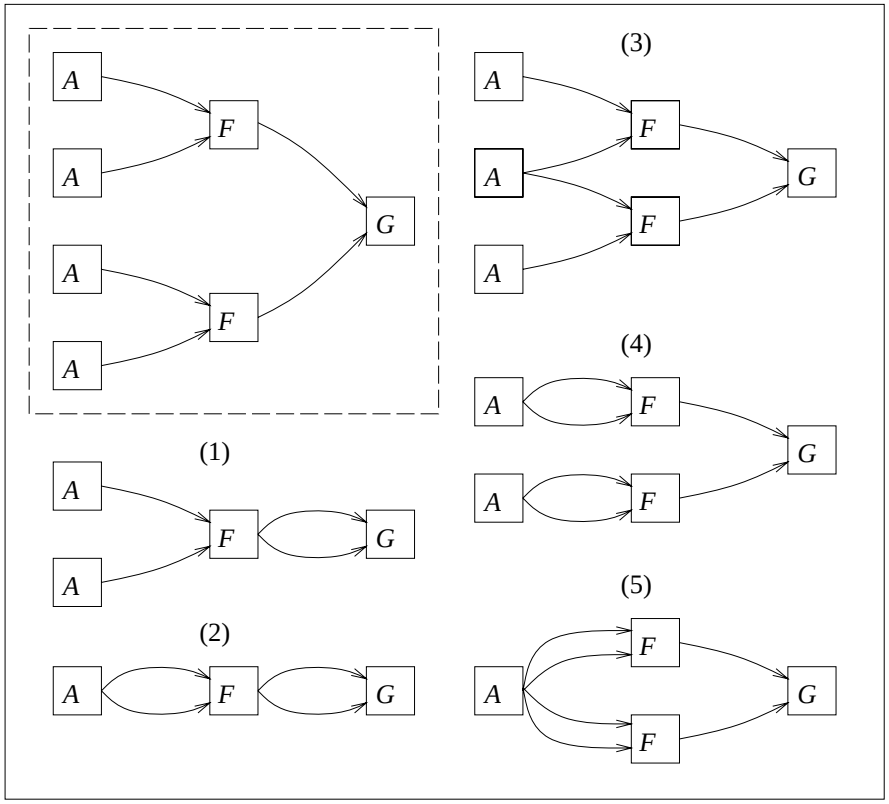


Figure 1.2: Various term graphs corresponding to a term

thus representing a shared resource. Figure 1.2 shows that there are various sharing graphs corresponding to a term $G(F(A,A),F(A,A))$. As mentioned earlier, the meaning of sharing changes depending on the computation concerned. If each node represents purely functional computation, the difference between these sharing graphs is just about the amount of computation. The final answer will be the same, but the sharing graph (2) presents the optimal way to get the answer. On the other hand, if A is a process which returns 0 or 1 non-deterministically and F and G calculate the sum of arguments, then the original term presents a computation which returns 0, 1, 2, 3 or 4, while (1) and (4) return 0, 2 or 4, whereas (2) and (5) returns just 0 or 2. (3) returns 0, 1, 2, 3 or 4 as the original term, but the probability would be changed.

Allowing cyclic bindings, sharing graphs get further flexibility. Let us look at some instances of cyclic sharing graphs (Figure 1.3). (1) and (2) present the simplest situations of cyclic sharing. In (1), the resource I refers to itself; (2) may seem odd as it does not involve any resource, but such a "self-referential pointer" or "trivial cycle" can occur even in a realistic situation. (3) is similar to (1), except that it has one additional input. A more sophisticated example is (4) where F and G mutually refer

