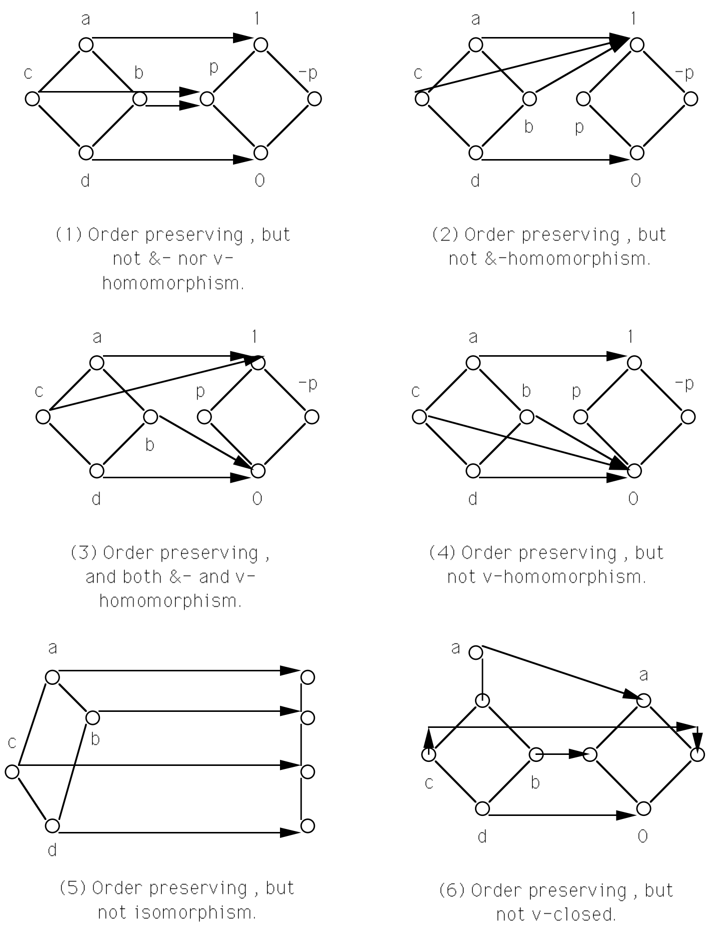
.1 Appendix

.1.1 Order and Structure Preserving Morphisms



(1) Order preserving , but not &- nor v-homomorphism.

(2) Order preserving , but not &-homomorphism.

(3) Order preserving , and both &- and v-homomorphism.

(4) Order preserving , but not v-homomorphism.

(5) Order preserving , but not isomorphism.

(6) Order preserving , but not v-closed.

Figure .1: Lattice Morphisms

[Spi89] Mike Spivey. "A Categorical Approach To The Theory of Lists." In *Mathematics of Program Construction*, volume 375 of *LNCS*. Spring-Verlag, 1989.

[Ste88] Mark Steedman. "Combinators and Grammars." In Emmon Bach Richard T. Oehrle and Deirdre Wheeler, editors, *Categorial Grammars and Natural Language Structures*. D.Reidel Publishing Company, 1988.

[Tar81] Robert Endre Tarjan. "A Unified Approach to Path Problems." *Journal Of ACM*, **28**(3):577–593, July 1981.

[Tho74] R. Thomason, editor. *Formal Philosophy : Selected Papers of Richard Montague*. Yale University Press, New Haven, 1974.

[Tiu90] Jerzy Tiuryn. "Type Inference Problems : A Survey." In B. Rovan, editor, *Proceedings of the 10th Conference on Automated Deduction*, volume 452 of *Lecture Notes in Computer Science*, pp. 105–120, Berlin, 1990. Springer-Verlag.

[Wad87] Philip Wadler. "Views : A Way for Pattern Matching to Cohabit with Data Abstraction." In B. Rovan, editor, *15th ACM Symposium of Principles of Programming Languages* , pp. 307–313. ACM, 1987.

[Wad90] Philip Wadler. "Comprehending Monads." In *Proc. ACM Conf. on LISP and Functional Programming*, Nice, France, June 1990. ACM.

[Wad92] Philip Wadler. "The Essence of Functional Programming." In *19th ACM Symposium of Principles of Programming Languages* , pp. 1–14. ACM, 1992.

[Wec92] Wolfgang Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[Zie81] W. Zielonka. "Axiomatizability of Ajdukiewicz-Lambek Calculus by Means of Cancellation Schemes." *Zeitschrift für Mathematishe Logik und Grundlagen der Mathematik*, **27**(3):215–224, 1981.

[Per91] Fernando C. N. Pereira. "Semantic Interpretation as higher-order deduction." In Jan van Eijck, editor, *Logics in AI : European Workshop JELIA'90*, volume 478 of *Lecture Notes in Artificial Intelligence*, pp. 78–96, Amesterdam, Holland, 1991. Springer-Verlag.

[PG82] Geoffrey K. Pullum and Gerald Gazdar. "Natural Languages and Context-Free Languages." *Linguistics and Philosophy*, **4**(4), 1982.

[Pie91] Benjamin C. Pierce. *Basic Category Theory For Computer Scientists*. MIT Press, 1991.

[PP91] Fernando C. N. Pereira and Martha E. Pollack. "Incremental Interpretation." *Artificial Intelligence*, **50**(1):37–82, june 1991.

[Pra71] Dag Prawitz. "Ideas and Results in Proof Theory." In J.E. Fenstad, editor, *Proceedongs of the second Scandinavian logic symposium*, volume 63 of *Studies in Logic*, pp. 235–209. North-Holland Publishing Company, 1971.

[Pra90] Vaughan Pratt. "Dynamic Algebra : Examples, Constructions, Applications." *Studia Logica*, **34**(2), 1990. 1992 ftp version.

[PS89] Lawrence C. Paulson and Andrew W. Smith. "Logic Programming, Functional Programming and Inductive Definition." In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, volume 475 of *Lecture Notes in Artificial Intelligence*. Springer-Verlay, 1989.

[PSV92] D. Stott Parker, Eric Simon, and Patrick Valduriez. "SVP – A Model Capturing Sets, Streams, and Parallelism." Technical Report 920020, UCLA, Computer Science Department, April 1992. short version appeared in *Intnl. Conf. on Very Large Data Bases*, Vancouver, Aug. 1992.

[RW88] Emmon Bach Richard T. Oehrle and Deirdre Wheeler, editors. *Categorial Grammars and Natural Language Structures*. D. Reidel Publishing Company, 1988.

[Sel85] Peter Sells. *Lectures on Contemporary Syntactic Theories : An Introduction to Government-Binding Theory, Generalized Phrase Structure Grammar, and Lexical-Functional Grammar*. CSLI, Center for the Study of Language and Information, 1985.

[KF70] A.N. Kolmogorov and S.V. Fomin. *Introductory to Real Analysis.* Dover Publications, Inc., 1970.

[Knu65] Donald E. Knuth. "On the Translation of Languages from Left to Right." *Information and Control*, **8**:607–639, 1965.

[KS86] Werner Kuich and Arto Salomaa. *Semirings, Automata, Languages*, volume 5 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1986.

[Lam58] Joachim Lambek. "The Mathematics of Sentence Structure." *American Mathematical Monthly*, **65**(3):154–170, 1958.

[Lam61] Joachim Lambek. "On the Calculus of Syntactic Types." In *Structure of Language and Its Mathematical Aspects.* Proc. of Symposia in Applied Mathematics, 1961. Vol.12.

[Llo89] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1989.

[Mal89] Grant Malcolm. "Homomorphism and Promotability." In *Mathematics of Program Construction*, volume 375 of *LNCS.* Spring-Verlag, 1989.

[Mee86] Lambert Meertens. "Algorithmics." In M. Hazewinkel J.W. de Bakker and J.K. Lenstra, editors, *Mathematics and Computer Science*, pp. 289–334. North-Holland, 1986.

[MG88] José Meseguer and Joseph A. Goguen. "Initiality, Induction and Computability." In M. Nivat and J.C. Reynold, editors, *Algebraic Methods in Semantics.* Cambrige University Press, 1988.

[Mil78] R. Milner. "A Theory of Type Polymorphism in Programming." *Journal of Computer and System Sciences*, **17**:348–375, 1978.

[Mit90] John C. Mitchell. "Type Systems for Programming Language." In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. The MIT Press, Elsevier, 1990.

[Moo88] Michael Moortgat. *Categorial Investigation : Logical and Linguistic Aspects of the Lambek Calculus.* Foris Publication, Dordrecht,Holland, 1988.

[Per90] Fernando C. N. Pereira. "Categorial Semantics and Scoping." *Computational Linguistics*, **16**(1):1–10, jan 1990.

[BM92] Richard S. Bird and Oege de Moor. "Solving Optimization Problem with Catamorphisms." In C.C. Morgan R.S. Bird and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *LNCS*. Springer-Verlay, 1992.

[BS91] Val Breazu-Tannen and Ramesh Subrahmanyam. "Logical and Computational Aspects of Programming With Sets/Bags/Lists." In B. Monien J. Leach Albert and M. R. Artalejo, editors, *Automata, Languages and Programming,18th International Colloquium, EATCS*, 1991.

[Bur75] W.H. Burge. *Recursive Programming Techniques*, chapter 3. Addison-Wesley, 1975.

[Bus88] Wojciech Buszkowski. "Generative Power of Categorial Grammars." In Emmon Bach Richard T. Oehrle and Deirdre Wheeler, editors, *Categorial Grammars and Natural Language Structures*. D.Reidel Publishing Company, 1988.

[etc85] D.H. Pitt etc. "Tutorial Section." In *Category Theory and Computer Science*, volume 248 of *LNCS*. Springer-Verlag, 1985.

[Gir87] Jean-Yves Girard. "Linear Logic." *Theoretical Computer Science*, **50**, 1987.

[GP87] Gerald Gazdar and Geoffrey K. Pullum. "Computationally Relevant Properties of Natural Languages and Their Grammars." In *The Formal Complexity of Natural Language*. D.Reidel Publishing Company, 1987.

[Hen60] Leon Henkin. "On Mathematical Induction." *American Mathematical Monthly*, **67**(4):323–356, April 1960.

[HP92] Shen-Tzay Huang and D. Stott Parker. "Types For Lazy Logic Programs : Computation and Infinite Objects." Technical Report 920029, UCLA Computer Science Dept., June 1992.

[Hue80] Gérard Huet. "Confluent Reduction : Abstract Properties and Applications to Term Rewriting Systems." *Journal Of ACM*, **27**(4):797–821, Oct. 1980.

[Jac85] Nathan Jacobson. *Basic Algebra I*. W.H. Freeman and Company, 1985.

[Kan92] Makoto Kanazawa. "The Lambek Calculus Enriched with Additional Connectives." *Journal of Logic, Language and Information*, **1**(1):141–171, jan 1992.

References

[ABK89] A. Agrawal, G.E. Blelloch, R.L. Krawitz, and C.A. Phillips. "Four Vector-Matrix Primitives." In *Proc. ACM SPAA*, 1989.

[Acz77] Peter Aczel. "An Introduction to Inductive Definition." In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies In Logic*. North-Holland Publishing Company, 1977.

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures : An Introduction to Category Theory for the Working Computer Scientist*. MIT Press, 1991.

[BBN91] Val Breazu-Tannen, O. Peter Buneman, and Shamim Naqvi. "Structural Recursion as a Query Language." In *Proc. Workshop on Data Base Programming Languages*, Napthlion, Greece, September 1991.

[Ben88a] Johan van Benthem. "The Lambek Calculus." In Emmon Bach Richard T. Oehrle and Deirdre Wheeler, editors, *Categorial Grammars and Natural Language Structures*. D. Reidel Publishing Company, 1988.

[Ben88b] Johan van Benthem. "The Semantics of Variety in Categorial Grammar." In Buzskowski et al., editor, *Categorial Grammars*, volume 25 of *Linguistic and Literary Studies in Eastern Europe*. D. Reidel Publishing Company, 1988.

[Bir67] Garrett Birkhoff. *Lattice Theory*. Providence,American Mathematical Society, 1967. 3rd edition.

[Bir87] Richard S. Bird. "An Introduction to The Theory of Lists." In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987.

[Bir92] Richard S. Bird. "Two Greedy Algorithms." *J. Functional Programming*, **2**(2), April 1992.

[BLW85] B.W. Bern, E.L. Lawler, and A.L. Wong. "Why Certain Subgraph Computations Require Only Linear Time." In *26th Symp. on Foundations of Computer Science*, Oct. 1985.

to unifying many problems, for instance, shortest path, systems of linear equation and network flow [Tar81], though not for algorithmic complexity.

- Along with the metaphor of rigid body, we can examine structures both richer and more restrictive than monoids. There are rich set of structures interesting to us, including groups, semirings, modules, trees, series-parallel nets, Halin graphs, etc. We are now particularly interested in an linear space extension for stream processing, for pattern processing and database applications.

  Our idea is to relax the underlying Abelian group of a linear space [KF70], as commutative monoids, together with the idea that homomorphisms are essentially distributive properties. Notice that, with commutative monoids, sets and bags are then instances of commutative monoids, with or without idempotency. The linear space approach can be used to further the exploration of the idea of duality in the rigid body metaphor, when cast in vector space.

- The analysis of the processes of constructing algebraic objects using universal algebra has the characteristic of universal construction. Furthermore, as the framework of dual-$\mathcal{M}$-sets takes a more concrete style, there is a connection using Category Theory. The notion of Monad [Wad90, Wad92] for structuring both declarative and imperative programs has also a close connection here, via the notion of adjunction. More generality seems promising in this line.

technical results here, this work presents our arguments for this belief. While serving as an inspiring metaphor, the idea of rigid body and related concepts from mathematics and physics would also help guide for further development of the framework of dual-$\mathcal{M}$-sets. The potential of the framework is surely worthy of further exploration, given the initial encouraging results reported in this dissertation.

6.1 Future Research

There are many directions to pursue here.

- For refining the foundations of the dual-$\mathcal{M}$-set, we are interested in order-theoretic aspects, in particular, state-based monotonicity. Homomorphism and structure-preservation is well-behaved, yet monotonicity and order-preservation is not, as we learn from modeling the Theory of Lists. As commented earlier, mathematical induction is useful but still far too limited in terms of computation. While not jumping into Turing computability, there are still spaces for exploration. Finite state transductions are obviously one area to search.

- The dual-$\mathcal{M}$-set framework is now primarily used for modeling and structuring. The aspect of problem solving, or algorithm development, is yet to be explored. There have been works along this line, using homomorphisms of initial algebra and incrementally refined the expression. Greedy methods and dynamic programming are two major techniques for program development; see [BM92, Bir92] for the former and [BLW85] for the latter. The idea of homomorphism of regular expression (or path expressions) are also used

concrete perspectives are more accessible.

We then apply the framework to model, comprehensively, two different systems: The Theory of Lists (also known as the Bird-Meertens formalism), for specification and systematic derivation of programs; and the Lambek Calculus, a deductive systems and formal foundation for linguistic applications in Categorial Grammars and formal semantics. Not only does the dual-$\mathcal{M}$-set successfully model these two seemingly disparate theories, it also provides new perspectives and directions for work. Beyond the modeling and new proofs of old results, we also develop extensions inspired by the perspectives that accompany the dual-$\mathcal{M}$-set modeling. In addition to new proof and perspective on the Homomorphism Lemma (and the ideal part), we examine the non-ideal part of the Theory of Lists, covering fictitious values, non-determinism, and the extension by inductive morphisms. The modeling of the Lambek Calculus by the dual-$\mathcal{M}$-set framework also provides a perspective leading to the issue of empty traces and pragmatical aspects of the Lambek Calculus as a formal foundation for linguistic competence. New variants are proposed whose properties are also examined.

As the rigid body metaphor is very basic in the modeling of information, monoid structures are prevalent in current mathematical program structures. The framework of dual-$\mathcal{M}$-sets aspires for a mathematical program development methodology exploiting this basic metaphor, in keeping with the common long term aim of developing programs as mathematical objects susceptible to mathematical analysis and manipulation.

We believe the dual-$\mathcal{M}$-set, while not yet an extensive theory, is a fruitful instance of the rigid body metaphor that is suitable for adaptation for software design and development. It is not merely another formal game. Beyond the other

CHAPTER 6

Conclusion

The notion of 'rigid body' in physics is an outlook which models information as the result of operator applications (measurements) on immutable objects (rigid bodies). The notion has a rich set of methods for modeling and problem solving. Inspired by this 'rigid body' metaphor, we look for possible analogous fragments in software development. A closer look at structure based data design, and pattern based programming practice, we focus on the examination of three processes of constructing a structured object : subalgebra closure, homomorphic image and term algebra formally, upon which applicability of equational and induction reasoning, and programming by patterns are based.

The new framework, the dual-$\mathcal{M}$-set, then evolves both as a means for structuring certain kinds of programs, and as a means for reasoning about aspects of software practice such as programming by patterns. This framework takes monoids as points and vectors, and structure preserving homomorphisms as linear operators. Order preserving endomorphisms (in particular, induction morphisms) are also included as basics. The conceptual framework treats both the objects and their functions as first class citizens. Instead of taking a conventional approach using category theory, we emphasize the local, yet universally uniform, analysis of the construction and interaction of subalgebraic closure and homomorphisms, and order-preserving morphisms. This approach pays off, as

1)	Axioms :		x	$\vdash_{PLC}$	x
2)	$(/)$ R. Introduction :	if	$\mathcal{T}; y$	$\vdash_{PLC}$	x
		then	$\mathcal{T}$	$\vdash_{PLC}$	(x/y)
2′)	$(\backslash)$ R. Introduction :	if	$y; \mathcal{T}$	$\vdash_{PLC}$	x
		then	$\mathcal{T}$	$\vdash_{PLC}$	$y\backslash x$
3)	Lemma :		$y; (y\backslash x)$	$\vdash_{PLC}$	x
			$(x/y); y$	$\vdash_{PLC}$	x
	Other Lemma :		$\mathcal{T}$	$\vdash_{PLC}$	x
4)	$(\bullet)$ L. Introduction	if	$\mathcal{U}; x; y; \mathcal{V}$	$\vdash_{PLC}$	z
		then	$\mathcal{U}; (x \bullet y); \mathcal{V}$	$\vdash_{PLC}$	z
5)	$(\bullet)$ R. Introduction	if	$\mathcal{U} \vdash_{PLC} x$	and	$\mathcal{V} \vdash_{PLC} y$
		then	$\mathcal{U}; \mathcal{V}$	$\vdash_{PLC}$	$(x \bullet y)$

Table 5.4: The Lemma Lambek Calculus

Lambek Calculus by finite cancellation schemes [Zie81]. As is easily checked, even with the L. Introduction rules replaced by such controlled use of cut rules, the lemma Lemma 5.4 (Adequacy) is still valid. In brief, while the resultant variant is less powerful, as shown in the previous proofs, we maintain theorems of linguistic significance and achieve a kind of meta-control within the object deduction system.

lemma access instance of cut rule respectively :

$$\frac{(x/y); y \vdash_{Lemma} x \qquad\qquad \mathcal{U}; x; \mathcal{V} \vdash_{PLC+Cut} z}{\mathcal{U}; (x/y); y; \mathcal{V} \vdash_{PLC+Cut} z \quad [\text{Cut at } x]}$$

$$\frac{y; (y\backslash x) \vdash_{Lemma} x \qquad\qquad \mathcal{U}; x; \mathcal{V} \vdash_{PLC+Cut} z}{\mathcal{U}; y; (y\backslash x); \mathcal{V} \vdash_{PLC+Cut} z \quad [\text{Cut at } x]}$$

Notice only lemma instances are used, not recursive invocation of deduction in PLC + Cut. Thus, if we allowed such controlled use of lemma access instances of cut rule, and postulate as lemmas well-behaved and linguistically significant rules, we can also remove the cut rule and come up with a more controlled deduction in PLC system with L. introduction rules.

5.4.5 The Lemma Lambek Calculus

Here we summarize the system what we have come up with Table (5.4):

together with the restricted lemma access instance of the cut rule :

$$\begin{array}{llll} \text{6) Cut Rule :} & \text{if} & \mathcal{T} & \vdash_{Lemma+Axiom} \quad x \\ & \text{and} & \mathcal{U}; x; \mathcal{V} & \vdash_{PLC+Cut} \quad y \\ & \text{then} & \mathcal{U}; \mathcal{T}; \mathcal{V} & \vdash_{PLC+Cut} \quad y \end{array}$$

This controlled use of cut rules as lemma access instances is not equivalent to the cut rule itself, due to the fact that there is no axiomatization of the

5.4.4 L-Elimination vs. Cut-Elimination

If we are more performance-oriented, the power of the cut rule to recover completeness seems to ruin its pragmatic use. Very often, it is the structure rule, of which the cut rule is an instance, to be removed from the pure logical inference rules [Pra71]. However, proof-theoretically, instances of the cut rule model the natural aspect of human reasoning, as appealing to lemma or previous gained knowledge. A controlled use of the cut rules should prove useful, before we hastily throw it away. In fact, we will show it is better to eliminate L. Introduction rules, with the help of cut rules, and to replace cut rules by instances of lemma access as a way of controlling cut rules.

We first notice that the L. Introductions can be simplified, by merging the instance $y \vdash_{PLC} y$ of Axiom rule into the rule, as

$$
\begin{array}{llll}
3) & (/)\ \text{L. Introduction}: & \text{if} & \mathcal{U}; x; \mathcal{V} \vdash_{PLC} z \\
 & & \text{then} & \mathcal{U}; (x/y); y; \mathcal{V} \vdash_{PLC} z \\
 & & & \\
3'') & (\backslash)\ \text{L. Introduction}: & \text{if} & \mathcal{U}; x; \mathcal{V} \vdash_{PLC} z \\
 & & \text{then} & \mathcal{U}; y; (y\backslash x); \mathcal{V} \vdash_{PLC} z
\end{array}
$$

Now, if we have the two lemmas $y; (y\backslash x) \vdash_{PLC} x$ and $(x/y); y \vdash_{PLC} x$, which can be derived using two instances of Axioms and one instance of L. Introduction, then it is possible to substitute the L.Introduction rule from the system with one

Basis (h=1) : *This is trivial, since only instances of axioms involved.*

Induction (h=n+1) : *Without loss of generality, let the root of height h=n+1 be an instance of* $(\backslash)$ *L. Introduction :*

$$\frac{\mathcal{T} \vdash_{LC} y \qquad\qquad \mathcal{U};x;\mathcal{V} \vdash_{LC} z}{\mathcal{U};\mathcal{T};(y\backslash x);\mathcal{V} \vdash_{LC} z} \quad [(\backslash)\ L.\ Intro.]$$

where both $\mathcal{T} \vdash_{LC} y$ *and* $\mathcal{U};x;\mathcal{V} \vdash_{LC} z$ *are of height less than n.*

By induction hypothesis, we have

1) $\mathcal{T} \vdash_{PLC+Cut} y$*, and*

2) $\mathcal{U};x;\mathcal{V} \vdash_{PLC+Cut} z$.