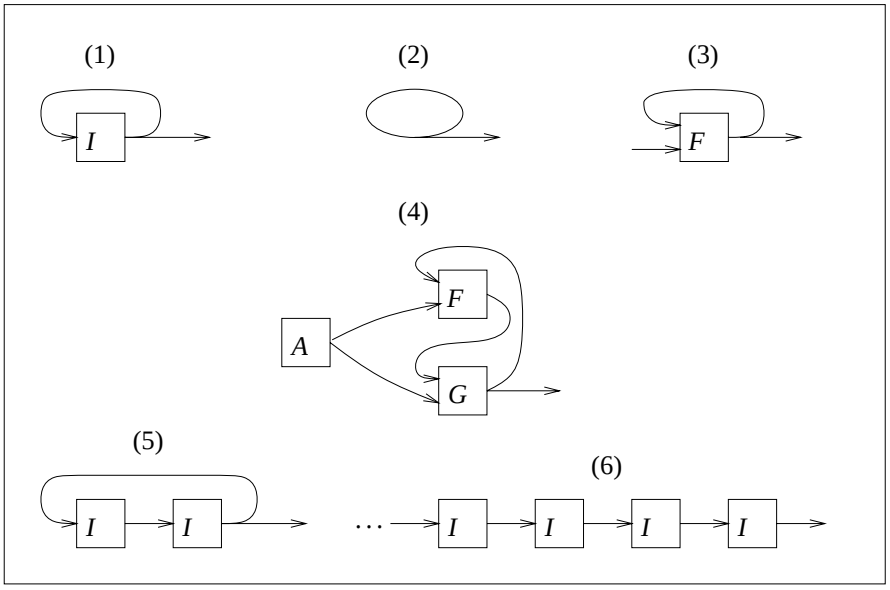


Figure 1.3: Cyclic sharing graphs

each other. (1) and (5) have the same tree-unwinding $I(I(I(I(\ldots))))$ as (6), but again it depends on the situation whether we should identify the meaning of (1) and (5) with (6).

Now we turn our attention to how to present term graphs concisely. Defining them as directed graphs, as we will do later, is not very informative; sharing graphs have more structural and algebraic properties than general directed graphs do, and we wish to capture this nature. A first hint comes from the observation above that sharing graphs can be obtained by enriching traditional terms (trees) with constructs for acyclic or cyclic sharing. Our programming example already suggests a convenient syntax for them - the let (letrec) blocks.

Actually similar notions have appeared in many places for presenting similar kind of (possibly circular) dependency relations. There are various versions of *systems of equations* for describing "non-well-founded sets" [5, 16] like

$$\begin{array}{lcl} x & = & \{y\} \\ y & = & \{x,z\} \\ z & = & \{x\} \end{array}$$

(The anti-foundation axiom states that this kind of system has a unique solution.) Similarly it is common to present a state transition system like finite state automata, and also concurrent processes, e.g. [63], by a system of equation

$$Clock \quad = \quad tick.Clock + break.Stuckclock$$

Yet another popular instance is the description of inductive (or recursive) types: for instance the type T of finite branching finite trees can be represented as a solution of

a system of equations

$$\begin{array}{lcl} T & = & F \\ F & = & 1 + T \times F \end{array}$$

(The terms can be generated by BNF

$$\begin{array}{lcl} t & ::= & \mathtt{span}(f) \\ f & ::= & \mathtt{nil} \mid \mathtt{cons}(t, f)\ .) \end{array}$$

These systems of equations have natural graph presentations, though it is possible that two different systems may describe the identical graph[1]. So there should be an equational theory on these systems which is sound and complete with respect to the graph interpretation.

We give such an axiomatization on our terms with the let/letrec blocks (which are of course an instance of systems of equations). Such notation has an advantage in allowing us equational and inductive structural reasoning about sharing graphs. We inductively construct (the presentations of) sharing graphs from variables (pointer names), function symbols (resources) and systems of equations. Thus, as the traditional algebraic theories for terms, we give equational theories for sharing graphs in terms of systems of equations for which we use the let/letrec-binding syntax. For instance, the acyclic sharing graphs in the first example can be presented as

(1) $\mathsf{let}\ x = F(A,A)\ \mathsf{in}\ G(x,x)$
(2) $\mathsf{let}\ y = A\ \mathsf{in}\ \mathsf{let}\ x = F(y,y)\ \mathsf{in}\ G(x,x)$
(3) $\mathsf{let}\ y = A\ \mathsf{in}\ G(F(A,y),F(y,A))$
(4) $\mathsf{let}\ y = A\ \mathsf{in}\ \mathsf{let}\ y' = A\ \mathsf{in}\ \mathsf{in}\ G(F(y,y),F(y',y'))$
(5) $\mathsf{let}\ y = A\ \mathsf{in}\ G(F(y,y),F(y,y))$