Furthermore, we can replace the root by the following compound instance, using only the restricted $(\backslash)$ *L. Introduction of the PLC system together with the cut rule :*

$$\dfrac{\dfrac{1)\ \mathcal{T} \vdash_{PLC+Cut} y \qquad \dfrac{\dfrac{y \vdash_{PLC} y \qquad x \vdash_{PLC} x}{3)\ y;(y\backslash x) \vdash_{PLC} x}\ [(\backslash)\ L.]}{}}{4)\ \mathcal{T};(y\backslash x) \vdash_{PLC+Cut} x}\ [1)\ Cut\ at\ y\ of\ 3)] \qquad 2)\ \mathcal{U};x;\mathcal{V} \vdash_{PLC+Cut} z}{\mathcal{U};\mathcal{T};(y\backslash x);\mathcal{V} \vdash_{PLC+Cut} z}\ [4)\ Cut\ at\ x\ of\ 2)]$$

This completes the induction and the proof. □

5.4.3 Power of The Cut Rule

We now show the cut rule,

$$
\begin{array}{llll}
\text{6) Cut Rule :} & \text{if} & \mathcal{T} & \vdash_{PLC+Cut} \; x \\
 & \text{and} & \mathcal{U}; x; \mathcal{V} & \vdash_{PLC+Cut} \; y \\
 & \text{then} & \mathcal{U}; \mathcal{T}; \mathcal{V} & \vdash_{PLC+Cut} \; y
\end{array}
$$

indeed adds extra deductive power to PLC system. The introduction of the cut rule into PLC system actually achieves completeness with respect the Lambek Calculus. To be precise, we have :

Lemma 5.5 (Power of The Cut Rule) $\vdash_{PLC+Cut} = \vdash_{LC}$

Proof 5.5 *($\Rightarrow$ Case :) This direction is easy, since the cut rule is eliminatability in the LC system [Lam58] and all rules in PLC + Cut are just instances of the LC system.*

($\Leftarrow$ Case :) Again by the cut-eliminatability in the system LC, we only need to consider cut-free proofs of the LC system, and the proof obligation is to establish that the original (\) and (/) L. rules can be restored, in PLC plus the Cut Rule. Equivalently, for any proof, $\mathcal{P}_{LC}$, in the LC system, we need have a corresponding proof, $\mathcal{P}_{PLC+Cut}$, in the PLC system with the cut rule.

Obviously this amounts to replacing each instance of L.Introduction in the proof $\mathcal{P}_{LC}$ by grafting several inference steps from the PLC or instances of the cut rule. We do the proof by induction on the height, h, of the proof tree of $\mathcal{P}_{LC}$. We only demonstrate the (\) L. Introduction, since the case for the (/) L. Introduction is symmetric.

G rule :

$$\dfrac{\dfrac{\dfrac{z \vdash_{PLC} z \qquad \dfrac{x \vdash_{PLC} x \qquad y \vdash_{PLC} y}{x;(x\backslash y) \vdash_{PLC} y}\ [(\backslash)\ L.]}{z;(z\backslash x);(x\backslash y) \vdash_{PLC} y}\ [(\backslash)\ L.]}{(z\backslash x);(x\backslash y) \vdash_{PLC} (z\backslash y)}\ [(\backslash)\ R.]}{x\backslash y \vdash_{PLC} ((z\backslash x)\backslash(z\backslash y))}\ [(\backslash)\ R.]$$

Application :

$$\dfrac{x \vdash_{PLC} x \qquad y \vdash_{PLC} y}{x;(x\backslash y) \vdash_{PLC} y}\ [(\backslash)\ L.]$$

Composition :

$$\dfrac{\dfrac{z \vdash_{PLC} z \qquad \dfrac{y \vdash_{PLC} y \qquad x \vdash_{PLC} x}{y;(y\backslash x) \vdash_{PLC} x}\ [(\backslash)\ L.]}{z;(z\backslash y);(y\backslash x) \vdash_{PLC} y}\ [(\backslash)\ L.]}{(z\backslash y);(y\backslash x) \vdash_{PLC} (z\backslash x)}\ [(\backslash)\ R.]$$

$\square$

Lemma 5.4 (Adequacy)

$$\begin{array}{ll}
\textit{M rule :} & x \vdash_{PLC} (y/(x\backslash y)) \\
& x \vdash_{PLC} ((y/x)\backslash y) \\
\textit{G rule :} & x/y \vdash_{PLC} ((x/z)/(y/z)) \\
& x\backslash y \vdash_{PLC} ((z\backslash x)\backslash(z\backslash y)) \\
\textit{Application :} & (x/y); y \vdash_{PLC} x \\
& y; (y\backslash x) \vdash_{PLC} x \\
\textit{Composition :} & (x/y); (y/z) \vdash_{PLC} (x/z) \\
& (z\backslash y); (y\backslash x) \vdash_{PLC} (z\backslash x)
\end{array}$$

Proof 5.4 *We do some cases and leave the symmetric counterparts for exercises.*

M rule :

$$\dfrac{\dfrac{x \vdash_{PLC} x \qquad\qquad y \vdash_{PLC} y}{x; (x\backslash y) \vdash_{PLC} y}\ [(\backslash)\ L.]}{x \vdash_{PLC} (y/(x\backslash y))}\ [(/)\ R.]$$

inclusion.

Lemma 5.3 (Proper Containment) $\mathcal{T} \vdash_{PLC} x \Rightarrow \mathcal{T} \vdash_{LC} x$

Proof 5.3 *The containment is obvious, since all instances of* $\vdash_{PLC}$ *are those of* $\vdash_{LC}$. *For the properness, note* $\vdash_{LC} y; (z/(y\backslash z))\backslash x \rightarrow x$ *:*

$$\dfrac{\dfrac{\dfrac{y \vdash_{LC} y \qquad z \vdash_{LC} z}{y; (y\backslash z) \vdash_{LC} z}\ [(\backslash)\ L.\ Intro.]}{y \vdash_{LC} (z/(y\backslash z))}\ [(/)\ R.] \qquad x \vdash_{LC} x}{y; (z/(y\backslash z))\backslash x \vdash x}\ [(\backslash)\ L.]$$

Yet, the last line is not a rule of $\vdash_{PLC}$. *Moreover, no other rules of* $\vdash_{PLC}$ *is applicable for the last line, since one of the preconditions must be* $y \vdash_{PLC} y$. *So it can not be derived in* $\vdash_{PLC}$. □

On the other hand, interesting theorems of linguistic significance are still derivable in this limited system PLC.

1)	Axioms :		x	$\vdash_{PLC}$	x	
2)	$(/)$ R. Introduction :	if	$\mathcal{T}; y$	$\vdash_{PLC}$	x	
		then	$\mathcal{T}$	$\vdash_{PLC}$	(x/y)	
3)	$(/)$ L. Introduction :	if	$(\mathcal{T} =)\ y \vdash_{PLC} y$	and	$\mathcal{U}; x; \mathcal{V} \vdash_{PLC} z$	
		then	$\mathcal{U}; (x/y); \mathcal{T}; \mathcal{V}$	$\vdash_{PLC}$	z	
$2'$)	$(\backslash)$ R. Introduction :	if	$y; \mathcal{T}$	$\vdash_{PLC}$	x	
		then	$\mathcal{T}$	$\vdash_{PLC}$	$y\backslash x$	
$3'$)	$(\backslash)$ L. Introduction :	if	$(\mathcal{T} =)\ y \vdash_{PLC} y$	and	$\mathcal{U}; x; \mathcal{V} \vdash_{PLC} z$	
		then	$\mathcal{U}; \mathcal{T}; (y\backslash x); \mathcal{V}$	$\vdash_{PLC}$	z	
4)	$(\bullet)$ L. Introduction	if	$\mathcal{U}; x; y; \mathcal{V}$	$\vdash_{PLC}$	z	
		then	$\mathcal{U}; (x \bullet y); \mathcal{V}$	$\vdash_{PLC}$	z	
5)	$(\bullet)$ R. Introduction	if	$\mathcal{U} \vdash_{PLC} x$	and	$\mathcal{V} \vdash_{PLC} y$	
		then	$\mathcal{U}; \mathcal{V}$	$\vdash_{PLC}$	$(x \bullet y)$	

Table 5.3: The Pragmatic Lambek Calculus (PLC)

y, or equivalently, immediately as an instance of Axiom. This is aimed at removing the derivation corresponding non-utterance, and thus intensional types, generated in the Lambek Calculus derivation.

The idea is based on the observation that instances of $(\backslash).L$ and $(/).L$ rules are the major source of derivations without concrete utterance, i.e. without uttered lexicons. On the other hand, we hope to maintain as many known nice linguistic rules as possible, such as the Montague rule and the Geach rule. Furthermore, the Cut rule will play a more important role as lemma access where more local language-specific constraints, such as subject noun phrase, i.e. subject raising, can be postulated and still properly controlled. As we shall see, these can be derived in the variant so they can be seen as limited instances of Cut rules.

The system is called the Pragmatic Lambek Calculus (PLC) with derivability $\vdash_{PLC}$, obviously in the pursuit of formalism of more linguistical reality, [GP87]. We also study the proof-theoretic properties and its relationship with the original system. The psycholinguistic reality of the PLC system appears to be an interesting research issue.

5.4.1 The Pragmatic Lambek Calculus (PLC)

The pragmatic version of the Lambek Calculus can be formalized as in the Table (5.3). Notice it includes a restricted version of the L rules.

5.4.2 Preserving Properties of Linguistic Significance

We first note indeed this is a proper subsystem, yet preserving all the interesting properties advocated as linguistically significant and meaningful. Notice the cut-rule is left out, as we will later show indeed more power is introduced with its

$\square$

Metalogically, the universality (or polymorphisms) of empty trace is being an identity for any type, and thus an idempotency element. This make the empty trace a serious exception for two structural rules, Thinning and Contraction, which are otherwise not applicable to the Lambek Calculus. To be precise, we have following two universal instances when parsing and syntactic type reduction are mixed :

$$\text{Thinning :} \quad \frac{\mathcal{T}, ([\,] : X) \vdash_{LC} \Delta}{\mathcal{T}, ([\,] : X/X), ([\,] : X) \vdash_{LC} \Delta}$$

$$\text{Contraction :} \quad \frac{\mathcal{T}, ([\,] : X/X), ([\,] : X/X) \vdash_{LC} \Delta}{\mathcal{T}, ([\,] : X/X) \vdash_{LC} \Delta}$$

To delimit these two structural rules, the handling of empty trace should be at the meta-level (control). Otherwise, the universality of the empty trace, and its proliferation would make the object-level handling complex and expensive.

5.4 A Pragmatic Variant and Its Properties

By the previous section, we identify some of the problems with the inadequacy for the Lambek Calculus for a pragmatic foundation for human language processing, as partially due to spurious nondeterminism by the empty trace, as being neutral element and representation of intensional objects. To resolve the problem of intensionality, we propose a variant by restricting the two ((\) and (/)) L. introduction rules to be applicable only for the special case where $\mathcal{T}$ is a singleton

Theorem 5.3 (Universal $[\,]$) *If $[\,] \in X$, for some Lambek syntactic type X, then $[\,] \in Y$ for every Lambek syntactic type Y.*

Proof 5.2 *Given any type X with $[\,] \in X$, we can derive $[\,] \in Z$, for arbitrary type Z in following steps. Assume T is an unknown token in the derivation, noticing that $T +\!\!+ [\,] = [\,]$ iff $T = zerolist$:*

- $$Application:\ \frac{([\,]:X) \vdash_{LC} ([\,] \in X)}{([\,] \in X)\ ;\ (T \in X\backslash X) \vdash_{LC} ([\,] \in X)}$$

  So we have $T = [\,]\ (\in (X\backslash X))$.

- *Since Montague rule is derivable,*

  $$Montague:\ \cfrac{\cfrac{(T : Z/X) \vdash_{LC} ([\,] \in Z/X)}{(T \in Z/X)\ ;\ ([\,] \in X) \vdash_{LC} ([\,] \in Z)}}{([\,] \in X) \vdash_{LC} ([\,] \in ((Z/X)\backslash Z))}$$

  Again, we have $T = [\,]\ (\in (Z/X))$.

- *Given $[\,] \in (Z/X)$ from above,*

  $$Application:\ \frac{([\,]:Z) \vdash_{LC} (T : Z)}{([\,]:Z/X)\ ;\ ([\,]:X) \vdash_{LC} (T : Z)}$$

  We have $T = [\,]\ (\in Z)$.

is not LR(k). But it is obviously a RL(0), by looking if the rightmost is a 'c' or not.

According to this example, together with the identification of deterministism with LR(k) grammar in the same article, any deterministic parsing mechanism or approach for incremental language processing should not have symmetry in its formalization. However, the Lambek Calculus has the obvious symmetry in its inference rules. Not only for theoretical interest, the left-to-right approach seems be to more faithful for psychological reality in human language processing. Notice, this is not to say all non-determinism comes from the symmetry of inference rules, nor does it insist on determinism being the primary concern.

5.3.2 Completion and Empty Traces

Another source of non-determinism and also inadequacy for human language processing is due to the empty trace. Here the notion of empty trace has a loose sense including non-uttered types, and intensional types, rather than just empty traces introduced by the principle of α-move [Sel85].

By its genesis, a type term is intensional, denoting a set abstraction of expressions. This is even complicated as such intensionality are externalized and internalized in the process of type reduction via $(/).L$ and $(\backslash).L$ rules. Such unuttered tokens add difficulties in parsing, if not unnecessarily. Notice, this aspect of empty trace as being of intensionality is not the same as ϵ-movement in finite automata. The ϵ-movement is only associated with certain states, rather than universal. On the other hand, there are the empty trace of arbitrary types, $[]$, if one type is ever allowed in the Lambek calculus. This can be seen by the following theorem.

petence model, it is not really a performance model. There are a couple variants adopting lexicalism, casting all information into the functor-argument structure of words, as of the Lambek Calculus: for example, Combinatorial Grammar [Ste88], incremental interpretation [PP91] and in semantics [Per90, Per91, Ben88b].

Here we discuss some of the problems, as illuminated by the dual-$\mathcal{M}$-sets framework. We first discuss the spurious non-determinism in the deduction, the symmetrical inference rules and also the empty traces. From these issues, we propose a pragmatic variant, which, while less powerful in type reduction, contains all linguistically significant rules, in particular, the Montague rule and the Geach rule.

5.3.1 Non-determinism and Left-to-Right Parsing

In his famous work, *On the Translation from Left to Right* [Knu65], Knuth develops LR(k) grammars also establishes a theorem making concrete the asymmetry of determinism with respect to leftward and rightward translation, using the equivalence of deterministic pushdown automata and LR(1).

Theorem 5.2 ([Knu65]) *The following production constitute an RL(0) grammar for which the corresponding language is not deterministic :*

$$S \longrightarrow A\,c, \quad S \longrightarrow B, \qquad A \longrightarrow a\,A\,b,$$

$$A \longrightarrow a\,b\,b, \quad B \longrightarrow a\,B\,b \quad B \longrightarrow a\,b.$$

This grammar corresponds to $\{a^n b^{2n} c\} \cup \{a^n b^n\}$. For any given stream with prefix a^k, there is no way to tell it is a prefix of $a^n b^{2n} c$ or of $a^n b^n$, for any k. Thus it

$$\text{Id}: \quad \frac{}{x \vdash x}$$

$$\text{Cut}: \quad \frac{\Gamma \vdash x \qquad \Delta, x, \Theta \vdash y}{\Delta, \Gamma, \Theta \vdash y}$$

and Logical Rules :

$$(/)\ \text{R. I.}: \quad \frac{\Gamma, y \vdash x}{\Gamma \vdash (y \multimap x)} \qquad (/)\ \text{L. I.}: \quad \frac{\Gamma \vdash y \qquad \Delta, x, \Theta \vdash z}{\Delta, (y \multimap x), \Gamma, \Theta \vdash z}$$

$$(\otimes)\ \text{L. I.}: \quad \frac{\Delta, x, y, \Theta \vdash z}{\Delta, (x \otimes y), \Theta \vdash z} \qquad (\otimes)\ \text{R. I.}: \quad \frac{\Gamma \vdash x \qquad \Delta \vdash y}{\Gamma, \Delta \vdash (x \otimes y)}$$

Notice both X/Y and $X\backslash Y$ correspond to $\mathcal{T}_{LL}(Y) \multimap \mathcal{T}_{LL}(X)$. So the Lambek Calculus refines the 'resource consciousness' of linear logic to provide also 'location consciousness'.

5.3 Linguistic Relevance

The Lambek calculus constitutes the foundation of Categorial Grammars, a framework based upon the principle of compositionality and correspondence between semantic type composition and syntactic type reduction. However, as a deductive system, it also has inherent limitations, with regard to computational purposes and natural language processing. In other words, while it could be a com-

- The formulation of the derivation relation, $\vdash$, of the Lambek calculus as a set of Horn Clause, is pretty obvious, as the inference rule are of implicational statements. Furthermore, it is very easy to see, inspired by the Curry-Howard isomorphism of types as propositions, that the Lambek Calculus is a special case of minimal propositional segment of linear logic [Gir87], where only positive proposition and implications are used. Precisely, there is only one formula at the left hand side, and the direct product (&) and direct sum ($\oplus$) is not used.

  The correspondence is as following Table (5.2);

Lambek Calculus	Linear Logic
x	proposition variable x
(X/Y)	$\mathcal{T}_{LL}(Y) \multimap \mathcal{T}_{LL}(X)$
$(X\backslash Y)$	$\mathcal{T}_{LL}(Y) \multimap \mathcal{T}_{LL}(X)$
$(X \bullet Y)$	$\mathcal{T}_{LL}(X) \otimes \mathcal{T}_{LL}(Y)$
$u_1; \cdots ; u_n$	$\mathcal{T}_{LL}(u_1), \cdots, \mathcal{T}_{LL}(u_n)$
$\mathcal{U} \vdash x$	$\mathcal{T}_{LL}(\mathcal{U}) \vdash \mathcal{T}_{LL}(x)$

Table 5.2: Correspondence of Lambek Calculus and Propositional Linear Logic

with Structural Rules :

terms as streams and patterns, made possible by the dual-$\mathcal{M}$-set framework. In comparison with the rigid body metaphor, these internal laws are additional or refined principles guiding the rigid body and their operators, such as (group) inverse, convergence and continuity.

- The pattern side is a monoid of monotone and antitone morphisms, as established upon the partial ordering, capturing the functions composed from the slash operators. Notice, these are not homomorphisms. The formulation of the predicate $\vdash_{LC}$ via Horn Clauses also serves as a confirmation. Indeed, we can simply read $\vdash_{LC}$ as equality, and then the Horn Clauses and also the Lambek Calculus, is obviously an implicational theory of identity, which by a characterization of universal algebra [Bir67], is not closed under homomorphic image. This again shows the need to consider monotone morphisms beyond structure homomorphisms, in the framework of dual-$\mathcal{M}$-sets. Notice, the partial ordering here is not inductive in general, for we can interpret type terms as sets of words.

- While there is no Λ in the Lambek Calculus, our modeling should be considered as a completion of a semigroup which does not require the neutral element Λ in either the stream side or the pattern side. The completion construction allows us to fully neglect the case when there is indeed no Λ involved. In other words, it is a conservative extension of the Lambek Calculus. However, the Λ does have linguistic significance when considered as the empty trace or null stream pattern. On the other hand, there are computational implications for parsing, because of its universal proliferation and its polymorphism (of type $\forall$ type $t.\Lambda : t$).

If the test $T \vdash_{LC} X$ succeeds (is true), we obtain the suffix after the guessed position as the residue. We use $inv(X)$ to represent the failure of Lambek derivation and to return the residue X, and operationally propagates the failure. Notice the $inv(X)$'s are only used as internal flags, or extra arguments, which can be further filtered out with additional interface between $\oslash$ and $guessFilter$. The advantage is that we are able to report the position of errors.

Also from the first clause of $guessFilter$, we have the identity for the monoid of patterns. We check, as a result of $(\bullet).L.$ rule and operationally the left-to-right search strategy, that :

$$X \oslash (Y \oslash T)) = (Y \bullet X) \oslash T.$$

□

5.2.3 Remarks

Many observations, of linguistic and logical nature, are gained via the modeling by the dual-$\mathcal{M}$-sets framework. Later we will discuss this further. Here, we give a few remarks specific to the modeling and the proof.

- The modeling is really about the monoidal aspect of the Lambek Calculus in action, namely, how it is used. There are also internal constraints, for example, the unification of type symbols and recursive call of the decision procedure. These internal laws are the content of the Lambek Calculus, in particular, about the slashed operators or laws of syntactic types. The modeling only extracts the $\bullet$ aspect, which is obviously a monoid. The insight here is the recognition of this monoid aspect, both its algebraic structure and its operational meaning for recognition and the duality of type

[X] $\vdash_{LC}$ [X].