As noted above, two different terms can represent the same graph; for instance, (3) can be presented as $\mathsf{let}\ y = A\ \mathsf{in}\ \mathsf{let}\ x = F(A,y)\ \mathsf{in}\ G(x,F(y,A))$, and our equational theory guarantees that this is equal to $\mathsf{let}\ y = A\ \mathsf{in}\ G(F(A,y),F(y,A))$. Similarly, the (finite) cyclic sharing graphs in the second picture correspond to

(1) $\mathsf{letrec}\ x = I(x)\ \mathsf{in}\ x$
(2) $\mathsf{letrec}\ x = x\ \mathsf{in}\ x$
(3) $\mathsf{letrec}\ x = F(y,x)\ \mathsf{in}\ x$
(the free variable y represents the unspecified input node)
(4) $\mathsf{letrec}\ x = A,\ y = F(x,z),\ z = G(x,y)\ \mathsf{in}\ z$
(5) $\mathsf{letrec}\ x = I(I(x))\ \mathsf{in}\ x$

A simple discipline of typing is naturally given, as for traditional algebraic theories, in which any sharing graph is equipped with its input and output types (sorts). This allows us to construct graphs by well-typed composition inductively.

[1] Actually, for these examples, we usually work up to some stronger equivalences than that of graphs; for instance two systems are often equated if they correspond to the same infinite unwinding, equivalently if they are "bisimular". But here we do not presuppose such specific semantic interpretations, and just compare the graphs concerned themselves.

Moreover, the rewriting rules on sharing graphs are easily presented on such an equational formulation, in similar manner to the usual term rewriting rules on algebraic theories. The only difference is that in each rewriting step we replace a subgraph by another (with the same typing), instead of replacing a subterm by another.

Such advantages of this style of presentation have already been emphasized and studied by Klop, Ariola and others in the context of graph rewriting theory [7, 11]. In this thesis we basically follow their ideas, but use them freely in a more general and semantic (algebraic) context. The merit of the equational presentation becomes clearer in developing the semantic counterpart of sharing graphs, as explained below.

1.4 Categorical Models for Sharing Graphs

Traditionally, the semantic account of sharing graphs has been given in specific models, most importantly as tree unwindings where two sharing graphs are identified if they represent the same (possibly infinite) tree. Such a semantics stands out if we use sharing graphs for representing efficient implementations of pure functional computation. In this thesis, however, we take a different starting point, for the following reasons.

1. We wish to keep as many choices of semantic models as possible, so that we can interpret various (impure) forms of computation flexibly. For instance, if we want to take non-determinism into account, the infinite tree unwinding semantics is inconsistent. Rather than starting from specific models and trying to interpret actual computation in them, we axiomatize the properties needed by the models of sharing, and then find intended models.

2. We wish to talk about the *class* of models. This enables us to prove general results on all models at once, and also to classify models in a natural manner. For instance, we will give relations between our sharing graphs and intuitionistic linear logic by comparing the classes of models.

For describing the classes of models of sharing graphs, we find category-theoretical languages useful. The canonical examples of the use of category theory in this direction are the correspondence between algebraic theories and cartesian categories (categories with finite products), as well as that between the simply typed lambda calculus and cartesian closed categories. Let us summarize these "standard" *categorical type theory* correspondence as below; to make the connection with cyclic sharing, we include the treatment of recursion in our picture (Figure 1.4). Following Lawvere [58], we give models of an algebraic theory by a *finite product preserving functor* from the *classifying category* (term model) of the algebraic theory into a cartesian category. Each function symbol F with arity $((\sigma_1, \ldots, \sigma_n), \tau)$ is interpreted as an morphism $[\![F]\!] : [\![\sigma_1]\!] \times \ldots \times [\![\sigma_n]\!] \to [\![\tau]\!]$ in the target cartesian category, where $[\![\sigma_i]\!]$, $[\![\tau]\!]$ present the objects associated with each sort σ_i, τ in the algebraic theory, and $\times$ is the (chosen) cartesian product. The interpretation is then inductively extended to all expressions (terms) in the algebraic theory – it determines a finite product preserving functor from the classifying category into the model category if and only if it satisfies the soundness property: if two expressions are provably equal in the theory, then