[T | Rest] $\vdash_{LC}$ [Y \ X]	:–	[Y,T | Rest] $\vdash_{LC}$ [X].
[T | Rest] $\vdash_{LC}$ [X/Y]	:–	append([T | Rest],[Y], PremiseTypes),
		PremiseTypes $\vdash_{LC}$ [X].
[X,Y | Rest] $\vdash_{LC}$ [(A • B)]	:–	append([P | ResP], [Q | ResQ], [X,Y | Rest]),
		[P | ResP] $\vdash_{LC}$ [A], [Q | ResQ] $\vdash_{LC}$ [B].
Antecedent $\vdash_{LC}$ [Z]	:–	append(Left, [(Y\ X) | V], Antecedent),
		append(U, [T | Res], Left), [T | Res] $\vdash_{LC}$ [Y],
		append(U,[X | V], Prem), Prem $\vdash_{LC}$ [Z].
Antecedent $\vdash_{LC}$ [Z]	:–	append(U, [(X/Y) | Right], Antecedent),
		append([T | Res], V, Right), [T | Res] $\vdash_{LC}$ [Y],
		append(U, [X | V], Prem), Prem $\vdash_{LC}$ [Z].
Antecedent $\vdash_{LC}$ [Z]	:–	append(U, [(X • Y) | V], Antecedent),
		append(U, [X, Y | V], Prem), Prem $\vdash_{LC}$ [Z].

where, guessFilter and the infix predicate $\vdash$ *are defined by the following two sets of Horn clauses:*[2]

guessFilter(Λ, T, T).

guessFilter(X, [ ], inv(X)).

guessFilter(X, T, Rest) :– append(Prefix,Rest, T),
 T $\vdash_{LC}$ X.

guessFilter(X, T, inv(X)) :– append(Prefix, Suffix, T),
 not(T $\vdash_{LC}$ X).

guessFilter(X, [inv(Z) | T], [inv(X) | T]).

[2]The use of negation, in 'not(T $\vdash_{LC}$ X)', is sound, as the relation is decidable. Of course, we would expect a sound implementation of search methods, to employ the negation as failure. In fact, since the predicate iterates through the complexity of type terms, number of operators, the concerned of search methods can be waived.

term in the pattern side is a function from $< M, `;', \Lambda >$ to $< M, `;', \Lambda >$, i.e., an endomorphism of $< M, `;', \Lambda >$.

5.2.2 Dual $\mathcal{M}$-set Modeling of the Lambek Calculus

Given the basis set up in the last section, the dual-$\mathcal{M}$-set modeling is obvious. The only thing left is to define is the map $\oslash$, and make sure it satisfies the required associativity and the property that Λ be the unit. We state the theorem and give a proof, by giving a pure Prolog, Horn Clause, definition of $\oslash$.

Theorem 5.1 *The formal system of the Lambek Calculus can be expressed as a dual-$\mathcal{M}$-set framework, $<< M, \bullet, \Lambda >, \oslash, < M, `;', \Lambda >>$.*

Proof 5.1 (Guess and Filter) *This proof explores the nondeterminism of the Lambek Calculus by guessing one position in the input stream and then submitting it to the decidable deduction procedure, as suggested by the dual interpretation of type terms as streams and patterns, the downward property of patterns as functions, and the decidability of the Lambek Calculus.*

$$
\begin{array}{lll}
Input & : & X, \\
 & & \\
 & & (t_0; \cdots, t_n) \\
 & & \\
Output & : & (t_j; \cdots; t_n), 1 \leq n
\end{array}
$$

$$X \oslash (t_0; \cdots; t_n) = Rest \stackrel{def}{=} guessFilter(X, (t_o; \cdots; t_n), Rest).$$

transduction, matching x as a sequence of type terms, an input stream. To avoid confusion from mixed object and function level, we call the left hand side, the stream side, and the right hand side, the pattern side. With the $(\bullet).L.$ rule, the stream side is the $\bullet$-free segment of $\mathcal{S}$, the universe of type terms. The stream side then is a monoid, $< M, `;', \Lambda >$, where M is the $\bullet$-free segment of $\mathcal{S}^*$. In stream side, $\bullet$ is interpreted as list concatenation, $+\!\!+$. Dually, on the pattern side, by the $(\bullet).R.$ rule, is also a $\bullet$-free segment of $\mathcal{S}$. The pattern side is also a monoid, $< M, \bullet, \Lambda >$, where M is the $\bullet$-free segment of $\mathcal{S}^*$. The $\bullet$ is interpreted as ordinary function composition, $\circ$, instead. Notice, both Λ in the two monoids are artificial elements, admitted by the completion of a semigroup to a monoid. Interpretations of Λ also differ. For stream side, it is the null stream, $[\,]$, whereas, for pattern side, the identity pattern, Id. Notice, the Λ is not part of the Lambek Calculus, as it is required that both sides of derivation formula, $x \vdash y$, can not be empty. We will discuss more about the introduction of Λ, in particular, its effect as an empty trace.

Notice, the dual interpretation of type terms and their sequencing is meta-theoretical, namely, the objects in the discourse are formal syntactic terms of the Lambek Calculus, not a specific model. Intuitively, the pattern side is better interpreted as relations than as the functions used here. This is consistent with the usual reading of patterns as (formal) languages, or recognizable sets. However, the proof-theoretic property, downward closure, together with the decidability of the Lambek Calculus, allow us to forge a special function uniformly out of the relational view. For instance, we can take the shortest prefix or longest prefix (given an finite input), w.r.t. $+\!\!+$ reading of stream side. Not committed to specific choices, we take the freedom to treat patterns as functions. So each type

Also, with the following :

$$\dfrac{\dfrac{X_1 \vdash X_2 \qquad\qquad X_3 \vdash X_4}{X_1/X_3; X_4 \vdash X_2}\ (/).\mathrm{L}}{X_1/X_3 \vdash X_2/X_4}\ (/).\ \mathrm{R}$$

by identifying X_1 and X_2, (X_3 and X_4), we have antitonicity at argument position (mononicity at functor position), for the / operator. Similarly, we have antitonicity for argument position and monotonicity at functor position for the \ operator.

$$\dfrac{\dfrac{X_1 \vdash X_2 \qquad\qquad X_3 \vdash X_4}{X_3; (X_4 \backslash X_1) \vdash X_2}\ (\backslash).L}{X_4 \backslash X_1 \vdash X_3 \backslash X_2}\ (\backslash).R$$

So, the curried unary counterpart of those binary operators would be monotone and antitone, for example, $\lambda x.(x \bullet y)$ and $\lambda x.(x/y)$ monotone, while $\lambda x.(y/x)$ and $\lambda x.(x \backslash y)$ antitone.

We also need the downward property, as an instance of the Cut rule, (or transitivity by interchanging $\vdash$ with $\leq$) :

Lemma 5.2 (Downward Closure) *If $x \vdash y$, then $\mathcal{T} \leq x$ implies $\mathcal{T} \vdash y$.*

Now, instead of reading type terms in both sides of a derivability formula, $x \vdash y$, at a same object level, we can in fact read x in the object level, and y at (pattern) functional level. This is intended to interpret y as a pattern or a

of highest cut instances. The decidability of the resulting system is immediate, since each rule introduces a connective and there are only a finite number of connectives in any given sequent.

Lemma 5.1 (Cut-Elimination and Decidability) *[Lam58] The cut rule is redundant. The Gentzen Sequent formulation of the Lambek Calculus is decidable, namely, for any two type terms $\mathcal{T}$, $\mathcal{U}$, it is decidable whether $\mathcal{T} \vdash \mathcal{U}$.*

5.2 Dual-$\mathcal{M}$-set Modeling of the Lambek Calculus

5.2.1 Partial Ordering and Monotonicity

Before we do the dual-$\mathcal{M}$-set modeling, a few observations will be of help for understanding the Lambek Calculus and the modeling. We note that, the interpretation of deduction relation and the downward closure property, capturing the notion of logical implication, has a general setting applicable to arbitrary deduction systems. In other words, the dual-$\mathcal{M}$-set modeling of the Lambek Calculus may be applicable to relevant logics, including, for example, the propositional segments of Horn Logic, intuitionistic logic and linear logics.

First we note that the deduction relation, $\vdash$, is a partial ordering, $\leq$, between type terms, modulo antisymmetry, similar to that of logical implication, as the Axiom provides reflexivity and the Cut rule transitivity.

It is also easy to see that $\bullet$ (product) is monotone in its two arguments, by the ($\bullet$) introduction rule, namely, $x \vdash x$ and $y \vdash z$, so $x \bullet y \vdash (x \bullet z)$ and $y \bullet x \vdash z \bullet x$.

1)	Axioms :	$x \vdash x$
2)	(/) R. Introduction :	$\dfrac{\mathcal{T}; y \vdash x}{\mathcal{T} \vdash (x/y)}$
3)	(/) L. Introduction :	$\dfrac{\mathcal{T} \vdash y \qquad \mathcal{U}; x; \mathcal{V} \vdash z}{\mathcal{U}; (x/y); \mathcal{T}; \mathcal{V} \vdash z}$
2′)	(\\) R. Introduction :	$\dfrac{y; \mathcal{T} \vdash x}{\mathcal{T} \vdash y \backslash x}$
3′)	(\\) L. Introduction :	$\dfrac{\mathcal{T} \vdash y \qquad \mathcal{U}; x; \mathcal{V} \vdash z}{\mathcal{U}; \mathcal{T}; (y \backslash x); \mathcal{V} \vdash z}$
4)	(•) L. Introduction	$\dfrac{\mathcal{U}; x; y; \mathcal{V} \vdash z}{\mathcal{U}; (x \bullet y); \mathcal{V} \vdash z}$
5)	(•) R. Introduction	$\dfrac{\mathcal{U} \vdash x \qquad \mathcal{V} \vdash y}{\mathcal{U}; \mathcal{V} \vdash (x \bullet y)}$

Table 5.1: The Lambek Calculus

where ';' is sequence concatenation:

$$\dfrac{\dfrac{\dfrac{\text{John} \qquad \text{never} \qquad \text{works}}{N \;;\; (N\backslash S)/(N\backslash S) \;;\; (N\backslash S)} \text{(initial type assignment)}}{N \;;\; N\backslash S} \text{((/).R.Intro.)}}{S} \text{((\\).L.Intro.)}$$

This example uses several rules of inference, which we can now describe.

5.1.1 Gentzen Sequent Calculus For the Lambek Calculus

As a formal system, the Lambek (Syntactic) Calculus is based on $\mathcal{S}$, the universe of type terms, which is the inductive closure of the set, $\mathcal{P}$, of primitive type symbols, by three operators (or connectives): ($\bullet$), (/), and (\), each of signature $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$. It can be developed as a deductive system with axioms and rules of inference. This system can be represented by the Gentzen sequent calculus, Table (5.1), among many equivalent formulation [Lam58, Lam61]. In the following, x, y, z are types in $\mathcal{S}$, and $\mathcal{T}$,$\mathcal{U}$,$\mathcal{V}$ are sequences of types.

$$6)\quad \text{Cut Rule :} \quad \frac{\mathcal{T} \vdash x \qquad\qquad \mathcal{U}; x; \mathcal{V} \vdash y}{\mathcal{U}; \mathcal{T}; \mathcal{V} \vdash y}$$

The Cut Rule can be eliminated, using a proof technique similar to (yet simpler than) Gentzen's Cut Elimination Theorem for intuitionistic propositional logic, by transforming a proof tree into cut-free proof tree using induction on the rank

Word	*Category*
John	N
big	N/N
works	$N\backslash S$
never	$(N\backslash S)/(N\backslash S)$
rarely	$(N\backslash S)/(N\backslash S)$
rarely	$(N\backslash S)\backslash(N\backslash S)$

Each category is a functional *pattern*, in a sense describing the role or effect of the word in a sentence. The word "John" yields a $Noun$. Verbs such as "works" map nouns into sentences (adding a verb to a noun gives a sentence), so they should be given a type like "$N \mapsto S$". In the Lambek calculus, this type is either $N\backslash S$ or S/N, which also captures insistence on specific word orderings. Specifically, $N\backslash S$ maps a *preceding* N to an S, while S/N would map a *following* N to an S, so the word "works" gives a sentence when preceded by a noun. English language categories (verb, adjective, etc.) are typically identifiable with a small number of type terms. For example, adverbs could be identified with $(N\backslash S)\backslash(N\backslash S)$ or $(N\backslash S)/(N\backslash S)$, i.e., something that maps preceding or following verbs to verbs.

The Lambek calculus can be viewed as a combinator algebra on linguistic categories. Sequences of words can be 'parsed' into sentences by reducing their category sequence to S. The following simple example shows how this works,

trace and the use of non-determinism by the Lambek Calculus for parsing. We propose alternatives, based upon those observations, aiming pragmatically at performance.

With most uses of the Lambek Calculus, the objects are words or phrases, and the types are linguistic categories. However, the calculus can also be treated as a general formal system, and the linguistic interpretation is not the only one possible. In the modeling, we use a Horn Clause formulation of the derivation relation in the Lambek calculus. As is well known, Horn logic is intuitionistic. As a result, the Lambek Calculus corresponds to a minimal propositional Linear logic.

While many of the results are not deep by themselves, we emphasize the organizing power offered by the inspiring dual-$\mathcal{M}$-set perspective. Indeed, the rationale for the empty trace and the pragmatic alternatives occurred to the author and were justified after the perspective was established.

5.1 The Classical Lambek Calculus

In the Lambek Calculus, English words are assigned categories (types). [1] For example, we can have the following *lexicon* — a type assignment between English words and Lambek type terms:

[1] The word "category" here is used in the sense of Aristotle and Kant, with the sense of "classification". This use is dominant in Linguistics, Philosophy, and Cognitive Science. It has only indirect connections with Category Theory.

CHAPTER 5

Reconstructing the Lambek Calculus

The Lambek Calculus was first introduced in late 1950s by Lambek [Lam58, Lam61] as a mathematical foundation for a syntactic calculus of English sentences. It is a very simple yet powerful deductive system with several variants, and it has a decision procedure based upon a proof similar to the Gentzen Cut-Elimination Theorem for intuitionistic propositional logic. However, while it received acclaim under the umbrella of Categorial Grammars of formal linguistics, it has hardly reached the attention of computer scientists or even logicians. Interested readers may consult [RW88, Ben88a, Bus88, Moo88]. Several variants of the Lambek Calculus and their executable implementations are developed in [HP92].

The organization of this chapter is as follows. First we introduce a sequent calculus version of the Lambek Calculus. Next, we proceed to model the Lambek Calculus as a dual-$\mathcal{M}$-set. Our basic idea here is that *types* (kinds, patterns, linguistic categories, ...) can be treated as *recognizers*, so they can be viewed as the dual space of the objects they recognize. Furthermore, we treat type terms dually as patterns and as elements in streams interchangeably. This is made possible by treating derivability as a partial ordering, and by identifying patterns as monotone morphism respecting this partial ordering. The modeling also points out a few issues of linguistic significance. Among others, it underscores the empty

4.5 Summary

In sum, the framework of dual-$\mathcal{M}$-set, $<< M, \circ, Id >, apply, < \Sigma^*, +\!\!+, [\,] >>$, is expressive enough to model not only the ideal part, but also the non-ideal part of the theory of lists, regarding the notion of *Computable Functions over Lists.* In particular, the Homomorphism Lemma and thus laws implied can be captured in the framework. The modeling using the dual $\mathcal{M}$-set framework also distinguishes itself by its the separation of syntactic and semantic aspects through the interface interpretation homomorphisms. The advantage of such a separation due the dual $\mathcal{M}$-set framework is particularly seen in the account of fictitious values, nondeterminism, and also inductive morphisms. The framework also allows incremental identification and introduction of new morphisms, especially in connection with mathematical induction and inductive morphisms. It also suggests inclusion of additional morphisms, especially order-theoretic morphisms.

As with any formal framework, more examples would justify the framework's usefulness and power. However, the analysis here illuminates the issues and development of ideas using the rigid body metaphor, and shows various advantages over developments with, say, mathematically sophisticated category theory.

Partial ordering and monotone morphisms, with emphasis on the special instance of the inductive chain, is a useful extension in the framework. Instead of jumping to the limit with complete partial orderings, we have taken the strategy of incremental enrichment. Thus we have isolated the partial orderings in semantic domains from the complete partial orderings of computation. We suspect this is best for programming in practice.

In general the converse does not hold, as the 'difference' between $h(C_i)$ and $h(C_{i+1})$ may not be a function depending only upon the 'difference' of C_i and C_{i+1}, or i. For instance :

Example 4.6 *Let α be a two element set, $\{a, b\}$, and β be the natural number, $< N, +, 0 >$. Consider the inductive chain, $\{d_i\}$, of $list(\alpha)$: $[\,] < [a] < [a]+\!\!+[b] < [a]+\!\!+[b]+\!\!+[a] < \cdots$ (alternating $[a]$ and $[b]$). The image, $\{r_i\}$, of an order preserving inductive morphism could be $0 < 0 + 0$ ($r_1 = 0$) $< 0 + 0 + 1$ ($r_2 = 1$) $< 0 + 0 + 1 + 3$ ($r_3 = 4$) $< 0 + 0 + 1 + 3 + 5$ ($r_4 = 9$) $< 0 + 0 + 1 + 3 + 5 + 10$ ($r_5 = 19$) $< 0 + 0 + 1 + 3 + 5 + 10 + 10$ ($r_6 = 29$) $< \cdots$, where the i-th term is the accumulative sum of preceding terms and a value from the current term. From the type specification, $\oplus$ must be $+$, and $f : \{a, b\} \rightarrow N$. However, for this instance $f(a)$ is required to be 0, 3, 10 in r_1, r_3 and r_5 respectively, which contradicts the well-definedness of a single-valued function.*

The example is basically a case where 'localized' uniformity may not always be available, and additional parameters or 'history' information are needed to allow such general inductions. This is also a case where primitive (and general) recursion is more expressive than mathematical induction. A typical example is the exponential function, m^n, definable by primitive recursion but not mathematical induction. However, for the special cases where $\oplus$ is $+\!\!+$, we do have the property, because the range is then an inductive model for which there exists a unique definition by mathematical induction from an inductive chain. This follows from the Recursion Theorem [Jac85, Hen60], even though the f would be complicated. This is the same as encapsulating and localizing the complexity inside the list (type) constructor. For example, the function, (prefix even)$(\cdot)$, is inductive, since (prefix even)$(\cdot) = (\oplus \not\to [\,]) \circ (id)^*$.

possible partly because we treat such mappings as peripheral mappings, in general, to those algebras representable by (curried) unary operations.

Lemma 4.5 (Inductive Morphism Lemma) *A function* $h :$ $< list(\alpha), +\!\!\!+, [\,] > \rightarrow < \beta, \oplus, e >$ *is an inductive morphism with respect to* $+\!\!\!+$ *if* $h = (\oplus\not\rightarrow\beta_1) \circ f^*$ *for some function* $f : \alpha \rightarrow \beta$*, and* $\beta_1 \in \beta$*, where* $\oplus$ *induces a natural ordering.*

Proof 4.3 *The domain and image of* f^* *are both inductive chains,* $[\,], [\,]+\!\!\!+[\alpha_1], \cdots$ *and* $[\,], [\,]+\!\!\!+[f(\alpha_1)], \cdots$*, isomorphic to a Peano number system. Furthermore,* $(\oplus\not\rightarrow\beta_1)$ *is defined by mathematical induction, and its image can be ordered by the rank of iteration (equivalently, the number of applications of* $\oplus$*).*

$\square$

Notice, the ordering constraint is essential, as can be seen for the following :

Example 4.5 *Let* α *be natural number,* β *be* $\{0, 1\}$*,* $\oplus$ *be the modulo 2 addition, and let* f *yield modulo 2 value, so* $f(x) = (x \bmod 2)$*:*

$\oplus$	0	1
0	0	1
1	1	0

	$f(x)$
$2 \times n$	0
$2 \times n + 1$	1

Now consider the set of chain of $\{[1, 2, 1, 2, 2, 1, 2, 2, 2, \cdots, 1,$ *(2 for i times)*$]\}$. *The image, of* $(\oplus\not\rightarrow 0) \circ f^*$*, is not partially ordered, furthermore it is not of fixed finite periodicity, and thus can never be a homomorphic image of an Peano Model, by Henkin's theorem.*

Example 4.4 *The following function,*

$$
\begin{array}{rcll}
prefix\ \ p & = & (\oplus \not\to [\,]) & \\
where\ \ a \oplus x & = & [a] +\!\!+ x & if\ p(a) \\
 & & [\,] & otherwise
\end{array}
$$

return the longest prefix in which each element satisfy the property p(), for example, prefix even [2,4,1,8] = [2,4],

$$
\begin{array}{lcccccccc}
Domain & : & [\,] & \rightarrow & [\,]+\!\!+[2] & \rightarrow & [\,]+\!\!+[2]+\!\!+[4] & \rightarrow & \cdots \\
Range & : & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 & & [\,] & & [\,]+\!\!+[2] & & [\,]+\!\!+[2]+\!\!+[4] & & [\,]+\!\!+[2]+\!\!+[4]
\end{array}
$$

Again, for each inductive chain, the Range is a non-Peano inductive chain, as it is not injective (failing **(P2)***). However, it is parameterized by chains, not by* α*; in other words, it is an inductive morphism with respect to the ordering on the chains.*

This further distinguishes the aspect of monotonicity of this function from the previous one, as a surjective mapping onto a Peano model can be restored if all such chains are considered.

□

4.4.1 Inductive Morphism Lemma

From above examples, and by the notion of inductive chain and inductive morphisms, we can extend the homomorphism lemma a little bit. Notice this is

This is seen as an instance of duality of a naturally induced partial ordering and its reverse.

4.4 Inductive Definition

We now provide examples of monotone morphisms, based on the notion of inductive chain. The first is a homomorphism, and as we prove, thus a monotone morphism. The second is a non-homomorphism, but is monotone, and in fact, a mapping from an inductive chain to another inductive chain.