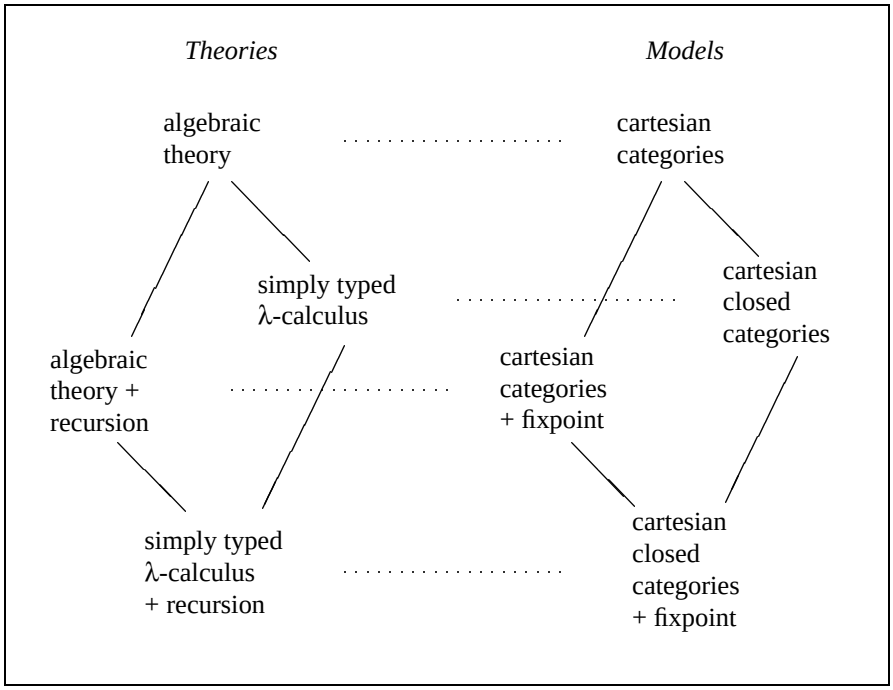


Figure 1.4: Algebraic theories and their models

their interpretation in the model is the same morphism. This is the basic picture of the theory-model correspondence in categorical type theory. A detailed account can be found, for instance, in [26].

This basic setting can be enriched with higher-order features, as well as recursive computation. For shifting to the higher-order extension, we require the existence of exponents, thus assume that the functor $(-) \times X$ has a right adjoint for each object X in the model category. Therefore we are led to the notion of *cartesian closed categories*, and again we get the theory-model correspondence between simply typed lambda theories and cartesian closed categories [26, 56] (this time the semantic interpretations are given as cartesian closed functors).

For recursion, the standard way is to assume a construction on the model cartesian (closed) category, called a *(parameterized) fixed point operator*

$$\frac{f : A \times X \to X}{f^\dagger : A \to X}$$

which is subject to the condition that $\langle id_A, f^\dagger \rangle ; f = f^\dagger$ (to be more precise, we assume that this construction is natural in A, so that the model is sound for the interpretation of substitutions). In the standard notation for a recursion operator on algebraic theories, this corresponds to

$$\frac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash \mu x.M : \sigma}$$

with the fix point equation $\mu x.M = M[\mu x.M/x]$. Many concrete examples of such categories are found in domain theory, where cartesian closedness and existence of fixed point operators are fundamental requirements for giving the denotational semantics of programming languages.

The main technical development in this thesis is to give, for sharing graphs, a precise analog of this standard categorical type theory. The equational theory presentation of sharing graphs via the let (letrec)-syntax is already very close to the standard algebraic theories, and it is natural to expect that there is a similar theory-model correspondence for sharing graphs.

The essential change is that, instead of cartesian categories, we take identity-on-objects, strict symmetric monoidal functors from cartesian categories to symmetric monoidal categories as the basic setting for interpreting the sharing graphs. Intuitively, the domain cartesian category is used for modeling the non-linear nature of sharing graphs – pointer names, or links, and also copyable-values (if they exist), are duplicated or discarded freely, hence will be interpreted in the cartesian category as we do for algebraic theories. On the other hand, the codomain symmetric monoidal category is for interpreting linear entities in sharing graphs; since we do not duplicate or discard the shared resources which are expensive or contain some side effect, they must be treated linearly. (The reader familiar with linear logic [36] may informally understand this by the analogy with the logical connectives & and $\otimes$ of linear logic; later we will give the precise connection between our models of sharing and those of propositional intuitionistic linear logic.) The strict functor between them is to relate these non-linear and linear natures. In short, the essence of models of sharing lies in the separation of non-linear and linear features which live at the same time in the

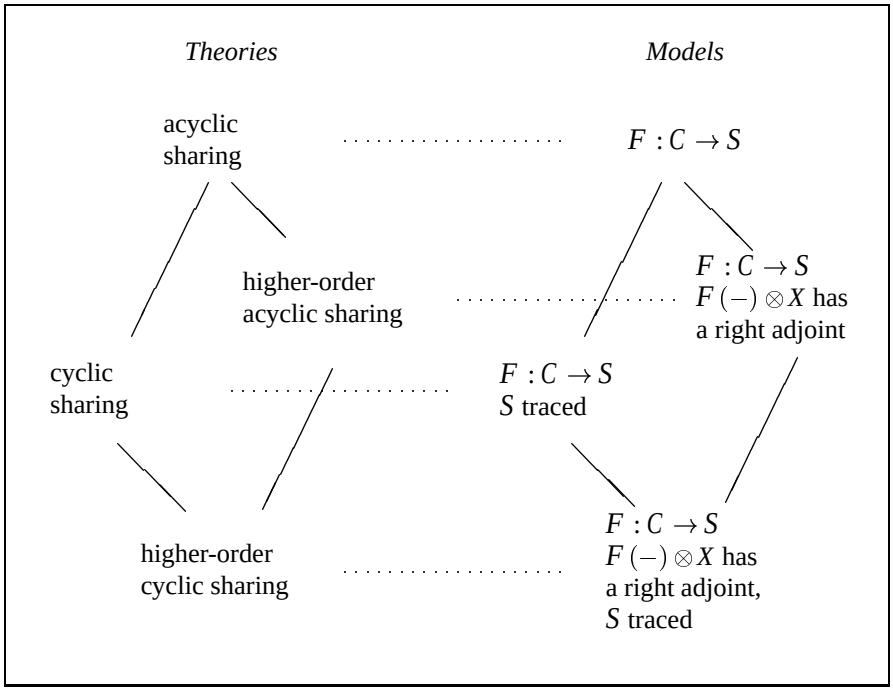


Figure 1.5: Sharing theories and their models

notion of sharing. Now we shall give our picture of the theory-model correspondence for sharing graphs (Figure 1.5). By $F : C \to S$, we mean an identity-on-objects strict symmetric monoidal functor F from a cartesian category C to a symmetric monoidal category S.

For interpreting higher-order features, we additionally require that $F(-) \otimes X$ has a right adjoint for each object X; this is the precise analog of cartesian closed categories for our setting. For interpreting cyclic sharing, we need a relatively new concept from category theory – *traced monoidal categories* [50]. Intuitively, a traced symmetric monoidal category is a symmetric monoidal category equipped with a construct for "feedback", called a *trace*:

$$\frac{f : A \otimes X \to B \otimes X}{Tr^X_{A,B}(f) : A \to B}$$

It would be helpful to understand that, in $Tr^X(f)$, f's output X is feedbacked, or linked, to f's input X. The formal axiomatization for a trace will be recalled later; we will see that it precisely corresponds to the equivalence on cyclic graphs, and the theory-model correspondence will be extended to the cyclic settings comfortably by assuming that the symmetric monoidal category S is traced.

The rewriting theories on sharing graphs are then simply modeled by the local-preorders on the symmetric monoidal category S of our models. Some graph rewriting systems, especially the *equational term graph rewriting* by Klop and Ariola, are close

to our theories and their semantic models.

Note that if we restrict our attention to the case that C and S are the same cartesian category and F is the identity functor, then we recover the standard categorical type theory as sketched before (a connection between traces and fixed point operators will be established later).

1.5 Relating Models

To demonstrate the advantage of our generic approach, we shall relate some known systems and ours by comparing their classes of models. Many people have pointed out that term graphs have some similarity with Girard's *linear logic* [36], in their resource-sensitive natures. Also it has been pointed that Moggi's computational lambda calculus [71] looks like higher-order graph rewriting systems. We give some formal accounts to these intuitive understandings, by first relating the classes of models, and then relating the theories as a corollary.

A model of propositional intuitionistic linear logic may be described as a symmetric monoidal adjunction between a cartesian closed category and a symmetric monoidal closed category [12, 18, 20]. It is easily seen that such a structure is essentially a special case of the structures we have for interpreting acyclic sharing graphs, as sketched above. Thus there is a sound translation from the equational theory of sharing graphs into that of intuitionistic linear type theory. But we can say more: this translation is conservative, thus a linear type theory is seen as a conservative extension of the theory of sharing graphs. To prove this, we use the standard model construction technique from category theory (Yoneda construction as the free symmetric monoidal cocompletion [44]).