Definition 4.1 (Inductive Morphism) *An inductive morphism, from $< \alpha, \oplus, e_\alpha >$ to $< \beta, \otimes, e_\beta >$, is an order preserving mapping from an inductive chain of ω length in α with a naturally induced ordering w.r.t. $\oplus$, whose image can be ordered as a (not necessarily inductive) chain in β.*

Example 4.3 *The head function returns the first element of any input list.*

$$head \stackrel{def}{=} \ll / \circ (id^*)$$

where $a \ll b = a$, $\forall a, b$. This is a homomorphism by the Homomorphism Lemma.

$$\begin{array}{lclcccccc}
Domain & : & [\,] & \rightarrow & [\,] \mathbin{+\!\!+} [a_1] & \rightarrow & [\,] \mathbin{+\!\!+} [a_1] \mathbin{+\!\!+} [a_2] & \rightarrow & \cdots \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
Range & : & [\,] & & [\,] \mathbin{+\!\!+} [a_1] & & [\,] \mathbin{+\!\!+} [a_1] & & [\,] \mathbin{+\!\!+} [a_1]
\end{array}$$

As shown, the Range is an inductive chain, but not a Peano model. Furthermore, although it is parameterized polymorphically (by α), the collection of such inductive model is simply $|\,\alpha\,|$ copies of a common non-Peano inductive chain. $\Box$

left-reduction $\oplus\!\not\rightarrow\! e$ and right-reduction $\oplus\!\not\leftarrow\! e$, defined as

$$\begin{aligned}
(\oplus\!\not\leftarrow\! e)[a_1, a_2, \ldots, a_n] &= a_1 \oplus (a_2 \oplus (\cdots (a_n \oplus e) \cdots)) \\
(\oplus\!\not\rightarrow\! e)[a_1, a_2, \ldots, a_n] &= (\cdots ((e \oplus a_1) \oplus a_2) \cdots) \oplus a_n.
\end{aligned}$$

This $f = (\oplus\!\not\rightarrow\! e)$ can be regarded as the solution of recursive equations :

$$\begin{aligned}
f &= g\, e \\
g\, e\, [\,] &= e \\
g\, e\, ([a] +\!\!+ x) &= g\, (e \oplus a) x
\end{aligned}$$

Also we have :

Lemma 4.3 (Specialization [Bir87]) *Every homomorphism can be defined as either a left or right reduction. More precisely,*

$$(\odot/) \circ (f^*) = (\oplus\!\not\rightarrow\! e) = (\otimes\!\not\leftarrow\! e),$$

where

$$\begin{aligned}
a \oplus b &= f(a) \odot b, \\
a \otimes b &= a \odot f(b).
\end{aligned}$$

To establish well-definedness, we need only notice that for all $\{a_1, \cdots, a_n \cdots\} \subseteq \alpha$, we have $[\,], [\,] +\!\!+ [a_1], [\,] +\!\!+ [a_1] +\!\!+ [a_2], \cdots$ is an inductive chain, and thus an inductive (not necessarily Peano) model for $e, e \oplus a_1, e \oplus a_1 \oplus a2, \cdots$.

Lemma 4.4 (Duality Lemma) *For all* $\oplus$ *and* $e_\oplus$, $(\oplus\!\not\rightarrow\! e_\oplus) = (\oplus_R\!\not\leftarrow\! e_\oplus) \circ$ *reverse, where* $a \oplus_R b = b \oplus a$.

4.3.3 Segments and Partitions

Bird introduces two segment operators, which find the prefixes and suffixes of a list:

$$\begin{aligned} inits\,[a_1, a_2, \ldots, a_n] &= [[], [a_1], [a_1, a_2], \cdots, [a_1, a_2, \ldots, a_{n-1}], [a_1, a_2, \ldots, a_n]] \\ tails\,[a_1, a_2, \ldots, a_n] &= [[a_1, a_2, \ldots, a_n], [a_2, \ldots, a_n], \cdots, [a_{n-1}, a_n], [a_n], []]. \end{aligned}$$

With these he defines operators like

$$segs \;=\; (+\!\!+/) \circ (tails^*) \circ inits$$

that yields a list containing all segments (contiguous sublists) of its argument list. Indeed, this mechanism provide a very powerful tool for program calculus in the theory.

In dual-$\mathcal{M}$-sets, these operators correspond to the notion of powerset, or precisely downward closure, of elements in an inductive chain. This is obviously a (topped) chain by itself. Most of all, since every chain is a distributive lattice, more algebraic structure can be gained from monoids by introducing meet or join operations. Morphisms would then preserve lattice structure while simultaneously remaining monoid homomorphisms. Notice, in our construction of the modeling dual-$\mathcal{M}$-sets, this is the case where elements of β are $< \{\alpha\}^*, +\!\!+, [\,] >$. The notion of downward closure and its dual, upward closure, and the induced lattice structure ask for further exploitation. It is the perspective from dual-$\mathcal{M}$-sets that gives this insight.

4.3.4 Homomorphism, Recursion, Directed Reduction

When the operator, $\oplus$, is not even associative, and so not a binary homomorphism, we have to use the directed version of the reduction operator $\oplus/$, namely,

handling, where errors have to be propagated properly. This understanding seems more precise than that given by the comments in [Bir87], which should be taken as an advantage of the dual-$\mathcal{M}$-set framework.

4.3.2 Selection and Indeterminacy

Bird argues that many optimization computations require a general selection operator, defined as:

$$\begin{aligned} a \downarrow_f b &= a, \quad if \quad f(a) < f(b) \\ &= b, \quad if \quad f(a) > f(b) \end{aligned}$$

A problem occurs when $f(a) = f(b)$ while $a \neq b$: what is the result, a or b? In Bird's account, we must make a consistent *choice* among a and b, with the restrictions being that $\downarrow_f$ must satisfy specific properties (associativity, idempotence, commutativity) and a general minimality law

$$f \circ (\downarrow_f /) = (\downarrow_{id} /) \circ (f^*),$$

where $\downarrow_{id}$ is the 'minimum' operator over lists. The minimization operator μ of recursion theory has something of a similar nature, in that it returns not the function value but the satisfying argument.

As in the theory of lists, it is unlikely that dual-$\mathcal{M}$-sets can capture selection as defined by Bird without complicating the theory, as its functional basis does not really provide the relational flavor of nondeterminism. Various alternative accounts of selection could be captured, however.

semigroup into a monoid.

We can thus reformulate the following operators in [Bir87]:

operator		*reduction*	*base monoid*
sum	$=$	$+/$	$< N, +, 0 >$
$product$	$=$	$\times/$	$< N, \times, 1 >$
$flatten$	$=$	$+\!\!+/$	$< list(\alpha), +\!\!+, [\,] >$
all	$=$	$\wedge/$	$< (\alpha \rightarrow Bool), \wedge, true >$
$some$	$=$	$\vee/$	$< (\alpha \rightarrow Bool), \vee, false >$
min	$=$	$\downarrow/$	$< Z, min, +\infty >$
max	$=$	$\uparrow/$	$< Z, max, -\infty >$

If we employ the standard natural ordering on the natural numbers $\mathcal{N}$, we will have $n \leq_d m$ (or $m \leq n$) iff $min(n, m) = m$, $\forall m, n \in (\mathcal{N} \cup \{\omega\})$. Notice then, no chain of $< \mathcal{N} \cup \{\omega\}, min(\cdot, \cdot), \omega >$ topped with ω can be a Peano model, as no compatible successor operations are injective on the point ω. This is an instance where the notion of naturally induces ordering could be more general than inductive chain, while mathematical induction is basically all we need for computable functions over lists.

Also, ordinary inductive programming by patterns or structures will 'fail' here, as no Peano model is available. We need to add a case for the value of ω, when composed with other inductive programs. This is a case similar to error

$< \{\alpha\}^*, +\!\!+, [\,] >$, *with its unique homomorphism, f, to* α, *we have the unique homomorphism from* $< \{\alpha\}^*, +\!\!+, [\,] >$ *to* β *via* $g \circ f$. *This is again the view of homomorphism mediated by the initial algebra :*

$$
\begin{tikzcd}
 & < \{\alpha\}^*, +\!\!+, [\,] > \arrow[dl, "f"'] \arrow[dr, "g\circ f"] & \\
< \alpha, \oplus, e_\alpha > \arrow[rr, "g"] & & < \beta, \otimes, e_\beta >
\end{tikzcd}
$$

4.3 Outside the Ideal World

So far, we have explored *Computable Functions Over Lists* in an interesting, ideal and powerful universe: **Monoids**. This is indeed a computationally friendly world, as far as homomorphisms are concerned. However, we cannot limit ourselves to this ideal world, as it obviously does not exhaust the content of *Computable Functions Over Lists*. Altogether, three other significant kinds of operator are proposed in [Bir87], and we discuss these individually.

4.3.1 Fictitious Values

When reduction operator is applied to an ordinary binary associative operator, for instance, $min(\cdot,\cdot)/$, the value $min(\cdot,\cdot)/[\,]$ become undefined. In this case, the usual domain for $min(\cdot,\cdot)$ — say the natural numbers $\mathcal{N}$ — can be adjoined with the element $min(\cdot,\cdot)/[\,]$, usually denoted as ∞ or ω.

Notice ω cannot be any element in its original domain $\mathcal{N}$. Further, we notice that $< \mathcal{N}, min(\cdot,\cdot) >$ is a semigroup. In fact, the introduction of 'fictitious values' is just the application of the embedding lemma, Lemma 3.1, which embeds a

Lemma 4.2 (Promotion Lemmas)

$$
\begin{aligned}
&* \textit{ promotion:} & f^* \circ (+\!\!+/) &= (+\!\!+/) \circ (f^{**}) \\
&\triangleleft \textit{ promotion:} & (p\triangleleft) \circ (+\!\!+/) &= (+\!\!+/) \circ (p\triangleleft^*) \\
&/ \textit{ promotion:} & (\oplus/) \circ (+\!\!+/) &= (\oplus/) \circ (\oplus/^*)
\end{aligned}
$$

Proof 4.2 *First, it is easy to check* $(p\triangleleft)$ *is a homomorphism preserving* $[\,]$ *and* $+\!\!+$. *So, there exists an* f *such that* $(p\triangleleft) = (+\!\!+/) \circ f^*$ *by the Homomorphism Lemma. Third, we do equational reasoning as follows :*

$$
\begin{aligned}
(p\triangleleft) \circ (+\!\!+/) &= (+\!\!+/) \circ f^* \circ (+\!\!+/) && \textit{Def. from 2nd Step.} \\
&= (+\!\!+/) \circ (+\!\!+/) \circ (f^*)^* && +\!\!+/ \textit{ promotion.} \\
&= (+\!\!+/) \circ (+\!\!+/)^* \circ (f^*)^* && +\!\!+/ \textit{ promotion.} \\
&= (+\!\!+/) \circ ((+\!\!+/) \circ f^*)^* && (\cdot)^* \textit{ law.} \\
&= (+\!\!+/) \circ (p\triangleleft)^* && \textit{Def. from 2nd Step.}
\end{aligned}
$$

$\square$

If both α and β are monoids, we have a result which is called promotion in [Mal89]. In our framework, this may better be called Interface Compatibility.

Example 4.2 *Let both* α *and* β *are monoids, given by* $< \alpha, \oplus, e_\alpha >$ *and* $< \beta, \otimes, e_\beta >$. *Furthermore, let* β *be the homomorphic image of* α *via* g. *Now, from the initiality of*

must be of homogeneous or parameterized type (for instance, a list of integers). Our proof, while localized to α and β, is a uniform proof, applicable to any α and β. Further, α and β need not be disjoint sets, or be monoids. This is particularly useful when we consider dual monoid structures on the same carrier. For instance, we can choose the carrier of natural numbers N, and consider the monoids $< N, +, 0 >$ and $< N, \times, 1 >$ as discussed in previous chapter. In practice, we can have lists of heterogeneous types, $[1, 2, a, 3.4, \cdots]$ and nested lists, $[1, [1], [[a]], [c]]$. As long as the base mapping, f, and interpretation map, $\oplus/$, in the diagram, are semantically or computational meaningful, the whole computation is meaningful too. In fact, they are pretty much isolated, in the sense being captured in the monoid M of homomorphisms only after being raised by $(\cdot)*$.

Furthermore, consider the function $length()$ with $f(\alpha) = 1$, the generator of natural numbers, where $\oplus$ is arithmetic addition and $\beta = \mathcal{N}$. The connection between binary homomorphisms and mathematical induction then become lucid also.

Example 4.1 *For any f, we have $f* = +\!\!+/ \circ (g*)$, where $g(a) = [f(a)]$. Similarly, $(p\triangleleft) = +\!\!+/ \circ (h*)$, where $h(a) = [a]$, if $p(a)$; $[]$ otherwise.*

The Homomorphism Lemma also provides a kind of proof paradigm for equality of functional terms in three steps. The first step is to establish one side as a homomorphism, the second is to hypothesize an intermediate function using the Lemma, and the third step is to manipulate the formula by laws of functors, natural transformation, and previously established laws. For example, we can prove the $\triangleleft$-promotion lemma, given the other two promotion lemmas.

$$
\begin{tikzcd}
< \{\alpha\}^*, +\!\!+, [\,] > \arrow[r, "f^*"] & < \{\beta\}^*, +\!\!+, [\,] > \arrow[d, "(\oplus/)"] \\
< \alpha, \odot, e_\alpha > \arrow[r, "f"] & < \beta, \oplus, e_\beta > \\
\Sigma = \alpha \cup \beta \arrow[ur, "g"] &
\end{tikzcd}
$$

Notice we do not really require α and β to be monoids. However, the unique homomorphism for interpreting $+\!\!+$ as $\oplus$ on β introduces (or imposes) such an algebraic structure.

We have completed our modeling of Theory of Lists in the dual-$\mathcal{M}$-set, $<< M, \circ, Id >, \oslash, < \Sigma^*, +\!\!+, [\,] >>$, where $\oslash$ is the ordinary function application. $\square$

From the proof above, we immediately obtain the Homomorphism Lemma, by β being a monoid, and by Theorem 2.6 (Representability) and the uniqueness of homomorphism from an initial algebra :

Lemma 4.1 (Homomorphism Lemma) *A function $h :< list(\alpha), +\!\!+, [\,] > \to < \beta, \oplus, e >$ is a homomorphism with respect to $+\!\!+$ if and only if $h = (\oplus/) \circ f^*$ for some unique function $f : \alpha \to \beta$.*

It may seem easier and more general to have a categorical proof of the lemma as an instance of adjunction. However, the categorical proof imposes a type constraint implicit in the slogan that *types are objects*. As a result, the lists

Claim 4.1 *The calculus in Bird's "ideal world" Theory of Lists (with concatenation ($+\!\!+$), map $((\cdot)^*)$, filter $((\cdot)\triangleleft)$, reduction $((\cdot)/)$, the* Homomorphism Lemma *and* Promotion Lemma*, is expressible directly (and completely) as a dual-$\mathcal{M}$-set.*

Proof 4.1 *We now construct the modeling dual-$\mathcal{M}$-set. As mentioned, the definition of map $(map((\cdot)^*)$ and filter $((\cdot)\triangleleft)$ immediately gives us that these are all homomorphisms. The set of endo-morphisms is a monoid, $< M, \circ, Id >$, where $\circ$ is function composition and Id is the (polymorphic) identity mapping. We need to consider what would be the corresponding $\mathcal{M}$-set. The Homomorphism Lemma suggests it would be the initial algebra generated from $\alpha \cup \beta$, namely, $< \Sigma^*, \bullet, \Lambda >$, where $\Sigma = \alpha \cup \beta$. For simplicity, we assume α and β are disjoint for now. By the universal extension property for initial algebra, Theorem 2.5 and also Theorem 2.8, a mapping f (resp. g) from $\Sigma to \alpha$ (resp. β), can be uniquely extended to a homomorphism $f*$ (resp. $g*$) from Σ^* to $list(\alpha)$ (resp. $list(\beta)$). Then $< \{\alpha\}^*, \bullet, \Lambda >$ and $< \{\beta\}^*, \bullet, \Lambda >$ are obviously submonoids of $< \Sigma^*, \bullet, \Lambda >$, and thus monoids. Henceforth, we will use more conventional notations for lists, replacing $\bullet$ with $+\!\!+$, and Λ with $[\,]$. Finally, we notice that the reduction $((\cdot)/)$ is applied only as an outermost homomorphism, if not enclosed by the map operator in which case they are already in the monoids of homomorphisms M. So, while not directly incorporated into the dual-$\mathcal{M}$-set framework, it is only a peripheral interpretation mapping, as an interface map, or precisely, the unique (semantic) interpretation homomorphism from the initial algebra $< \Sigma^*, +\!\!+, [\,] >$. All the main interesting computation are encapsulated in the monoid of homomorphisms, M, on the initial algebra generated from the collections of raw data, α and β, etc.*

This is pertinent to our view that the initial algebra mediates all the semantic homomorphisms, (c.f. Figure 2.1.1), as follows, with $f = g\downarrow_\alpha$:

will argue below that the framework of dual-$\mathcal{M}$-set provides such a setting.

4.2 Inside the Ideal World

The theory of lists can be divided into two parts. The first part enjoys great simplicity and elegance, and satisfies many laws. Thus this part is termed as residing "inside the ideal world" of computable functions over lists. The rest is of less theoretic uniformity, though it could be more important and useful, and thus described as "outside the ideal world".

Here we first present the modeling of the ideal part in the dual-$\mathcal{M}$-set, in the following claims. In a later section, we introduce more morphisms into the dual-$\mathcal{M}$-set to show the part outside the ideal world can also be properly captured.

---

specifications :

$$
\begin{array}{llllll}
1) & (\cdot)^{*} & : & (\alpha \to \beta) & \to & (list(\alpha) \to list(\beta)) \\
2) & f_1 & : & \alpha \to \beta & & \\
3) & f_1 * & : & list(\alpha) & \to & list(\beta) \\
4) & f_2 & : & \beta \to \gamma & & \\
5) & f_2 * & : & list(\beta) & \to & list(\gamma) \\
6) & (\cdot) \circ (\cdot) & : & (\beta \to \gamma) \times (\alpha \to \beta) & \to & (\alpha \to \gamma) \\
7) & f_2 \circ f_1 & : & \alpha & \to & \gamma \\
8) & (f_2 *) \circ (f_1 *) & : & list(\alpha) & \to & list(\gamma) \\
9) & (f_2 \circ f_1) * & : & list(\alpha) & \to & list(\gamma)
\end{array}
$$

It is easy to see $()*$ is a functor, preserving morphism composition and identity.

phism (structure preservation), lifted from ordinary functions by these higher order operators, runs strong here. Indeed, this is a simple instance of dual-$\mathcal{M}$-sets where $\mathcal{M}$ consists of only (endo-)homomorphisms over a (syntactic) initial algebra, having a unique homomorphic interpretation on a semantic domain (α). Yet homomorphism is not the only theme present.

In the past few years, there has been increasing interest in finding a satisfying formal setting for these operators, particularly as they are finding their way into parallel programming languages [ABK89], database systems [BBN91, BS91, PSV92] and transformational program construction [Bir92, BM92].

Many people have argued that category theory is a promising setting for reconstructing the theory of lists, given its higher order nature. In particular, the above operators can easily seen to follow the categorical notions of functor, natural transformation, and adjunction.[1]

Spivey presented such a preliminary category theoretical analysis in [Spi89], covering the 'ideal' (homomorphic) fragment of the theory. However, Spivey's account does not cover the case of filters, and most significantly, does not cover directed reduction. These are very useful functions with slight differences from structure preserving homomorphisms. In fact, Bird has argued that these function play important roles in developing programs from specifications, at least for greedy algorithms [Bir92]. A completely satisfactory setting has been elusive. We

[1]The mental translation of perspective to the categorical view is pretty obvious intuitively. First we consider all involved data as objects, with type-as-sets objects. If we consider homomorphism as morphism, so the composition of homomorphisms is closed and identity mapping serves as the unit, then we immediately get a category. So, if we lay down the type specifications of these operators, currying n-ary functions into unary ones, it is routine to verify axioms of categorical notions of functor ($()*$, $()\triangleleft$), natural transformation ($()/$) and adjunction ($()*, \mathcal{U}, [\cdot]$), where $\mathcal{U}$ is a forgetful functor, an adjoint of $()*$, and $[\cdot]$ is the unit, a natural transformation mapping an object α to $list(\alpha)$).

For instance, if $f_1 : \alpha \rightarrow \beta$ and $f_2 : \beta \rightarrow \gamma$ are arbitrary functions, we have the following type

concatenation ($+\!\!+$):	$[1,2]+\!\!+[3,4] \;=\; [1,2,3,4]$	
map ($*$):	$succ * [1,2,3,4] \;=\; [2,3,4,5]$	
	$even * [1,2,3,4] \;=\; [false, true, false, true]$	
filter ($\triangleleft$):	$even \triangleleft [1,2,3,4] \;=\; [2,4]$	
reduction ($/$):	$+/[1,2,3,4] \;=\; 10$	

Table 4.1: Bird-Meertens Operators

With these one can derive various laws:

$$\begin{aligned}
f * (x+\!\!+y) &= (f * x)+\!\!+(f * y) \\
p \triangleleft (x+\!\!+y) &= (p \triangleleft x)+\!\!+(p \triangleleft y) \\
p \triangleleft (p \triangleleft x) &= p \triangleleft x \\
(p\triangleleft) \circ (f*) &= (f*) \circ ((p \circ f)\triangleleft) \quad \text{(where `}\circ\text{' is composition)} \\
\oplus/(x+\!\!+y) &= (\oplus/x) \oplus (\oplus/y).
\end{aligned}$$

The notation may become more suggestive with parentheses, as in for example $((succ)*)([1,2,3,4]) = [2,3,4,5]$. Notice that the operators are higher order, as they map unary functions to unary functions. Furthermore, the laws listed obviously show that $f*$, $p\triangleleft$ are endo-(homo)morphisms of a monoid, $< \mathcal{T}, +\!\!+, [\,] >$, where $\mathcal{T}$ is the domain of x and y. The reduction apply to binary operators, and return also (binary) homomorphisms too. Clearly the theme of homomor-

CHAPTER 4

Reconstructing the Bird-Meertens Formalism

The Theory of Lists developed by Richard Bird [Bir87] is a system of notation and a calculus for specifying and manipulating what Bird calls *Computable Functions Over Lists*. It organizes established results for list operators ('map' and 'APL reduction') developed since the 1960's (c.f. [Bur75]) into a more complete system of notation in which one can state many basic laws.