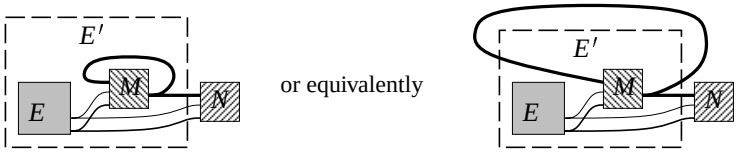
The connection with Moggi's work [71, 72] is much more straightforward. The models for acyclic higher-order sharing will be shown to be essentially the same as his models for computational lambda calculus, with an assumption that the associated monad has a commutative strength. As a special instance of the theory developed by Power and Robinson [77, 76], we describe this comparison.

1.6 Recursion from Cyclic Sharing

One of the traditional methods of interpreting a recursive program in a semantic domain is to use the least fixed-point of continuous functions. However, as already mentioned, in the real implementations of programming languages, we often use some kind of shared cyclic structure for expressing recursive environments efficiently. For instance, the following is a call-by-name operational semantics of the recursive call, in which free x may appear in M and N. We write $E \vdash M \Downarrow V$ for saying that evaluating a program M under an environment E results a value V; in call-by-name strategy an environment assigns a free variable to a pair consisting of an environment and a program.

$$\frac{E' \vdash N \Downarrow V \quad \text{where } E' = E \cup \{x \mapsto (E', M)\}}{E \vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ N \Downarrow V}$$

That is, evaluating a recursive program letrec $x = M$ in N under an environment E amounts to evaluating the subprogram N under a cyclic environment E' which references itself. One may see that it is reasonable and efficient to implement the recursive (self-referential) environment E' as a cyclic data structure as below.



Also it is known that if we implement a programming language using the technique of sharing, the use of the fixed point combinator causes some unexpected duplication of resources [9, 57]; it is more efficient to get recursion by cycles than by the fixed point combinator in such a setting. This fact suggests that there is a gap between the traditional approach based on fixed points and cyclically created recursion.

Our semantic models for higher-order cyclic sharing turn out to be the setting for studying recursive computation created by such a cyclic data structure, more specifically cyclic lambda graphs [10, 8]. We claim that our new models are natural objects for the study of recursive computation because they unify several aspects on recursion in just one semantic framework. The leading examples are

- the *graphical (syntactical) interpretation* of recursive programs by cyclic data structures motivated as above,
- the *domain-theoretic interpretation* in which the meaning of a recursive program letrec $x = F(x)$ in x is given by the least fixed point $\bigcup_n F^n(\bot)$, and
- the *non-deterministic interpretation* where the program letrec $x = F(x)$ in x is interpreted by $\{x \mid x = F(x)\}$, the set of all possible solutions of the equation $x = F(x)$.

Each of them has its own strong tradition in computer science. However, to our knowledge, this is the first attempt to give a uniform account on these well-known, but less-related, interpretations. Moreover, our higher-order cyclic sharing theories and cyclic lambda calculi serve as a uniform language for them.

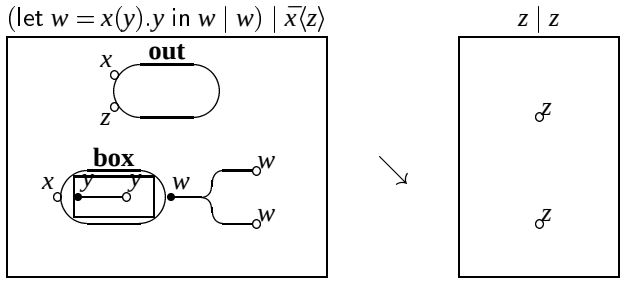
1.7 Action Calculi as Graph Rewriting

Finally we show that our framework can accommodate Milner's *action calculi* [68], a proposed framework for general interactive computation, by showing that our sharing theories, enriched with suitable constructs for interpreting parameterized constants called controls, are equivalent to the closed fragments of action calculi [34, 75] and their higher-order/reflexive extensions [66, 67, 61].

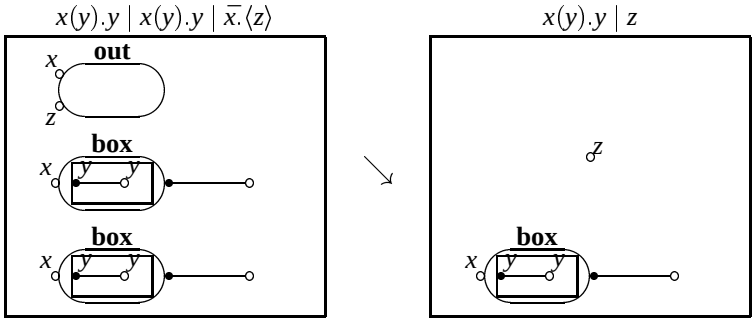
The dynamics, the computational counterpart of action calculi, is then understood as rewriting systems on sharing theories, and interpreted as local preorders on our models. In this sense, we understand action calculi as generalized graph rewriting

systems – and regard the notion of sharing as one of the fundamental concepts of action calculi.

To demonstrate how sharing is used in action calculi, we shall consider two situations representable in the action calculus-version of the π-calculus [69, 64] as presented in [68] (see Chapter 8, Example 8.1.6).



We may regard this situation (not representable in the original π-caclulus!) as a broadcasting; there is an announcer $x(y).y$ who gets a message via a telephone number x and then broadcasts it; her/his program is monitored by two listeners $w|w$. Therefore the received message z is broadcast (duplicated) to the listeners. Compare this and the unshared version $x(y).y \mid x(y).y \mid \bar{x}\langle z\rangle$, where we have two persons who share the same telephone number x. So we don't know which person will receive the message z, and there are two possible reactions (in both cases the result is $x(y).y \mid z$, thus one person remains unchanged:



Further sophisticated and complicated examples will be available by allowing cyclic sharing (reflexion) and higher-order constructions.

All of our semantic results on sharing graphs equally apply to action calculi (with some care on the treatment of controls). The conservativity of intuitionistic linear type theory over action calculi (as reported in [13]), the correspondence between higher-order action calculi and Moggi's work (as described in [35]), and the analysis of recursive computation in reflexive action calculi (c.f. [61]) are obtained as corollaries of results on sharing graphs.

Figure 1.6 is a summary of the correspondence between our theory of sharing graphs and action calculi:

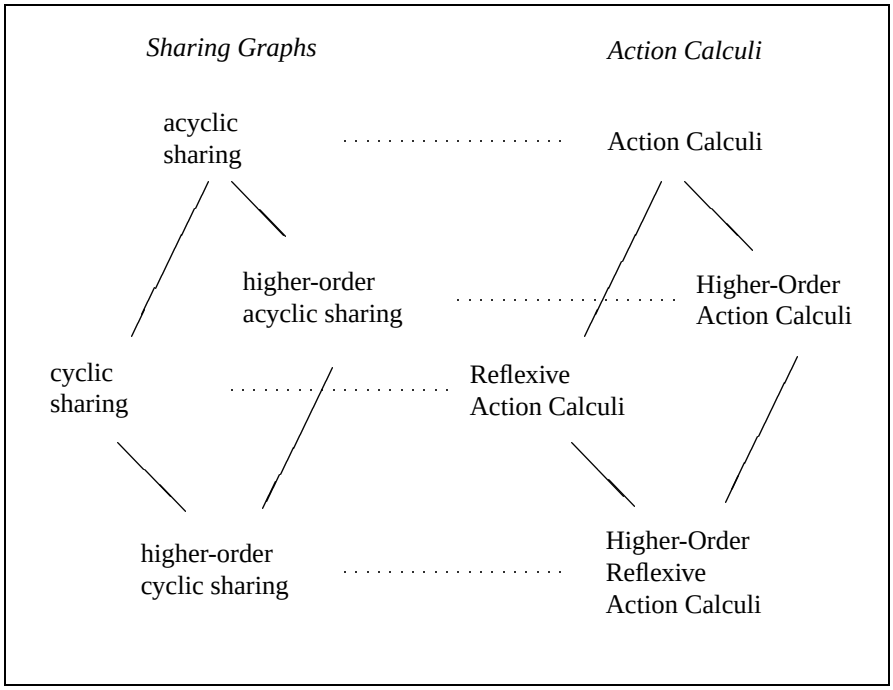


Figure 1.6: Sharing theories and action calculi

We hope that our work provides a bridge between graph rewriting theory and concurrency theory.