To get perspective on the theory, it is helpful to analyze the system from the level of categories of objects, namely monoids and morphisms (homomorphisms in particular, and also monotone morphisms, as we will see). On lists, the interaction of algebraic homomorphisms and order-theoretical monotonic functions, precisely the morphisms between inductive chains, is particularly interesting. We can apply dual-$\mathcal{M}$-sets to arrive at a satisfactory account of what is meant by Bird's notion of *Computable Functions Over Lists*.

4.1 Summary Of Theory of Lists

Below we will investigate individually the operators introduced by Bird. To summarize the flavor of his theory, however, note that Bird defines a variety of operators including:

(b). For an ω-chain of cardinality $n \leq \omega$, $< P_c = \{p_i \mid 0 \leq n\}, \leq_{P_c} >$ we can simply define an endomorphism, ρ, from P_c to P_c, as $\rho(p_i) = p_{i+1}$. By the Recursion Theorem (Theorem 2.11), $< P_c, \rho, p_0 >$ is a homomorphic image of a definition by mathematical induction from a Peano model. $\square$

Thus, morphisms defined by mathematical induction are the same as morphisms from an ω-chain of cardinality ω to another ω-chain. Obviously, these morphisms are monotone.

Now we can relate the homomorphisms on monoids with mathematical induction via induction chains. Notice that monoids are algebras having a single binary operation, while induction chains have a single unary operation. A connection follows from the isomorphism between the monoid generated with singleton alphabet ($\{x\}^*$) and the induced natural ordering $n \leq m$ if and only if $x^n \leq x^m$, $\forall m, n \in \mathcal{N}$. In cases where more than one generator is used (such as applications in lists or words from an alphabet Σ) the identification can still prevail, since the proof above is on arbitrary (ω-) inductive chains. Namely, it is uniform for any (ω-) inductive chain, in the sense that it is universally generalizable to any sets of inductive chains. In practice, for lists and words, we do have such uniformity when sets of inductive chains are considered.

This completes the embodiment of both homomorphism and (mathematical) induction as a basis of the dual-$\mathcal{M}$-set framework, via the notions of ω-chain and induction chain. Of course, the definition of naturally induced partial ordering admits more than just induction chains. This generality will permit exploitation in further applications.

Definition 3.6 (Chain) *A sub-poset, $< P_c, \leq >$, of a poset $< P, \leq >$, where $P_c \subseteq P$, is called a chain (a 'simply' or 'totally' ordered subset) if $\forall x, y \in P_c$, either $x \leq y$ or $y \leq x$.*

Definition 3.7 (ω-Chain) *A chain $< P_c, \leq >$ of a poset $< P, \leq >$ is called an ω-chain, if the cardinality of P_c is $\leq \omega$.*

Definition 3.8 (Inductive Chain) *An inductive chain on a poset $< P, \leq >$ is an ω-chain from the 'cover' relation of $\leq$. More precisely, $p_0, p_1, \cdots, p_n$ $(n < \omega)$ is an inductive chain if p_{i+1} covers p_i, $\forall i. 0 \leq i \leq n-1$.*

It is easy to see that the usual arithmetic ordering is an inductive chain, which formally represented as $0 < succ(0) < \cdots < succ^i(0) < \cdots$. As discussed previously, definition by mathematical induction is exactly the homomorphism fron this inductive chain whose image are either Peano models or Inductive models. Indeed we have this simple theorem :

Theorem 3.2 **(a).** *Any Peano model $< S, \rho, e >$ can be ordered as an inductive chain.*

(b). *Any ω-chain is an image, defined by mathematical induction, of a Peano model.*

Proof 3.5 *(a). By definition, we have $e \neq \rho(X), \forall X \in S$ and ρ is injective, so $\rho^i(e) \neq \rho^j(e), \forall i \neq j$. So we have an ω-chain, $e < \rho^i(e) < \rho^j(e)$, $\forall i, j, 0 < i < j$. Furthermore, it is an inductive chain. Otherwise, there will be some k and x such that $\rho^k(e) < x < \rho^{k+1}(e)$. Since S is an Peano model, x must be $\rho^n(e)$ for some n, $n \neq k, n \neq (k+1)$, which is impossible.*

Claim 3.1 *When the natural order* $\leq$ *induced from* $< M', 0, \oplus >$ *exists, the enlargement of the dual-*$\mathcal{M}$*-set* $<< M, 1, \bullet >, \oslash, < M', 0, \oplus >>$ *with monotone functions respecting* $\leq$ *incorporates order-preservation as an extension of structure-preservation.*

Proof 3.4 *The claim is justified first by the fact that structure preserving morphisms (homomorphisms) are captured by* $< M, 1, \bullet >$*. Second, monotone morphisms respecting the naturally induced partial ordering subsume structure preserving morphisms (respecting* $\oplus$*). The proof of monotonicity, for any homomorphism* $h(\cdot)$ *respecting* $\oplus$*, is :*

- $h(z) \geq h(0), \ \ \forall z \in \alpha.$
- *If* $x \leq y$*, then* $x \oplus z = y$*. So* $h(x) \oplus h(z) = h(x \oplus z) = h(y)$*, and thus* $h(x) \leq h(y)\ \forall x, y \in \alpha$*.*

Thus, enlarging the class of acceptable morphisms to include monotone morphisms is compatible with the original (homo-)morphism composition. □

3.3.1 Homomorphisms and Mathematical Induction

As mentioned, the framework of dual-$\mathcal{M}$-sets seeks to capture homomorphisms and mathematical induction as a basis for a rigid body metaphor of software. To fulfill that, we can consider a subset of a naturally induced partial ordering which corresponds to induction. The notion needed, called an inductive chain, is pretty simple indeed.

Definition 3.5 (Cover) *We say 'a covers b' in a poset* $< P, \leq >$*, if* $a > b$*, but that there is no* $x \in P$ *such that* $a > x > b$*.*

Example 3.11 *The monoid* $< B, \vee, 0 >$ *of a Boolean algebra* B *(or of a general lattice with least element 0) is naturally ordered:* $b_1 \leq b_2$ *if and only if* $b_1 \vee b_2 = b_2$*.*

The reverse-naturally induced orderings in the above examples are also naturally induced orderings, due to the commutativity of their specific monoid operators. This is not the case for the following examples.

Example 3.12 *The monoid of words* $< \Sigma^*, \bullet, \Lambda >$*, from an alphabet* Σ*, is naturally ordered by the usual lexicographic ordering with respect to an alphabetic ordering* $\leq_\Sigma$*:*

$$w_1 \leq w_2 \ (wrt \ \ \leq_\Sigma) \ \ \textit{if and only if} \ \exists w_3 . w_1 \bullet w_3 = w_2.$$

This is the common prefix ordering modulo alphabet ordering.

Example 3.13 *The monoid* $< 2^{\Sigma^*}, \bullet, \{\Lambda\} >$*, of the common (closed) semiring of regular sets over an alphabet* Σ*,* $< 2^{\Sigma^*}, +, \bullet, ()^*, \emptyset, \{\Lambda\} >$*, is naturally ordered by a kind of prefix ordering:*

$$L_1 \leq L_2 \ \textit{if and only if} \ \exists L_3, L_1 \bullet L_3 = L_2.$$

This corresponds to a quotient ordering between formal languages as that of divisibility in natural numbers. Notice that it is not the same as the natural ordering as subset inclusion as it would be, if we took the monoid to be $< 2^{\Sigma^*}, +, \emptyset >$*.*

By this notion of naturally induced ordering, we contain an important interaction between structure and order, as in the following claim. Recall that a monotone mapping preserves the partial ordering from the domain to the range, namely, if $x \leq_1 y$ then $f(x) \leq_2 f(y), \forall x, y$.

structure, and allow us to properly extend the class of structure-preserving homomorphisms to include order-preserving morphisms.

Definition 3.4 (Naturally Induced Order) *A partial order* $\leq$ *is called* naturally induced *from a monoid* $< M, 0, \oplus >$*, if 0 is a least element for* $\leq$*, and*

$$\forall x, y \in \alpha. \quad x \leq y \ \ \textit{if and only if} \ \ \exists z \in \alpha.\, x \oplus z = y.$$

It is called reverse-naturally induced*, if 0 is the least element for* $\leq$*, and*

$$\forall x, y \in \alpha. \quad x \leq y \quad \textit{if and only if} \quad \exists z \in \alpha. \ \ z \oplus x = y.$$

Finally, it is dual-naturally induced*, if*

$$x \leq_d y \ \ \textit{if and only if}\ y \leq x$$

where $\leq$ *is either naturally or reverse-naturally induced from* $< M, \oplus, 0 >$*.*

The requirement that the special constant 0 be the least element is required when considering order-preserving homomorphisms. In general, however, this could be relaxed. Notice that the naturally induced order is defined directly upon the underlying set α, rather than via subset inclusion on the power set of α as is common in denotational semantics.

Example 3.9 *The natural numbers* $< N, +, 0 >$ *are naturally ordered by the usual (total) arithmetic ordering:* $m \leq n$ *if and only if* $\exists k.m + k = n$*.*

Example 3.10 *The positive natural numbers* $< N, \times, 1 >$ *are naturally ordered by the divisibility ordering:* $m \leq n$ *if only if* $\exists k, m \times k = n$*. This is a proper (non-total) partial ordering.*

$\square$

Because a direct product of monoids is a monoid, whenever α is a monoid, then $< \mathcal{M}, \oslash', \beta \times \alpha >$ is a dual-$\mathcal{M}$-set.

3.3 Partial Ordering in Dual-$\mathcal{M}$-Sets

Algebraic structures are sets, called carriers, equipped with internal operations. When considering mappings between algebras, it is natural to emphasize homomorphisms — 'structure preserving' mappings that keep algebraic operations in correspondence.

On the other hand, it is also common to consider relations between elements of the carriers, particularly partial ordering relations, and emphasize 'order preserving' mappings. Recall that a partial ordering on a set α, is a binary relation (a subset of $\alpha \times \alpha$) which is reflexive, antisymmetric and transitive. Computations are often characterized in terms of the constructor complexity of a representation, rather than of arbitrary concrete semantic algebra satisfying a set of equations.

Order preserving mappings are not always easy to reconcile with structure preserving mappings. Complicated interactions can arise between the two. This is illustrated in the Appendix, studying various cases that occur with mappings on lattices. However, well-behaved instances do exist. An important such instance is provided by mathematical induction, since '$\leq$' and '$+$' can be mutually defined on the natural numbers: $m \leq n$ if only if $\exists k . n = m + k$. A similar instance arises in the practice of programming by patterns.

In the dual-$\mathcal{M}$-set framework, we will be interested in certain naturally induced orderings. These natural orderings both are compatible with the monoid

framework, as the following lemma shows.

Lemma 3.2 (Accumulation Lemma) *Given a monoid $\mathcal{M} = < \beta, \oplus, e >$, if α is a $\mathcal{M}$-set with associated map $\oslash$, then $< \mathcal{M}, \oslash', \beta \times \alpha >$ is also a $\mathcal{M}$-set with associated map $\oslash'$, where $\beta_i \oslash' (\beta_k, \alpha_j) = (\beta_i \oplus \beta_k, \beta_i \oslash \alpha_j)$.*

Proof 3.3 *Given $\mathcal{M} = < \beta, \oplus, e >$ is a monoid and α an $\mathcal{M}$-set, we only have to verify followings two properties : let $\beta_k, x, y \in \mathcal{M}$, and $\alpha_j \in \alpha$,*

Identity :

$$
\begin{aligned}
& e \oslash' (\beta_k, \alpha_j) \\
= \quad & (e \oplus \beta_k, e \oslash \alpha_j) && \textit{Def. of } \oslash' \\
= \quad & (\beta_k, e \oslash \alpha_j) && \textit{Id. of } \mathcal{M} \\
= \quad & (\beta_k, \alpha_j) && \alpha \textit{ is a } \mathcal{M}\textit{-set}
\end{aligned}
$$

Composition :

$$
\begin{aligned}
& (x \oplus y) \oslash' (\beta_k, \alpha_j) \\
= \quad & ((x \oplus y) \oplus \beta_k, (x \oplus y) \oslash \alpha_j) && \textit{Def. of } \oslash' \\
= \quad & (x \oplus (y \oplus \beta_k), x \oslash (y \oslash \alpha_j)) && \textit{Assoc. of } \mathcal{M}\textit{, and Def. of } \oslash \\
= \quad & x \oslash' (y \oplus \beta_k, y \oslash \alpha_j) && \textit{Def. of } \oslash' \\
= \quad & x \oslash' (y \oslash' (\beta_k, \alpha_j)) && \textit{Def. of } \oslash'
\end{aligned}
$$

Notice that :

- *The enrichment is indeed a result of type coercion. Namely, elements of a set are coerced into elements of a boolean vector space. The latter is equipped with addition, multiplication and complement, and is much richer than the given set of (algebraically unstructured) raw data.*

- *The matrix representation of mappings naturally imposes a monoid structure for the composition of mappings, rather than simply a collection of unrelated mappings, as originally isolated m and n from the monoid of the function space $A \rightarrow A$. The monoid structure of matrices is inherited from the dual-$\mathcal{M}$-set, and enriched as a result of enrichment of the $\mathcal{M}$-set A. Further, the matrix representation, though partly inherited, makes explicit the underlying universe of m and n.*

- *The mapping is naturally extended to other compound values, for example, $< 1, 1 >$ in A.*

- *New mappings are added to the function space, for example, with non-zero entries in positions differing from those in m, n, such as with :* $\begin{bmatrix} 0 & n_{12} \\ n_{21} & m_{22} \end{bmatrix}$.

- *The value set is now a (boolean) vector space, and function composition becomes matrix multiplication, which admits more properties for calculus. This provides a better setting for algebraic and arithmetic about programs, similar to the idea of algorithmics as transformational programming [Mee86].*

While the operators are supposed to be non-destructive actions, we can register and accumulate their side-effects. This can also be done in the dual-$\mathcal{M}$-set

isomorphism. $\mathcal{L}$ is a homomorphism, as $\mathcal{L}(0) = 0_L$ and $\mathcal{L}(m \oplus n) = (m \oplus n)_L = m_L \circ n_L$. For injectivity, we have, if $m_L = n_L$, i.e., $\forall x \in M$, $m_L(x) = n_L(x)$, which must be true for $x = 0$, namely, $m = m_L(0) = n_L(0) = n$. $\square$

We can also give a non-isomorphic embedding, representing an $\mathcal{M}$-set in a boolean vector space. This is an instance where the rigid body is cast into a more richly structured universe (one equipped with boolean sum, product and complement) beyond the raw data representable with a monoid. In general, using the constant function, $\lambda x.a$, $\forall a \in \mathcal{A}$, we can also embed both components of dual $\mathcal{M}$-set in the function space $\mathcal{A} \to \mathcal{A}$. The left translations used in the proof above can be seen as linear transformations on a monoid.

Example 3.8 (Nontrivial Non-Isomorphic Representation) *Now we give a more concrete example, illustrating advantages that result when raw data are represented in a richer coordinate system. Suppose we are given initially the following raw data : an $\mathcal{M}$-set $A = \{a_1, a_2\}$, and two elements from the monoid of transformations on A :*

$$m = \begin{bmatrix} a_1 & \mapsto & m_{11} \oslash a_1 \\ a_2 & \mapsto & m_{22} \oslash a_2 \end{bmatrix}, \quad n = \begin{bmatrix} a_1 & \mapsto & n_{12} \oslash a_2 \\ a_2 & \mapsto & n_{21} \oslash a_1 \end{bmatrix}.$$

Now if we embed A in its characteristic function space, namely, functions from A to a two-element set (naturally structured as a boolean algebra), $A \to Bool$. We have then :

$$a_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, a_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad m = \begin{bmatrix} m_{11} & 0 \\ 0 & m_{22} \end{bmatrix}, \quad n = \begin{bmatrix} 0 & n_{12} \\ n_{21} & 0 \end{bmatrix}$$

functions on α, where $\oslash$ is function application. $\square$

As mentioned, The notion of $\mathcal{M}$-set is to treat elements in M as operators acting externally on the $\mathcal{M}$-set $\mathcal{A}$, without regard to its internal algebraic structure. In this case, elements of M should be elements in the function space on $\mathcal{A}$. If further information about $\mathcal{A}$ is available, this information also applies to elements in the function space.

The function space is obvious a monoid, with composition as the associative operator and the map, $\lambda x.x$, as the identity. Although the natural numbers $< N, +, 0 >$ also form a monoid, a simple argument based on cardinality prevents an isomorphism between them. However, submonoids of function spaces may still be of interest, and then an isomorphism becomes possible. In fact, we do have such an isomorphism, as is shown by a version of Cayley's Isomorphism Theorem for monoids.

Theorem 3.1 (Cayley's Theorem) *Any monoid is isomorphic to a monoid of transformations.*

Proof 3.2 *For any $m \in M$, we define the left-translation map $m_L : x \mapsto m \oplus x$, $\forall x \in M$. In typed lambda notation, $m_L = \lambda x : M.(m \oplus x)$. First we establish that $< M_L, \circ, 0_L >$, where '$\circ$' is composition and $M_L = \{m_L \mid m \in M\} = \{\lambda x : M.(m \oplus x) \mid m \in M\}$ is a monoid of transformations. It is easy to see $0_L = \lambda x : M.x$ is the identity. We also have, for any $m_L, n_L \in M_L$, $(m_L \circ n_L)(x)$ $= m \oplus (n \oplus x) = (m \oplus n) \oplus x = (m \oplus n)_L(x)$, $\forall x \in M$, by the associativity of M. So $m_L \circ n_L = (m \oplus n)_L$, and M_L is is closed under associative composition.*

Next we prove that the mapping $\mathcal{L}(m) = m_L : m \mapsto \lambda x.(m \oplus x)$, or equivalently $\mathcal{L} = \lambda m : M.\lambda x : M.(m \oplus x)$, from $< M, \oplus, 0 >$ to $< M_L, \circ, o_L >$ is a monoid

morphisms, in which elements of M are functions and the operation $\bullet$ (ordinary function composition) is a combinator. Here one binary combinator ($\bullet$) and one constant combinator (the identity combinator) satisfy a set of algebraic laws (the monoid laws). Second, following the perspective on homomorphism and induction from the previous chapter, this definition considers both function space and object space at once. As functions define a monoid by the operation of function composition, this presents no problems. Furthermore, the set of endomorphisms (homomorphism with same domain and range) is also a monoid; see Theorem 2.10 (Monoid of Endomorphisms). Third, the word '*computable*' stresses not only our interest in effective functions, but also the importance of a common representation scheme for $m \in \mathcal{M}$ and $a \in \alpha$. In other words, the underlying sets M' of α and M of $\mathcal{M}$ are both *effectively representable* with a common set. This may deviate a little from the rigid body metaphor, as usually we would expect an underlying ring of values, e.g. the ring of real numbers, $\mathcal{R}$, and set the dual space as of type $\alpha \rightarrow \mathcal{R}$ and the class of homomorphisms as $Hom_{\{+,\times\}}[\alpha, \mathcal{R}]$. Our proposal here is of encoding, namely, assuming there is a one-to-one correspondence between $\mathcal{R}$ and α, whose inverse can be used as an interpretation mapping. Notice, this is the same as the treatment of generating sets X in $\mathcal{T}_\Sigma(X)$. This is also the approach used in modeling recursive functions in Turing machines. More about the embedding representation will be said after a few examples.

Example 3.6 $< M, \bullet, 1 >$ *can be the monoid of lists, where* $< M', \oplus, 0 >$ *is the monoid of list transductions,* 0 *is the identity function,* $\oplus$ *is function composition and, again,* $\oslash$ *is ordinary function application.* $\square$

Example 3.7 *If* α *is a vector space, then* $\mathcal{M}$ *can be the (dual) space of linear*

in formal power series [KS86], and path-related algorithms [Tar81]. We list the equations here :

$(i) \quad (ab) \oslash v = a \oslash (b \oslash v),$

$(ii) \quad (a+b) \oslash v = (a \oslash v) \oplus (b \oslash v),\ \ \textit{and}\ a \oslash (v \oplus u) = (a \oslash v) \oplus (a \oslash u),$

$(iii) \quad 1 \oslash v = v\ \ \textit{and}\ 0 \oslash v = 0,$

$(iv) \quad a \oslash 0 = 0.$

$\square$

3.2 Dual-$\mathcal{M}$-Sets

For our purpose, we specialize the notion of $\mathcal{M}$-set by essentially requiring both the monoid $\mathcal{M}$ and the $\mathcal{M}$-set $\mathcal{A}$ acted upon by $\mathcal{M}$ to be submonoids of the same monoid α. Furthermore, for computation, we would like to have a collection of terms that can represent both $\mathcal{M}$ and $\mathcal{A}$. Intuitively, this corresponds to taking into account the algebraic structure of the $\mathcal{M}$-set as the subject of actions by the monoid $\mathcal{M}$.

We formalize these intuitions as follows :

Definition 3.3 (dual-$\mathcal{M}$-set) *An $\mathcal{M}$-set $< \mathcal{M}, \oslash, \alpha >$ is a* dual-$\mathcal{M}$-set *if $\alpha =< M', \oplus, 0 >$ is a monoid, $\mathcal{M} =< M, \bullet, 1 >$ is the monoid of (*computable*) functions of type $\alpha \rightarrow \alpha$, and $\oslash$ is function application.*

A few remarks about this definition. First, this formalization obviously has a higher-order aspect. In particular, $< M, \bullet, 1 >$ is a combinator algebra of

Example 3.2 $< M, \bullet, 1 >$ *can be the monoid list*(α) *of lists, where* 1 *is the empty list,* $+\!\!+$ *is ordinary list concatenation, and* $(m \bullet n) = (n +\!\!+ m)$*, i.e., '*$\bullet$*' is* reversed *list concatenation. Let* $\mathcal{A}$ *be a* state space*, and* $\oslash$ *be a state transition function* $\mathcal{M} \times \mathcal{A} \to \mathcal{A}$ *whereby* $(n +\!\!+ m) \oslash a = (m \bullet n) \oslash a = (m \oslash (n \oslash a))$. Finite state machines *are thus easily modeled with* $\mathcal{M}$*-sets. If instead we let* $\mathcal{A}$ *be a set of (output list,state)-pairs and let* $\oslash$ *be a transduction operator*

$$m \oslash (p, s) \quad = \quad (\,(p +\!\!+ h(m, s)),\, \delta(m, s)\,)$$

where h *gives incremental output and* δ *gives a new state, then* finite state transducers *are modeled in the same way. Finally, if* δ *is trivial (or analogously, if the state space is trivial), we obtain*

$$m \oslash p \quad = \quad p +\!\!+ h(m).$$

Thus the state transductions become simple list homomorphisms h*, such that* $h(m \bullet n) = h(m) \bullet h(n)$*, or equivalently* $h(n +\!\!+ m) = h(n) +\!\!+ h(m)$*.* □

Example 3.3 *If* $\mathcal{A}$ *is an additive abelian group, then the* $\mathcal{M}$*-set is actually an* $\mathcal{M}$-module *provided that* $\mathcal{M}$ *is a ring, and it is a* vector space *provided that* $\mathcal{M}$ *is a field.* □

Example 3.4 *If the* $\mathcal{M}$*-set* $\mathcal{A}$ *is restricted to be a boolean algebra and* $\mathcal{M}$ *a regular algebra, then we have a* Dynamic Algebra *and its variety, as in [Pra90].*

□

Example 3.5 *If* $\mathcal{A}$ *is an additive monoid* $< A, \oplus, 0 >$*, then the* $\mathcal{M}$*-set is in fact a* $\mathcal{M}$*-semimodule provided* $\mathcal{M}$ *is a semiring, or precisely, a ring whose additive component is only a commutative monoid. This notion finds applications*

kinds of algebra. They are very common in computer science, appearing for example as $< N, +, 0 >$, $< N, \times, 1 >$, $< list(\alpha), +\!\!+, [\,] >$, $< Bool, \vee, \text{false} >$, $< Z \cup \{\infty\}, min, \infty >$, $< Z \cup \{-\infty\}, max, -\infty >$, $< SortedLists, Mergesort, [\,] >$, etc. Because of their frequent occurrence and simple structure, they are qualified to define rigid bodies of software.

Sometime the monoid identity element may not be required, giving a semigroup. Also, there could be trivial semigroups, such as the empty set. However, it is a simple fact that we can always embed a semigroup into a monoid.

Lemma 3.1 *Any semigroup, $< M, \oplus >$, can be embedded in a monoid $< M', \oplus, e >$.*

Proof 3.1 *If $< M, \oplus >$ is in fact a monoid already, then it has an identity element, and we just let e be that identity. Otherwise, take e not in M, define $M' = M \cup \{e\}$, and extend the definition of $\oplus$ on M to be an operator on M' by defining $e \oplus m = m = m \oplus e$, $\forall m \in M'$.* $\square$

Our applications require modeling both of a set of objects, and of the possible transformation (external morphisms) applicable to the objects and their composition. This leads us to what is commonly referred to as an $\mathcal{M}$-set.

Definition 3.2 ($\mathcal{M}$-set) *Given a monoid $\mathcal{M} = < M, \bullet, 1 >$, a $\mathcal{M}$-set is a set $\mathcal{A}$, together with a mapping $\oslash : M \times \mathcal{A} \rightarrow \mathcal{A}$, usually denoted as $(m, a) \mapsto ma$ (or $m \oslash a$ or $m(a)$), such that $1(a) = a$ and $(m \bullet n)(a) = m(n(a))$, for all $a \in \mathcal{A}$, $m, n \in \mathcal{M}$.*

Example 3.1 *$< M, \bullet, 1 >$ can be the monoid of functions $\mathcal{A} \rightarrow \mathcal{A}$, where 1 is the identity function, $\bullet$ is function composition, and $\oslash$ is function application.* $\square$

CHAPTER 3

Dual-$\mathcal{M}$-Sets

In this chapter we develop an algebraic structure called the dual-$\mathcal{M}$-set that occurs naturally in many contexts, and models many applications of interest to us. In its formal content, the model is intended to capture homomorphisms and induction, and allow further development of a rigid body metaphor for software. Roughly speaking, a dual-$\mathcal{M}$-set is a monoid together with its 'dual space' (the set of structure-preserving mappings from a monoid to itself, also called its endo-morphisms). The basic approach is to consider data as a monoid structured object, defining instances of rigid bodies. Programs are then operators or transformations or (endo-)morphisms of monoids which, with ordinary program composition, constitute a monoid too. As in other fields, duality is both a powerful and useful concept.

3.1 Monoids and $\mathcal{M}$-Sets

Definition 3.1 (Monoid) *A monoid* $\mathcal{M} = < M, \bullet, 1 >$ *is a set* M*, together with an associative operator* $\bullet : M \times M \rightarrow M$*, for which 1 is an identity. That is,* $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ *and* $1 \bullet x = x \bullet 1 = x$ *for all* x*,* y*,* z *in* M*.*

Intuitively, a monoid is a set equipped with an internal associative operator together with a special identity element. Monoids are one of the most basic

is not so, due to the use of parameters $\vec{x}$, not only as numbers but also numbers of different types than that of y. This pint will be made more clear later. We remark that, the exponentiation function $Exp(m,n) = n^m$ is definable in limited primitive recursion, with $\phi_m = 1$, and $\psi_m(y) = m \times y$ substituted in the form $LPR(\phi, \psi, 1)$. In other word, $Exp = LPR(1, \lambda m.\lambda y.m \times y, succ(0))$. It is also a known fact that exponentiation function is not definable by a homomorphism from a Peano number [Hen60]. For instance, in a modulo 3 number system, we have $0 \equiv 3 \equiv 1+2$, but $1 \equiv 2^0 \equiv 2^3 \not\equiv (2^1 + 2^2) mod 3 \equiv 0$.

Conversely, it is easy to see any homomorphism is definable in the form of limited primitive recursion.

$$\begin{aligned} \rho(0) &= 0^\sigma \\ \rho(succ^n(x)) &= succ^\sigma(\rho(succ^{n-1}(x)) \end{aligned}$$

with 0^σ as $\phi_{\vec{x}}$ and $succ^\sigma$ as $\psi_{\vec{x}}$ in $LPR(\phi, \psi, 0)$.

So we have this simple observation as a theorem :

Theorem 2.14 (Proper Iteration) *Limited primitive recursion properly contains the homomorphic images of the Peano number system.*

given $\phi(\vec{x})$ and $\psi(\vec{x}, \bullet)$, in notation, $LPR(\phi, \psi, n)$, as defined this limited form :

$$\begin{aligned} f(\vec{x}, 0) &= \phi(\vec{x}) \\ f(\vec{x}, succ(y)) &= \psi(\vec{x}, f(\vec{x}, y)) \end{aligned}$$

while, in general, for unlimited primitive recursion, the second clause is

$$f(\vec{x}, succ(y)) = \psi(\vec{x}, y, f(\vec{x}, y))$$

It is clear, in this form, the last argument $(succ(y))$ is used as an iterator (loop control variable for iterations), the evaluation of $\psi(\vec{x}, \bullet)$ only depends on $\phi(\bullet)$ and $\vec{x}$, yet independent of y. Alternatively, within the loop, the control variable can not be explicit used for the computation, as does in unlimited primitive recursion.

A few steps of expansion, together the replacement of parameters with indexing, namely, $\phi(\vec{x}, \bullet)$, will show :

$$\begin{aligned} f(\vec{x}, 0) &\equiv f_{\vec{x}}(0) \\ &= \phi(\vec{x}) \\ f(\vec{x}, succ(y)) &= \overbrace{\psi(\vec{x}, \cdots, \psi(\vec{x})}^{y+1}, \phi(\vec{x})) \cdots) \\ &\equiv \overbrace{\psi_{\vec{x}}(\cdots \psi_{\vec{x}}}^{y+1}(\phi_{\vec{x}}) \cdots) \\ &= f_{\vec{x}}(f(\vec{x}, y)). \end{aligned}$$

The set $\Psi = \{\psi_{\vec{x}}^n(\phi_{\vec{x}}) \mid n \in \omega\}$ is a Peano model. The function definable in limited primitive recursion is very close to a homomorphism, indexed by a n-tuple $\vec{x}$, from the natural number system $< \mathcal{N}, succ, 0 >$ to $< \Psi, \phi_{\vec{x}}, \psi_{\vec{x}} >$. This

Notice the function $f(x, y) = h_x(y)$ is the operation of ordinary addition '+' in the natural number system. Notice that the defined function is parameterized by an element, x, from N. Furthermore, the function defined has to follow the same inductive model, different from that for the Recursion Theorem. Combining these two theorems, we can start from a Peano model, using homomorphisms to define a function on an inductive image, and then use the Induction Definition Theorem to 'shift' with a parameter x. This process is indeed unique, as can be seen from the following theorem.

Theorem 2.13 ([Hen60]) *Let $\mathcal{N} =< N, 0_N, s_N() >$ be a Peano model and $\mathcal{M} =< M, 0_M, s_M() >$ an arbitrary model. Then $\mathcal{M}$ is a homomorphic image of $\mathcal{N}$ if and only if $\mathcal{M}$ is an inductive model.*

Thus, if we want to employ definition by mathematical induction, we should ensure that the homomorphic image has an inductive structure. This can be a correctness criterion for programs based on inductive programming.

2.3.4 Homomorphism vs. Iteration

Obvious, homomorphisms and definition by induction enjoy a kind of iteration. However, as a computation method, iteration is more powerful than homomorphism. In this section, we look at a limited form of primitive recursion, which embodies a common notion of nested iteration in programming. The proper containment of homomorphism within general iterations thus indicates a limitation of its computation power. This serves as a drive for extending beyond homomorphism, in particular, monotone functions.

Consider a limited form of primitive recursion for a (n+1)-ary function $f(\vec{x}, \bullet)$,

Notice $f(n+1) = \phi^n(\eta),\ \forall n \in \mathcal{N}$, with $\phi^0(x) = x$.

Example 2.1 *The recursion theorem fails for the non-Peano inductive model* $K =< \{0,1\}, 0, s() >$*, where* $s(x) = 1, \forall x \in K$*, and the function* $\phi(0) = 1$*,* $\phi(1) = 0$. $\square$

Example 2.2 *Both domains for the two functions,* silly *and* card*, are word algebras* $\mathcal{T}_\Sigma(\alpha)$*, not modulo the monoid congruence. Furthermore, they are not Peano models. For instance,* $[\,] +\!\!+ [a]$ *and* $[a]$ *are different elements and yet the successor operation has to assign them the same value, namely,* $S([\,] +\!\!+ [a]) = S([a])$*, which contradicts injectivity* **(P2)***. This is the reason they fail to be proper inductive definitions using mathematical induction.* $\square$

Example 2.3 *For any* $a_i \in \alpha$*, the system* $< \{[\,], [\,] +\!\!+ [a_i]\}, [\,], S() >$*, where* $S([\,]) = [\,] +\!\!+ [a_i]$ *and* $S([\,] +\!\!+ [a_i]) = [\,] +\!\!+ [a_i]$*, is a non-Peano inductive model. Also for any finite sequence,* $a_1, \cdots, a_n$ *from* α*, the model,* $< Fin(\alpha), [\,], S() >$*, where* $Fin(\alpha) = \{[\,]\} \cup \{[\,] +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_i] \mid 0 \leq i \leq n\}$*,* $S([\,]) = [\,] +\!\!+ [a_1]$*,* $S([\,] +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_i]) = [\,] +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_{i+1}]$*,* $S([\,] +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_n]) = [\,] +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_n]$*, is a non-Peano inductive model.* $\square$

Theorem 2.12 (Induction Definition [Hen60]) *In any inductive model* $< N, 0, S() >$*, and for any* $x \in N$*, there is a unique unary operation* h_x *on* N*, satisfying :*

1. $h_x(0) = x$,

2. $h_x(S(y)) = S(h_x(y)), n \in \mathcal{N}$.

by induction on the ranks of terms. Fourth, most importantly, it serves as sources of references for many ideas to be developed, for example, multiple signature and representations of the Peano model with and without equations, partial ordering and monotonicity and inductive chains, and sharing vs. copying for the addition and multiplication monoids.

Definition 2.7 (Peano Models and Inductive Models) *A model* $< N, 0, S() >$, *where* N *is a set, 0 an element of* N *and* $S()$ *an unary operation on* N, *is called Peano Model if it satisfies the following.*

P1. *For all* $x \in N$, $S(x) \neq 0$*;*

P2. *For all* $x, y \in N$, *if* $x \neq y$ *then* $S(x) \neq S(y)$*;*

P3. *If* G *is any subset of* N *such that (a)* $0 \in G$, *and (b) whenever* $x \in G$ *then also* $S(x) \in G$, *then* $G = N$.

The axiom **(P3)** is normally referred to as the axiom of mathematical induction. Inductive models are used to refer to those that satisfy only the axiom of mathematical induction **(P3)**. **(P2)** is equivalent to stating that $S()$ is injective (a one-to-one mapping).

Theorem 2.11 (Recursion Theorem [Hen60, Jac85]) *Let* S *be a set,* ϕ *a map of* S *into itself,* η *an element of* S. *Then there exists one and only one map (homomorphism)* f *from a Peano model* $< \mathcal{N}, 0, \lambda n.(n+1) >$ *(the natural numbers) to* S *such that*

1. $f(0) = \eta$,

2. $f(n+1) = \phi(f(n)), n \in \mathcal{N}$.

a clever programming technique for finding a desired homomorphism or demonstrating its nonexistence.

We should emphasize that not all programming problems are tractable with mathematical induction. As we know, induction is only a subset of primitive recursion. Of course, programs definable by mathematical induction comprise a very useful and significant segment of computable functions. In practice, we may mix structural with other value domains, and then factor out the parts of the program based on mathematical induction.

The subject and proofs of theorems here can in fact be seen as a special formal instance of last section, by taking the signature Σ to be $\{0, succ()\}$. The term algebra $\mathcal{T}_\Sigma$, which is initial by Theorem 2.7 (Uniqueness and Existence of Initial Algebra), is then a Peano model. Alternatively, we say a Peano model is initial among the class of Σ-algebra. Notice that this class is equation free. In fact, the inductive models are images of unique homomorphisms from $\mathcal{T}_\Sigma$ satisfying additional equations represented as congruence induced by the unique homomorphism. The mathematical induction is simply an instance of structural induction (or generalization the other way around), by Theorem 2.8, with X being the empty set and $Y = \{0\}$.

The reasons for including this part are manifold. First, it can serve as a comprehensive testimony of the power, methods and results of universal algebra. Second, the common natural number system is very useful and prevalent, not only to computer science but also many other fields. Third, the principle of Mathematical Induction for definition and proof, while subsumed by the more general Structural Induction, has actually served both at the object level and the meta level of the theory. For instance, the generation and definition of terms are

□

Notice that the mapping, called assignment, $X \to Y$, being injective is critical for the uniqueness. Otherwise, say, both x_1 and x_2 are mapped into y_1, and we can flip the extended homomorphism by interchanging x_1 and x_2. On the other hand, this restriction is not essential, as any mapping can be factored by a surjection followed by an injection. In other words, we can consider the smallest X for such mappings, by taking congruence, and start from the injection derived.

We next present two metatheorems about structures of a class of homomorphisms, and thus their image and inverse image which are subalgebras by Theorem 2.3.

Theorem 2.9 (Subalgebra Lattice) *Any intersection of Σ^A-subalgebra of T_i of $\mathcal{A}$ is a Σ^A-subalgebra, and A is a Σ^A-subalgebra of itself.*

The subalgebra of any abstract algebra is thus a complete lattice.

Let $Hom_\Sigma[\mathcal{A}, \mathcal{B}]$ denote the set of morphisms from $[A, \Sigma^A]$ to $[B, \Sigma^B]$. We have this easy result :

Theorem 2.10 (Monoid of Endomorphisms) *The set of morphism into an abstract algebra forms a monoid, namely, $< \{\rho \in Hom_\Sigma[\mathcal{A}, \mathcal{A}]\}, \circ, id_A >$, for all Σ and $[A, \Sigma^A]$.*

2.3.3 Induction

In this section, we adapt results on mathematical induction [Hen60]. These results are very strong in establishing uniqueness of definability by mathematical induction. Uniqueness implies well-definedness on the one hand, and also gives

Theorem 2.8 (Homomorphism and Structural Induction) *For all injective $\eta : X \to Y$, $Y \subseteq A$, any Σ-algebra $\mathcal{A}$ containing Y, the following are equivalent :*

(1) : *The unique homomorphism $h : \mathcal{T}_\Sigma(X) \to \mathcal{A}$ is surjective.*

(2) : *$\mathcal{A}$ has no proper subalgebra containing Y.*

(3) : *(Structural Induction): If P is a subset of A, such that*

a. $y \in P, \forall y \in Y$.

b. *for each $\sigma \in \Sigma$ of arity n, if $t_i \in P$ implies $\sigma^A(t_1, \cdots, t_n) \in P$.*

then $P = A$.

Proof 2.8 (1) $\Rightarrow$ (2) : *Suppose $\mathcal{A}$ has a proper Σ-subalgebra, $\mathcal{B}$, containing Y. The inclusion map $\iota : \mathcal{B} \to \mathcal{A}$ is surely a homomorphism, injective but not surjective. By the initiality of $\mathcal{T}_\Sigma(X)$, there is a unique homomorphism, $g : \mathcal{T}_\Sigma(X) \to \mathcal{B}$. By the compositionality of homomorphisms, we have $\iota \circ g : \mathcal{T}_\Sigma \to \mathcal{B}$, another homomorphism which is non-surjective. By the uniqueness of initiality again, $h = \iota \circ g$, contradicting h being surjective.*

(2) $\Rightarrow$ (3) : *Obviously P is a Σ-subalgebra $\mathcal{A}$ containing Y, if it satisfies the condition of (3a) and (3b). Since by (2), $\mathcal{A}$ is the least Σ-subalgebra containing Y. So it must be $P = A$, the conclusion of (3).*

(3) $\Rightarrow$ (1) : *Suppose h is not surjective. By the theorem of Image and Inverse Closure Theorem 2.3, the image $h(\mathcal{T}_\Sigma(X))$ is a Σ-subalgebra and satisfying (3a), containing X, and (3b), Σ-closed. However, as h is assumed nonsurjective, it is only a proper subset of A. Namely, $h(\mathcal{T}_\Sigma(X)) \neq A$, contradicting assumption (3).*

Theorem 2.6 (Representability) *Every Σ-algebra $\mathcal{A}$ generated from X is a homomorphic image of the term algebra $\mathcal{T}_\Sigma(X)$.*

Proof 2.6 *Consider the inclusion map, $f : X \to A$, defined as $f(x) = x, \forall x \in X$, namely, treating $x \in X$ as a variable and an element in A. By the Universal Extension Property, there is a unique homomorphic extension $f^* : \mathcal{T}_\Sigma(X) \to \mathcal{A}$, such that $f^*(x) = f(x), \forall x \in X$. By the image property, $f^*(\mathcal{T}_\Sigma(X))$ is a subalgebra of $\mathcal{A}$. Obviously $X \subset f^*(\mathcal{T}_\Sigma(X))$. By definition, $\mathcal{A}$ is the least subalgebra of itself containing X, we have $f^*(\mathcal{T}_\Sigma(X)) = \mathcal{A}$. So, f^* is surjective and $\mathcal{A}$ is its image.* $\square$

Taking the subalgebra and generation into account, $\mathcal{T}_\Sigma$ with an empty set of generators is an initial algebra in the class of all Σ-algebras. Namely, there is a unique homomorphism from $\mathcal{T}_\Sigma$ to $\mathcal{A}$, for all $\mathcal{A}$ in this class.

Theorem 2.7 (Uniqueness and Existence of Initial Algebra) *All initial Σ-algebras are isomorphic. For any Σ and equations E, there is an initial (Σ, E) algebra.*

Proof 2.7 *It is easy to see the uniqueness. Given two initial algebra, $\mathcal{A}$ and $\mathcal{B}$, and associated unique homomorphism $h : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B} \to \mathcal{A}$. By the composition of homomorphism and the uniqueness property, we have $g \circ h = id_\mathcal{A}$ and $h \circ g = id_\mathcal{B}$. So h and g are bijections.*

For the existence, it is standard to verify that the term algebra quotient by the set of equations, $\mathcal{T}_\Sigma(X)/E$, is an initial algebra among the class of (Σ, E)-algebras. $\square$

$x, y \in ker(h)$, and $[x] = [y]$. Since h is surjective, so is f. It is also easy to see that f is a homomorphism, as

$$\begin{aligned} f([\sigma^{A/ker(h)}([x_1], \cdots, [x_n])]) &= f([\sigma^A(x_1, \cdots, x_n)]) \\ &= h(\sigma^A(x_1, \cdots, x_n)) \\ &= \sigma^A(h(x_1), \cdots, h(x_n)) \\ &= \sigma^{A/ker(h)}(f([x_1]), \cdots, f([x_n]))]. \end{aligned}$$

So f is an isomorphism. □

Theorem 2.5 (Universal Extension Property) *Let Σ be a signature and X be a set of variables. Every mapping f from X to any Σ-algebra $\mathcal{A}$ admits a unique homomorphism f^* from the Σ-term algebra $\mathcal{T}_\Sigma(X)$ over X to $\mathcal{A}$.*

Proof 2.5 *The required mapping $f^* : T_\Sigma(X) \to A$ is defined as follows :*

(1) : $f^*(x) = f(x), \forall x \in X$.