1.8 Overview

Chapter 2 introduces the notion of sharing graphs and the corresponding simply typed equational theories, called sharing theories. We emphasize the algebraic, structural nature of sharing graphs via the equational presentations, which leads us to the semantic development in the following chapters.

In Chapter 3 we study the category-theoretic models of acyclic sharing theories. In terms of symmetric monoidal categories and functors, we describe the class of models, and establish the soundness and completeness, in a similar way to the standard categorical type theory.

In Chapter 4 we give a higher-order extension of acyclic sharing. The models of this setting are obtained by assuming additional conditions formulated as adjunctions, and we repeat the same pattern as in Chapter 3.

As an application of our approach, in Chapter 5 we relate our acyclic sharing theories with notions of computation and intuitionistic linear type theory by comparing their classes of models.

In Chapter 6 we give the models of cyclic sharing, by additionally using the notion of traced monoidal categories. After reviewing traced monoidal categories, we establish the expected properties of our models, again in the same way as Chapter 3.

Chapter 7 describes higher-order cyclic sharing. The models of this setting, obtained by combining those in Chapter 4 and Chapter 6, are of particular interest as they support a generalized form of recursive computation. We look at this in some detail, together with the connection with cyclic lambda calculi.

Chapter 8 is devoted to showing that Milner's action calculi can be accommodated in our framework.

Finally, in Chapter 9, we conclude this thesis with some discussions towards further research.

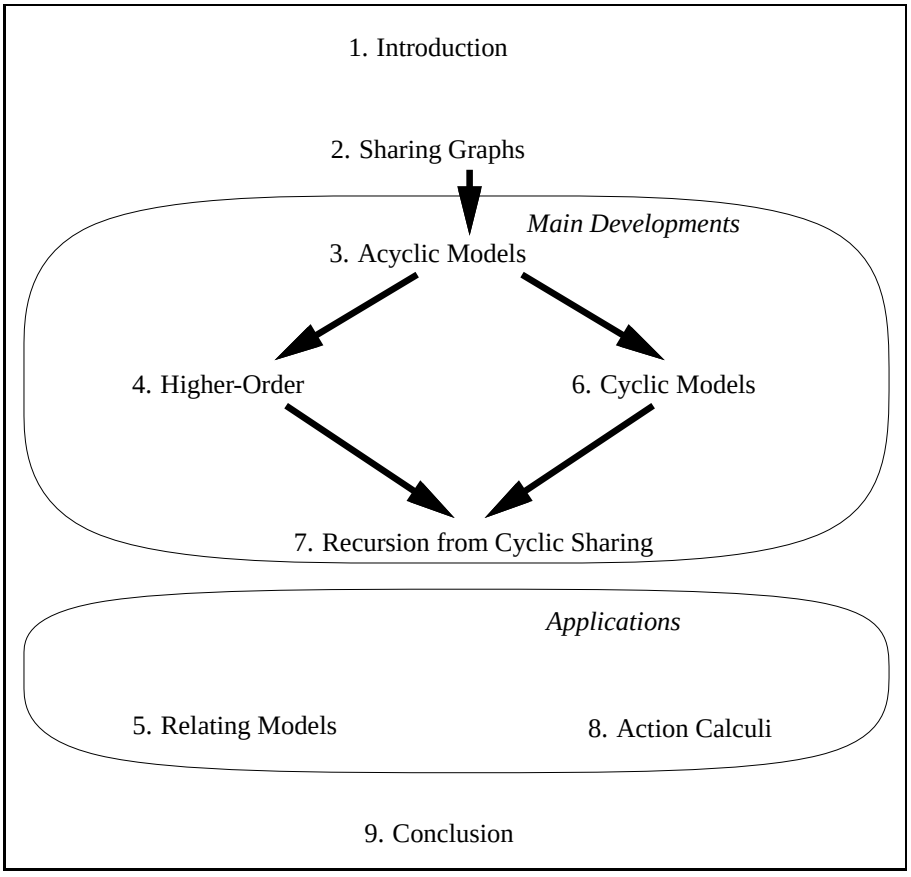


Figure 1.7: Overview of this book