(2) : $f^*(\sigma(t_1, \cdots, t_n)) = \sigma^A(f^*(t_1), \cdots, f^*(t_n))$ *for all n-ary operation symbols in Σ and all terms $t_1, \cdots, t_n$ in $T_\Sigma(X)$.*

First, f^* is well-defined as being total and single-valued. Second, f^* is an extension, as it extends f from (1), Third, f^* is an extension. This is the fact that the realization of σ on $\mathcal{T}_\Sigma(X)$ is by $\sigma^{T_\Sigma(X)}(t_1, \cdots, t_n) = \sigma(t_1, \cdots, t_n)$. So (2) becomes $f^*(\sigma^{T_\Sigma(X)}(t_1, \cdots, t_n)) = \sigma^A(f^*(t_1), \cdots, f^*(t_n))$. Fourth, if there is another homomorphism extension, g, with $g(x) = f(x), \forall x \in X$, from $\mathcal{T}_\Sigma(X)$ over X to $\mathcal{A}$, we use the Principle of Structural Induction on the property that $g(x) = f^*(x)$, $\forall x \in T_\Sigma(X)$. □

Namely, $f_\alpha^A(x_1, \cdots, x_n) \in \rho^{-1}(U)$. So $\rho^{-1}(U)$ is Σ^A-closed and thus a subalgebra of $\mathcal{A}$. $\square$

From the proof, we also notice, in particular, if T is generated by k elements $x_1, \cdots, x_k \in A$, then $\rho(T)$ is generated by $\rho(x_1), \cdots, \rho(x_k) \in B$. As a result, if ρ is injective on $\alpha = \{x_1, \cdots, x_n\} \subset A$, then the Σ^A-subalgebra T generated from α is isomorphic to the Σ^B-subalgebra $\rho(T)$ generated by $\{\rho(x_1), \cdots, \rho(x_k)\} \subset B$.

As a corollary, we have

Corollary 2.1 *For any $\rho \in Hom_\Sigma[\mathcal{A}, \mathcal{A}]$, $\rho(S)$ is a subalgebra of $\mathcal{A}$. In other words, $[\rho(S), \Sigma^B \downarrow \rho(S)]$ is a Σ^B-algebra.*

The following theorems constitute the basis of programming by (term structure) patterns, as a mechanism of definition. As can be seen the well-definedness is immediate as a result of Theorem 2.5, and completeness by Theorem 2.6. Moreover, structural induction can be used for other program correctness properties. The relationship can be summarized as in Theorem 2.8, which is an easy generalization of Proposition 16 of [MG88].

Theorem 2.4 (Homomorphism Theorem) *If h is a surjective homomorphism from a Σ-algebra $\mathcal{A}$ to a similar algebra $\mathcal{B}$, then $\mathcal{B}$ is isomorphic to $\mathcal{A}/ker(h)$.*

Proof 2.4 *Let h be a surjective homomorphism from $\mathcal{A}$ to $\mathcal{B}$, define a mapping $f : A/ker(h) \to B$ by*

$$f([x]) = h(x), \forall x \in A.$$

First, f is well-defined, for if $[x] = [y]$, then $f([x]) = h(x) = h(y) = f([y])$. Furthermore, f is injective, because if $f([x]) = f([y])$, then $h(x) = h(y)$ so

$\bigcup\{gen^n(X) \mid n \in \mathcal{N}\}$, where $x_i \in gen^{k_i}(X)$, taking k to be the maximal among k_i, we have $x_i \in gen^k(X)$, so, by definition, $\sigma^A(x_1, \cdots, x_n) \in gen^{k+1}(X) \subseteq \bigcup\{gen^n(X) \mid n \in \mathcal{N}\}$. So we have, by being the least subalgebra,

$$\bigcap\{T_i \mid X \subset T_i, \mathcal{T}_i\ \Sigma^A\text{-}subalgebra\ of\ \mathcal{A}\} \subseteq \bigcup\{gen^n(X) \mid n \in \mathcal{N}\}.$$

Conversely, by induction, we have

$$gen^n(X) \subseteq \bigcap\{T_i \mid X \subset T_i, \mathcal{T}_i\ \Sigma^A\text{-}subalgebra\ of\ \mathcal{A}\}, \forall n.$$

So

$$\bigcup\{gen^n(X) \mid n \in \mathcal{N}\} \subseteq \bigcap\{T_i \mid X \subset T_i, \mathcal{T}_i\ \Sigma^A\text{-}subalgebra\ of\ \mathcal{A}\}.$$

□

Now we move to properties of homomorphisms and their relation with congruence vis the notion of kernel.

Theorem 2.3 (Image and Inverse Closure) *For all $\rho \in Hom_\Sigma[\mathcal{A}, \mathcal{B}]$, (1) if T is a subalgebra of $\mathcal{A}$, then $\rho(T)$ is a subalgebra of $\mathcal{B}$, (2) if U is a subalgebra of $\mathcal{B}$, then $\rho^{-1}(U)$ is a subalgebra of $\mathcal{A}$.*

Proof 2.3 *(1) Given $f_\alpha \in \Sigma$ and $y_1, \cdots, y_n \in \rho(T)$, choose $x_1, \cdots, x_n \in T$ such that $\rho(x_i) = y_i$. Since T is a subalgebra, $f_\alpha^A(x_1, \cdots, x_n) \in T$. By definition of morphism, we have*

$$f_\alpha^A(y_1, \cdots, y_n) = f_\alpha^B(\rho(x_1), \cdots, \rho(x_n)) = \rho(f_\alpha^A(x_1, \cdots, x_n)) \in \rho(T).$$

So, $\rho(T)$ is a Σ^B-closed, namely, a subalgebra of $\mathcal{B}$.

(2) Given $f_\alpha \in \Sigma$ and $x_1, \cdots, x_n \in \rho^{-1}(U)$, by definition, we have $\rho(x_i) = y_i$. Again, by the definition of morphism,

$$\rho(f_\alpha^A(x_1, \cdots, x_n)) = f_\alpha^B(\rho(x_1), \cdots, \rho(x_n)) = f_\alpha^B(y_1, \cdots, y_n) \in U.$$

Theorem 2.1 (Structural Induction) *Let $\mathcal{A}$ be a Σ-algebra generated by X, namely, the smallest Σ-algebra containing X. To prove a property $\mathcal{P}$ holds for all elements in A, it suffices to show the validity of :*

(1) Basis : $\mathcal{P}(x), \forall x \in X$.

(2) Ind : *For all n-ary $\sigma^A, n \in \mathcal{N}$, if ([Ind. Hypo.]) $\mathcal{P}(a_i)$, for $a_1, \cdots, a_n \in A$, then $\mathcal{P}(\sigma^A(a_1, \cdots, a_n))$,*

Proof 2.1 *Let A_P denote the set $\{x \mid \mathcal{P}(x), x \in A\}$. Obviously, $A_P \subseteq A$. By the induction basis, we have $X \subseteq A_P$, and by the induction step, A_P is closed. By assumption, A is the smallest Σ-closed set containing X, so $A \subseteq A_P$. We have $A = A_P$.* □

We next identify the constructive iteration and non-constructive set-theoretic intersection for the generation of a smallest Σ-subalgebra.

Given a subset X of the carrier of a Σ-algebra $\mathcal{A}$, we denote by $\Sigma^A(X)$ the set of all elements generated in one step from X. Formally,

$$\Sigma^A(X) = \{\sigma^A(x_1, \cdots, x_n) \mid \sigma \in \Sigma_n, n \in \mathcal{N}, \text{ and } x_1, \cdots, x_n \in X\}.$$

Define a mapping $gen()$ from 2^A to A by $gen(X) = X \cup \Sigma^A(X)$ and its finite powers (iterations) as $gen^0(X) = X$ and $gen^{n+1}(X) = gen(gen^n(X))$ for $0 \leq n$.

Theorem 2.2 (Iterative Generation)

$$\bigcap\{T_i \mid X \subset T_i, \mathcal{T}_i \ \Sigma^A\text{-}subalgebra\ of\ \mathcal{A}\} = \bigcup\{gen^n(X) \mid n \in \mathcal{N}\}.$$

Proof 2.2 *First $\bigcup\{gen^n(X) \mid n \in \mathcal{N}\}$ is indeed a Σ^A-subalgebra containing X. It is obvious that $X \subseteq \bigcup\{gen^n(X) \mid n \in \mathcal{N}\}$. For closure, if $x_1, \cdots, x_n \in$*

The closure of a set α with respect to Σ is denoted as $\mathcal{C}l_\Sigma(\alpha)$

Definition 2.4 (Homomorphisms) *Let $\mathcal{A} = [A, \Sigma^A]$, $\mathcal{B} = [B, \Sigma^B]$ be a similar abstract algebra, namely, with the same signature Σ and its realization as Σ^A and Σ^B respectively. A function $\rho : A \to B$ is a (homo)morphism of $\mathcal{A}$ into $\mathcal{B}$ if for all $f_\alpha \in \Sigma$,*

$$f_\alpha^B(\rho(x_1), \cdots, \rho(x_n)) = \rho(f_\alpha^A(x_1, \cdots, x_n)).$$

Usually, the homomorphisms are simply referred to as morphisms.

Definition 2.5 (Σ-term) *Let Σ be a signature and X be a set of variables. A Σ-term over X is defined inductively as*

(1) : *for all $x \in X$, x is a Σ-term over X.*

(2) : *if $\sigma \in \Sigma$, $n \in \mathcal{N}$, and $t_1, \cdots, t_n$ are Σ-terms over X, then $\sigma(t_1, \cdots, t_n)$ is a Σ-term over X.*

The set of Σ-terms over X is denoted as $\mathcal{T}_\Sigma(X)$.

Definition 2.6 (Term Algebra) *Let Σ be a signature and X be a set of variables. The algebra $\mathcal{T}_\Sigma(X)$ with carrier $T_\Sigma(X)$, and realization $\sigma^{T_\Sigma(X)}$ of $\sigma \in \Sigma$ as*

$$\sigma^{T_\Sigma(X)}(t_1, \cdots, t_n) = \sigma(t_1, \cdots, t_n), \forall t_1, \cdots, t_n \in T_\Sigma(X),$$

is called the Σ-term algebra over X, or simply, the term algebra.

2.3.2 Theorems about Subalgebra and Homomorphisms

We first introduce the set-theoretical version of structural induction as a metatheorem for the study of carriers of any Σ algebra.

2.3 Foundations for Homomorphism and Induction

2.3.1 Basic Definitions

Let us first give some basic definitions, and fix some notation:

Definition 2.1 (Signature) *A signature is a set Σ, whose elements are called operation symbols, together with a mapping $\pi : \Sigma \to N$, called the arity function, assigning to each operation symbol its finite arity (number of arguments).*

Definition 2.2 (Σ-Algebra and Realization) *A realization of an n-ary operator symbol in a set A is an n-ary (internal) total operation on A. Given a signature, a Σ-algebra $\mathcal{A}$ is a pair $[A, \Sigma^A]$, where A is a nonempty set, called the carrier of $\mathcal{A}$, and a family $\Sigma^A = \{\sigma^A \mid \sigma \in \Sigma\}$ of realizations σ^A of all operation symbols σ in Σ.*

Note we use $\mathcal{A}$ for an algebra, but use A for its carrier, a set. A given set could be the carrier of many algebras of different signatures. For example, N in $< N, +, 0 >$ and $< N, +, \times, 0, 1 >$, and $\{0,1\}$ in $< \{0,1\}, \vee, 0 >$ and $< \{0,1\}, \wedge, 1, ()^{-1} >$. Further, we are dealing only with finite total operations, neither infinite nor partial operations. However, there could be an infinite number of operation symbols and their realizations, for example, $< N, +, 0, \{(\lambda x. \lfloor x \times m \rfloor) \mid m \text{ a positive real number}\} >$.

Definition 2.3 (Σ-Subalgebra) *By a Σ^A-subalgebra of an abstract Σ-algebra $\mathcal{A} = [A, \Sigma^A]$ is meant a subset T of A which is closed under the operations Σ^A, or Σ^A-closed. Namely, we require that if $f_\alpha \in \Sigma^A$ and $x_1, \cdots, x_{n(\alpha)} \in T$, then $f_\alpha(x_1, \cdots, x_{n(\alpha)}) \in T$.*

operators amounts to mathematical induction (or iteration). The latter is seen as the Recursion Theorem (Theorem 2.11). The former is easily seen in the definition of homomorphism, for example,

$$h(x +\!\!\!+_\alpha y) = h(x) +\!\!\!+_\beta h(y).$$

As suggested by the introduction of function space, it is natural to consider extensions with mappings that are less uniform than homomorphisms, such as those dictated by the law of (right) distributivity. The first candidates to consider are obviously projective mappings, such as

$$h(x +\!\!\!+_\alpha y) = h(x)$$

and/or

$$h(x +\!\!\!+_\alpha y) = h(y),$$

and their iterated applications. Indeed, the simple case of homomorphisms on a similar algebra of only one unary operator and those of only one binary operator embody the programs defined by homomorphism and induction. This is particularly vivid by noticing that the initial term algebra on single binary operator, $\mathcal{T}_{\{\bullet,0\}}(\emptyset)$, and the initial term algebra on a single binary operator, $\mathcal{T}_{\{succ(),0\}}(\emptyset)$, meet at the common natural number which is a Peano model, initial (in other words, isomorphic to $\mathcal{T}_{\{succ(),0\}}(\emptyset)$) among similar algebras of a single unary operator.

These perspectives constitute the basics, following the simple rigid body metaphor, of the dual-$\mathcal{M}$-set framework.

monoids, it bears most connection with mathematical induction and consequently it is easier to introduce useful partial orderings compatible with its algebraic structure.

Second, there is the issue about single copy vs. multiple copies of a common carrier. To be precise, both $< N, +, 0 >$ and $< N, \times, 1 >$ are homomorphic images of the initial algebra $\mathcal{T}_\Sigma(\alpha)/\theta_E$, where Σ is $\{\bullet, \Lambda\}$ with E the laws of associativity and identity. However, despite the common carrier of the natural numbers, they cannot be formally identified. Otherwise, their neutral elements would be identified ($0 = 1$), in conflict with the intended algebra. Theoretically this could be resolved by using a direct product (i.e. multiple copies), or by using the enriched signature with two binary operations as a semiring. The former is not intuitive and the latter introduces concerns about additional equational laws such as distributivity (which can be important). Unfortunately, these are not uncommon situations.

Instead of dual copies of similar algebras with identical carriers, one of them can be incorporated as a function space. In particular, one can be the homomorphisms represented by uniform definability. Both objects and their function space (dual) then should be taken into account and in a uniform flavor. This is happily the case for natural numbers, as multiplication can be treated as a homomorphism over addition. In fact, this philosophy is also taken in the vector space approach in mathematics and physics. Furthermore, the adoption of (dual) function space opens up fruitful avenues for the expansion of interesting mappings and concepts indeed.

Third, as an observation, pragmatically the notion of homomorphism on binary operators amounts essentially to distributivity, and homomorphism on unary

of this unification problem may not be the main issue; after all, undecidability and NP-completeness are well recognized. Rather, the issue may be the indirect effects on programming practice. Deciding term equivalence introduces 'confusion', i.e., two terms having a same denotation, into programming, and is apt to lead to careless mistakes like those in the examples.

Despite the above-mentioned issues, there are a few more points to make concerning the rigid body metaphor. The combination of the perspective in the following points and the advantages of the algebraic approach (iterative closure, homomorphism and induction), have contributed significantly to the shaping of the idea of dual-$\mathcal{M}$-sets.

First, there are some neighboring abstract algebras just next to, yet excluded by, the class of Σ-algebras. They could be well-behaved enough to live consistently and helpfully with the strict Σ-algebra. A rigid body with a small hole should also be amenable to the same analytical methods of perfect rigid bodies. The cases we have particularly in mind include substituting semigroups or semirings for monoids, and inductive models for Peano models.

While they could be incorporated as homomorphic images, they have implications in terms of the representation of the initial algebra. Take, for instance, the case of monoids with associated operator $\bullet$. We can form the term algebra as a semigroup ($\mathcal{T}_{\{\bullet\}}(\alpha)$), taking the quotient only modulo associativity (giving lists), and then applying a completion embedding into a monoid, $\mathcal{T}_{\{\bullet,e\}}(\alpha)$ by introducing an extra constant, e, as the unit. This is depicted as follows :

$$\alpha \ \hookrightarrow \ \mathcal{T}_{\{\bullet\}}(\alpha) \ \hookrightarrow \ \mathcal{T}_{\{\bullet,e\}} = \mathcal{M}(\alpha) = \mathcal{T}_{\{\bullet\}}(\alpha) \cup \{e\}$$

This process, called completion, is of interest to us, in particular. For the case of

The methodology of this approach confirms, to a certain degree, the rigid body metaphor. First a collection of raw data, α, is given, and the programmer determines its concrete semantic universe, $\mathcal{A}$ and $\mathcal{B}$ in this case. Then a class of (equationally defined) abstract algebras, the class of similar algebra having same equational constraints as $\mathcal{A}$, is selected for representation of programs, giving a combined signature and a term algebra, $\mathcal{T}_\Sigma(\alpha)$. An advantage provided by the methodology is that programming by patterns or homomorphisms is then possible, giving a 'linear' way of reasoning about programs.

2.2 Perspectives

The analysis we presented in the previous section identifies two major construction processes for programs. The processes respectively work by iterative closure on the smallest subalgebra of the intended universe, and by homomorphism or programming with structural induction. The soundness of structural induction within the smallest subalgebra or for surjective homomorphisms should also be stressed (Theorem 2.8 (Homomorphism and Structural Induction)). In our rigid body metaphor, the intended universe is inhabited by points and vectors; and the homomorphisms are linear operators, programmable and subject to mathematical reasoning and manipulation. The intuition about the processes of construction also make concrete our methods for software development following the rigid body metaphor. Obviously, a pragmatic test is needed to evaluate the significance of a rigid body metaphor for software.

The approach described here should avoid some programming errors. Seemingly trivial errors may happen quite often in connection with the issue of deciding equivalence between terms in $\mathcal{T}_\Sigma(\alpha)$ with a set E of equations. The intractability

gebra $\mathcal{T}_\Sigma(\alpha)$, either with or without equations. Its goal is data abstraction and representation independence. Syntactically, the practice of programming by patterns is adequate for definition by homomorphism and reasoning by structural induction. Unfortunately, programming by patterns is not equivalent to definition by homomorphism, and this difference can lead to unintended semantics, unsound reasoning and bad programs. The diagram below, in which all arrows are homomorphisms, summarizes the connection of the three aspects of program development based upon such an algebraic framework, disregarding the issue of equational congruence.

$$\begin{tikzcd}
 & \mathcal{T}_\Sigma(\alpha) \arrow[d] \arrow[dr] & \\
\alpha \arrow[ur] \arrow[r] \arrow[dr] & \mathcal{C}l(\alpha) \arrow[r] \arrow[d] & \mathcal{B} \\
 & A \arrow[ur] &
\end{tikzcd}$$

Figure 2.3: Three Algebraic Paths

The lowest path in this diagram illustrates the intended semantics of programs mapping α to β, with intended domain of application $\mathcal{A}$. Depending upon information about α with respect to $\mathcal{A}$ and about the range of β, the middle path could be sufficient semantically. The upper path is the indirect mediation for computational and representational feasibility through $\mathcal{T}_\Sigma(\alpha)$.

argue that its primary virtue lies in its making possible programming by inductive definition [PS89]. Unfortunately, the semantic nature of Herbrand terms makes them hard to manipulate directly as computational objects. Furthermore, in general, problems are pushed into type coercion which introduces differences between $\mathcal{C}l_\Sigma(\alpha)/\theta_E$ and $\mathcal{C}l_\Sigma([\alpha])/\theta_E$. For instance, with respect to the word algebra $< \{a, b\}^*, \cdot, \epsilon >$, let α be $\{a, a \cdot b\}$, and E be the laws of monoids. Then the monoidal closure with α, $\mathcal{C}l_\Sigma(Conv_A(\alpha))/E$ is $(a + b)^*$, while $\mathcal{C}l_\Sigma([\alpha])/E$ gives us $(a + a \cdot b)^*$. The point is that $a \cdot b$ is a composite aggregate as an element of the monoid $\{a, b\}^*$, and is an atom in $\{ab, b\}^*$. Semantic information that should be provided as type conversion can be mistaken simply as syntactic tokens or placeholders.

Now the two earlier 'bad' definitions, silly and card, can be explained as being defined on $\mathcal{T}_\Sigma(\emptyset)$, not $\mathcal{T}_\Sigma(\emptyset)/\theta_E$. They fail to be consistent with the congruence θ_E. That is, elements in a given congruence class are mapped to different values, contradicting well-definedness. This 'error' can be taken either as a failure to confirm the requirement of homomorphism or missing the semantic requirement as of (monoid) equational constraints between $\mathcal{T}_\Sigma(\emptyset)$ terms. However, we would emphasize there is also the aspect of structural induction that is mistaken too. By that, we mean that the definition is not a definition by (mathematical) induction. Precisely, while the range is intended to be a natural number system, the domain is not an initial algebra for the common Peano natural number system.

2.1.2 Summary

The algebraic approach, using initial algebras to specify the semantics of abstract data types for specification and program verification, centers around the term al-

the equational properties of sets respected by the set cardinality function.

In some cases, programs may seem to be purely semantically based, merely defining homomorphisms between similar algebras. In the more general sense of programming by patterns, we can still see how $\mathcal{T}_\Sigma$ mediates among algebras in Figure 2.1.1. Here mappings out from $\mathcal{T}_\Sigma(\alpha)$ are unique homomorphisms, and

$$\begin{tikzcd}
\mathcal{A} \arrow[rr] \arrow[dd] & & \mathcal{C} \arrow[dd] \\
& \mathcal{T}_\Sigma(\alpha) \arrow[ul] \arrow[ur] \arrow[dl] \arrow[dr] & \\
\mathcal{B} \arrow[rr] & & \mathcal{D}
\end{tikzcd}$$

Figure 2.2: Homomorphisms Mediated by a Term Algebra

every term homomorphism and its compositions with operators in a concrete semantic algebra on the outside can be uniquely factored. So in this sense, programming by patterns or structural homomorphisms is genuinely based on homomorphisms starting with $\mathcal{T}_\Sigma(\alpha)$.

In comparison, the semantic subalgebra closure possesses no such obstacle, essentially programming and reasoning based on the inductively generated universe. This style of programming is indeed frequently mentioned as one of the important aspects of Logic Programming; not only are its semantics based on induction over the Herbrand Universe [Llo89], but in fact, Paulson and Smith

$$
\begin{tikzcd}
{[\alpha]} \arrow[d] & \alpha \arrow[l] \arrow[r] \arrow[dl] \arrow[dr] & Conv_A(\alpha) \arrow[d] & \\
\mathcal{T}_\Sigma(\alpha) \arrow[rr] \arrow[dr] & & \mathcal{C}l_\Sigma(Conv_A(\alpha)) \arrow[r, hook] & \mathcal{A}_\Sigma \\
& \mathcal{T}_\Sigma(\alpha)/\theta_E \arrow[ur] & &
\end{tikzcd}
$$

Figure 2.1: Summary of Constructions

There are several nice consequences of this approach that will be elaborated below. First, programming by patterns (homomorphisms) is justified by several results: Theorem 2.7 (Uniqueness and Existence of Initial Algebra), Theorem 2.6 (Representability), and Theorem 2.5 (the Universal Extension Property). Second, it is possible to reason about properties of homomorphisms thus defined, as justified by Theorem 2.8 (Homomorphism and Structural Induction).

While the syntactic process above seems more natural, there is this additional homomorphism from $\mathcal{T}_\Sigma(\alpha)$ to $\mathcal{T}_\Sigma(\alpha)/\theta_E$ which is non-trivial most of the time. This quotient homomorphism is either computationally expensive or without an effective decision procedure (as is common in rewriting and equational unification [Hue80, Wec92]). Also, in practice the difference between the two term algebras is a source of subtleties and error, and often they are carelessly mistaken to be the same. In particular, a homomorphism like length differs significantly from an arbitrary mapping like silly, and a term algebra homomorphism like card omits

by a homomorphism θ_E, where E is a set of equational laws, so that θ_E is the corresponding congruence induced by E.

This gives us our second process for constructing algebras, combining syntactic construction and homomorphic images. Since congruence and kernel of a homomorphism are dual notions, and any composition of homomorphisms is still a homomorphism, this process can be made more efficient when the ultimate homomorphism is known. For instance, efficiency comes from bypassing the computation of mediating congruences. The process of construction can be described by following diagram :

$$\begin{tikzcd} \alpha \arrow[r, leftrightarrow] & {[\alpha]} \arrow[r, hook] & \mathcal{T}_\Sigma(\alpha) \arrow[r] \arrow[dr] & \mathcal{T}_\Sigma(\alpha)/\theta_E \arrow[d] \\ & & & \mathcal{A}_\Sigma \end{tikzcd}$$

The two processes of construction can be put together for summary and comparison in the following diagram, where $[\alpha]$ represents the set α treated as units (constants) in Σ:

Notice, while the intended ultimate algebra is $\mathcal{A}_\Sigma$ in both cases, we actually get only $\mathcal{C}l_\Sigma(Conv_A(\alpha))$ and $\mathcal{T}_\Sigma(\alpha)/\theta_E$, respectively. They are then mediated to $\mathcal{A}_\Sigma$, by an inclusion injection map (still a homomorphism). Indeed, the property of being able to mediate all homomorphisms through $\mathcal{T}_\Sigma(\alpha)$ and $\mathcal{T}_\Sigma(\alpha)/\theta_E$ for the class of Σ-algebra and Σ-algebra modulo equations E respectively (see Theorem 2.6 (Representability) and Theorem 2.5 (Universal Extension Property)) is a key point in programming by patterns in particular, and in software methodologies based upon Initial Algebra semantics of abstract data types in general.

The first process of construction is the generation of a subalgebra, which by Theorem 2.2 (Iterative Generation), can be iteratively constructed from the given set (of raw data) α, and is actually equivalent to the smallest subalgebra containing α, denoted as $\mathcal{C}l_{\Sigma}(\alpha)$. This can be seen as an instance of definition by induction, as in [Acz77] and [PS89], where each operation corresponds to a rule of generating new elements. Notice, however, the subset α is in fact more than an ordinary set — it is a subset of the carrier A of an algebra $\mathcal{A}$. Specifically, elements of α already satisfy the same constraints of equations of $\mathcal{A}$, and the algebraic constraints of internal operations of the ultimate algebra may be satisfied by subsets of α. An extreme example is when α is in fact A. In this sense, α is not an arbitrary set, but is a typed structure. Also, the process of generating an algebra from an ordinary set must actually consist of two steps, type coercion and algebraic closure. This is a bottom-up iterative accumulation process, starting from α, generating new elements and adding to α, and repeating this until no new elements are generated. In each step algebraic constraints are checked implicitly for 'newly' generated elements, as a result of set union which eliminates duplicates. Thus the process can be summarized as follows, where '$\hookrightarrow$' is another way of writing '$\subseteq$':

$$\alpha \mapsto Conv_A(\alpha) \hookrightarrow \mathcal{C}l_{\Sigma}(Conv_A(\alpha)) \hookrightarrow \mathcal{A}_{\Sigma}$$

Alternatively, we can perform this process top-down. Instead of checking constraints at each iteration, we treat the given set α as raw data (atoms or units), variables or placeholders. We thus treat them as new constants enriching the original signature Σ. Neglecting any internal information, we then simply close this enriched signature to give a (word) term algebra ($\mathcal{T}_{\Sigma}(\alpha)$), and then get the required algebra by its equational quotient. This quotient can be defined

'linear' operators on rigid bodies, which are programmed by representational patterns. This perspective gives some motivation for the development of dual-$\mathcal{M}$-sets, along with an appreciation for the applicability of the rigid body metaphor.

The first sections of this chapter are written now in a way that assumes the reader is familiar with concepts from universal algebra and inductive reasoning. Formal foundations on universal algebra and inductive reasoning are covered in the last section of this chapter, mainly in order to avoid interrupting the flow of ideas. Readers may wish to jump to the formal treatment at the end of this chapter first, if the treatment earlier in the chapter presupposes concepts that seem unfamiliar. Although an informal understanding should be enough, for the sake of precision we feel it is important to go into the proofs of these foundations. Most of the foundations are already pretty well known [MG88, Wec92, Bir67], but the organization of results and their constructive proof helps develop not only understanding of the theorems, but also new perspectives on their application.

2.1.1 Processes of Construction

Given any collection of raw objects, α, there are basic mechanisms that 'construct' an algebra satisfying a set of properties on α. In universal algebra [Bir67, Wec92], in particular, for any equationally defined class of similar algebra (called a variety), upon when the above mentioned application is based, three semantic constructions are used : subalgebra, homomorphic image and (sub)direct product. One syntactic construction, the *term algebra*, is also used to establish a representative among a class of similar algebras. In fact, it is this syntactic construction that forms the basis of data representation upon which programming and computation can be based.

matical mechanisms for reasoning about programs.

However, issues concerning these programs can be subtle, and careless programming leads to errors. For instance, consider the definition from [Wad87]:

```
silly Nil             =  1
silly (Unit x)        =  2
silly (xs Join ys)    =  silly xs + silly ys
```

and the expression from [BS91]:

```
let fun  card(Empty)           =  0
   |     card(SngSet(_))       =  1
   |     card(Union(s1,s2))    =  card(s1) + card(s2)
   in
         card(Union(SngSet(A), Union(Empty, SngSet(B))))
```

Both definitions lead to 'invalid' functions. The usefulness of structural matching and definition by induction for program correctness has been emphasized in the past [Wad87], particularly where correctness is proven by equational reasoning and induction.

In this chapter we try to get some perspective on the subtleties reflected by the two examples above. Below we first describe various processes for constructing a representation, an instance of a rigid body. We then analyze homomorphisms,

CHAPTER 2

Algebraic Data Representations And Programming

2.1 Programming By Homomorphisms

In many programming languages it is common to find programs like the following, involving a simple decomposition over a structure:

$$\begin{aligned} \mathsf{length\ Nil} &= \mathsf{0} \\ \mathsf{length\ (x\ Cons\ xs)} &= \mathsf{1} + \mathsf{length\ xs} \end{aligned}$$

Such divide-and-conquer programs implement *homomorphisms* on the structure in question (in this case, lists). The data structure construction operator (Cons) is mapped homomorphically to an operator (+) in another algebra. More generally, the monoid of lists is mapped homomorphically by length to the monoid of natural numbers.

Structure-based homomorphisms are very useful, and in fact have become an established part of programming practice in patterns and structure representations. They embrace the long-term point of view that *programs are structures susceptible to mathematical analysis.* In particular, they integrate algebra and structural induction into programming practice — two of the strongest mathe-

plication with types in programming language [Mil78, Tiu90, Mit90] and logics [Gir87].

Our primary contribution lies in identifying the notion of the dual-$\mathcal{M}$-set, an algebraic structure that we use throughout in expressing basic results. The usefulness and power has been tested in the above-mentioned non-trivial systems. Beyond the technical results and extension in the modeling, the most significant contribution comes perhaps from the perspective the new framework introduces. We believe the generality and applicability of the dual-$\mathcal{M}$-sets merits further exploration.

As readers may speculate, there is quite a parallel between the dual-$\mathcal{M}$-sets and categorical concepts, such as functors, natural transformations and adjunctions. Indeed, our larger objective (witnessed by the role here of duality) is the exploitation of adjunction in computer science. However, it seems best under the constraints of a dissertation simply to restrict attention to dual-$\mathcal{M}$-sets for the moment, as these give a compelling and useful abstraction. A work starting from category theory will doubtless be of greater generality; nonetheless, it may also lose the insight provided from concrete examples. In retrospect, we believe the strategy of limiting the discussion here on dual-$\mathcal{M}$-sets has paid off.

alize in categorical framework to optimal solutions using *catamorphisms* [BM92], homomorphisms on initial objects in a category.

Most of basic material on induction and universal algebra is pretty well-known [Jac85, Bir67, Wec92, MG88, Acz77]. Spivey in [Spi89] also did a categorical analysis of the theory of lists. However, he only dealt with the ideal part, namely, those around the Homomorphism Theorem. This part is in fact very clear, and powerful, giving an instance of adjunction [AL91, etc85, Pie91]. Our analysis focuses on those non-homomorphic (or less homomorphic) aspects of the theory of lists. Specifically, we focus on fictitious values, direct reduction, and inductive mappings, which give non-trivial fragments of the theory. In fact, our analysis also extends the theory itself, by considering the order theoretic aspect. The notions of Monad, categorically equivalent to the notion of adjunction, is recently advocated as a method of organizing programs by Wadler [Wad90, Wad92]. The idea is essentially to emphasize the power of encapsulating computations inside a type constructor (specifically the *list* constructor) in the theory.

The Lambek Calculus was first proposed in [Lam58] as different framework from Transformation Grammars. The interest faded away when it is proved to be equivalent in competence to Context Free Grammar [Bus88]. The revival is first due to its use in formal semantics, particularly in English, pioneered by Montague [Tho74]. The focus in context freeness, or phrase structure, in natural language processing [PG82, GP87] such as in Generalized Phrase Structure Grammars and Lexical Functional Grammars [Sel85], Categorial Grammars [RW88] and Combinatorial Grammars [Ste88], help revive the interests in the Lambek Calculus and various variants [Ben88a, HP92] and extensions [Moo88, Kan92]. As the Lambek Calculus a type reduction system and a logic about types, it also bears some im-

1.4 Related Work

The idea of a 'rigid body' and the related background knowledge is somehow accumulated in years of academic study. As the idea of rigid body has become almost common sense, it is hard to trace the exact source of influence among the works presented here.

It is very well known that structure and ordering are important concepts useful not only in theoretical computer science, but also in programming practice and in mathematics. Structural induction is well known such as an algebraic tool in program specification and correctness proofs, development of abstract data types, and domain theory for program semantics.

For programming practice and applications, structural induction and recursion are also of central importance. Database queries using structural recursion [BBN91] and structural homomorphism [PSV92, BS91], and programming by recursive pattern-matching [Wad87], have received attention recently. In large databases of complex-structured data, for example, efficient processing results from partitioning of computations in accordance with data constructors.

Structural homomorphisms on regular expressions were also used to give a unifying framework for path problems [Tar81], a class of problems of wide applicability. Recently, Bern etc. [BLW85] also apply the notion of homomorphisms for deriving linear algorithms for some optimality of subgraph, using dynamic programming techniques for decomposing recursive constructed graphs. The Bird-Meertens formalism is also used for systematic derivation of programs, a practice coined as Algorithmics by Meertens [Mee86]. Bird use the notation to explore greedy algorithms [Bir92] using the generating-filter-select paradigm, and gener-

by homomorphism and mathematical induction.

We then extract the notion of the dual-$\mathcal{M}$-set by generalizing from this analysis. The idea is to consider simultaneously the objects and its function space, which are both monoids. Structure-preserving homomorphisms and order-preserving morphisms are then introduced, incrementally, into the function space, the dual space of monoid objects. Definitions, examples and basic properties are presented.

After introducing the formal framework, we apply to two comprehensive systems, to test the expressiveness of dual-$\mathcal{M}$-sets and the rigor of the concepts they embody. In Chapter Four, we apply it to the theory of lists, also known as Bird-Meertens formalism for functional calculus on lists. In Chapter Five, we apply it to the Lambek Calculus. A detailed introduction to specific theory and results are provided in the chapters. In brief, the results are pretty encouraging, not only in term of modeling the two systems, but in the perspectives and guidance provided by the dual-$\mathcal{M}$-sets framework. We work out some of the suggested extensions. In particular, we define the induction morphism for the Theory of Lists, and the analysis of universality of empty traces and pragmatic variants of the Lambek Calculus. While other results were known in the original theories, new proofs are also provided in this work along the dual-$\mathcal{M}$-set framework and the philosophy around the metaphor of rigid body.

We then conclude with a summary and discussion of future research.

with a compositional operator and identity. This monoid structure is inherited when richer representational coordinate systems are superimposed.

To test the power of the dual-$\mathcal{M}$-set framework, we apply it to the analysis of two different systems in current work. The first is the Theory of Lists, also known as the Bird-Meertens Formalism for computable functions over lists [Bir87, Mee86]. The second is the Lambek Calculus [Lam58, Lam61], a deduction system for syntactic types, a foundation of formal semantics [Ben88b, Tho74] and Categorial Grammars [RW88, Moo88]. Not only does the dual $\mathcal{M}$-set successfully model these two seemingly disparate theories, it also provides new perspectives and directions for work.

We believe the dual-$\mathcal{M}$-set is a fruitful aspect of the rigid body framework that is suitable for adaptation for software design and development. While not yet a extensive theory, it is not merely another formal game. Beyond the other technical results here, this work presents our argument for this belief.

1.3 Organization

In Chapter Two, we first examine ways of constructing free monoids for use in programming by patterns, and equational and inductive reasoning. In this we use notions and results in universal algebra [Bir67, Wec92] and in mathematical induction [Hen60]. In particular, we use the subalgebra construction and homomorphic image from universal algebra, and Henkin's results in mathematical induction about the uniqueness of Peano models of number systems and their homomorphisms. Not to be lost in formal investigation, we actually start with some real cases and conduct the analysis in an informal but informative style. We show how programming errors arise due to the constraints required for definition

Roughly speaking, a dual-$\mathcal{M}$-set is a monoid together with its 'dual space' (the set of structure-preserving mappings from a monoid to itself, also called its endomorphisms). The basic motivation is to consider data as a monoid structured object. Programs are then operators or transformations or (endo-)morphisms of monoids which, with ordinary program composition, constitutes a monoid too. These dual monoids enjoy the basic properties of linear spaces (vector spaces), and therefore should permit us to adapt the rigid body metaphor to software structures somehow.

For concreteness and a source of basic insights, we start with an analysis of a simple yet useful paradigm of software practice, known variously as *pattern-oriented programming*, *structural recursion*, or *inductive programming*. The rigid body metaphor is surprisingly helpful in understanding the basic attributes of this paradigm. Furthermore, it helps in understanding how the structural aspect and order-theoretical aspect of the paradigm interact, and how we can reconcile homomorphisms and mathematical induction. We therefore analyze this paradigm (both in its applicability and methods of construction) in a general context as algebraic structures and partial ordering on which induction can be meaningfully defined.

With this foundation, we then formalize the notion of a dual-$\mathcal{M}$-set with examples to illustrate its generality and potential. We have found dual-$\mathcal{M}$-sets particularly useful as a theory for *list processing*, with dual monoids for list construction and list operators. Lists also have a naturally induced ordering for mathematical induction. In studying homomorphism on lists it is also helpful to start from the view point of *duality*, and notice that dual lists (mappings from lists to lists) have the same structure as lists: they form a monoid – an algebra

geometry, and analysis.

Indeed, a compelling long-term point of view in software and programming language features is that *programs are structures susceptible to mathematical analysis*. Programming languages and concepts should be designed to support this point of view. There are many mathematically-based methodologies already, including transformational programming, program logics, algebraic methods, logic programming, etc. However, none of these provides a satisfactory basis for addressing the questions above.

1.2 The Dual-$\mathcal{M}$-Set

In this dissertation, we propose a framework, called dual-$\mathcal{M}$-sets, as a testbed adapting our basic intuition about rigid bodies and linear transformations to the context of software. We have come to the belief that, just as the rigid body in physics has the virtue of being representable with points and vectors, and provides the nice mathematical properties of linear spaces, the notion of rigid body in software corresponds to representability with monoids (a semigroup with identity) together with homomorphisms and mathematical induction.

The dual-$\mathcal{M}$-set is a generalization of the very common $\mathcal{M}$-set, a notion originating from the idea of a monoid acting on a set. In fact, the '$\mathcal{M}$' in these terms comes from the word 'monoid'. Monoids are useful in defining aggregate objects, such as sets or lists, which are constructed as compositions of a number of smaller objects. Monoids are also the simplest mathematical structure underlying the notion of an 'operator', since these operators can be composed associatively into other operators.

upon the body as *operators*. When the operators are restricted to be linear, we obtain what is called a *linear space*, and the situation is particularly nice. Also, aggregate objects (many-body systems) can be decomposed structurally into individual objects represented as multiple points in a common coordinate system. This modeling is quite 'modern' from the Computer Science perspective, relying for example on structures to represent aggregate objects and coordinates, and relying on semantic values (integers, reals or vectors) or procedures (coordinate transformations, rotations, translations) for representing measurements or other attributes of bodies.

Can we have a corresponding notion in software modeling and development? Is there a rigid body counterpart for programs and data? How about operators on these rigid bodies? What are the representational mechanisms for software rigid bodies?

A straightforward approach for answering these questions is to (try to) adapt the algebraic foundations of physics for software problems. For example, specific operations on algebraic structures are used to model transformations, and also to model the ordering in measurement and calculation. Furthermore, these rest on a common coordinate system representation, in which particles can be points and operators be transformations on points, and higher order notions such as vectors, matrices, and duality can be defined and manipulated. Intuitively, we would normally expect data to be a rigid body and a program to be a transformation, both represented as points or entities in a common space. In such a scenario, programming practice could be analogous to manipulating points, vectors and transformations, rather than lines of code. Furthermore, reasoning about programs would benefit from many mathematical tools, including algebra,

CHAPTER 1

Introduction

1.1 Motivation: The Rigid Body Metaphor

In physics and other applied fields, the concept of a *rigid body* has had enormous impact. When we adopt the rigid body metaphor, we make the general assumption that objects can be modeled as solid, immutable entities – rigid bodies. Transformations on and measurements of these objects are non-destructive, i.e., the result of an operation on any rigid body is still the same rigid body. Of course, the rigid body is not all there is; there are important other phenomena like, for example, colors, temperature, energy and almost-rigid bodies. However it is indeed a very useful concept with a powerful modeling basis.

This rigid body is often the critical concept in scientific *inquiry*, and is used in the *modeling, manufacturing and manipulation* of real-world objects. When it comes to reconciling models with reality, decisions about models have to be made consistent with one's objectives. Very satisfactory mathematical foundations have been obtained for rigid bodies, yielding models that are simultaneously general, precise, and and realistic in many scientific settings.

One foundation of particular importance is the *coordinate system*. It has become common to represent a rigid body as a point or a vector in some coordinate system, and represent *forces, measurement, or transformations* acting

The Theory of Lists (also known as the Bird-Meertens formalism), for specification and systematic derivation of programs; and the Lambek Calculus, a deductive systems and formal foundation for linguistic applications in Categorial Grammars and formal semantics. Beyond the modeling and new proofs of old results, we also develop extensions inspired by the perspectives that accompany the dual-$\mathcal{M}$-set modeling.

As the rigid body metaphor is very basic in the modeling of information, monoid structures are prevalent in current mathematical program structures. The framework of dual-$\mathcal{M}$-sets aspires for a mathematical program development methodology exploiting this basic metaphor, in keeping with the common long term aim of developing programs as mathematical objects susceptible to mathematical analysis and manipulation. With the encouraging results from its applications to the two examples, we believe that the framework of dual-$\mathcal{M}$-sets is a powerful and promising tool with wide applicability.

Abstract of the Dissertation

Dual $\mathcal{M}$-Sets : The Theoretical Framework and Applications

by

Shen-Tzay Huang

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1993

Professor D. Stott Parker, Chair

A new framework, the dual-$\mathcal{M}$-set, is proposed both as a means for structuring certain kinds of programs, and as a means for reasoning about aspects of software practice such as programming by patterns.

The framework is based upon the 'rigid body' metaphor in physics, an outlook which models information as the result of operator applications (measurements) on immutable objects (rigid bodies). This framework takes monoids as points and vectors, and structure preserving homomorphisms as linear operators. Order preserving endomorphisms (in particular, induction morphisms) are also included as basics. The conceptual framework treats both the objects and their functions as first class citizens. Instead of taking a conventional approach using category theory, we emphasize the local, though universally uniform, analysis of the construction and interaction of subalgebraic closure and homomorphisms, and order preserving morphisms.

We apply the framework to model, comprehensively, two different systems:

Shen-Tzay Huang and D. Stott Parker, A Study of Variants of Lambek Calculus and Its Implementation in Prolog, UCLA Computer Science Department, Technical Report, CSD-920029, June, 1992.

Shen-Tzay Huang and D. Stott Parker, Types For Lazy Logic Programs : Computation and Infinite Objects, UCLA Computer Science Department, Technical Report, CSD-920030, June, 1992.

Vita

Dec. 4, 1959	Born, Chia-I County, Taiwan
July, 1982	B.S. Information Engineering and Computer Science, National Taiwan University, Taipei, Taiwan
July, 1986	M.S. Information Engineering and Computer Science, National Taiwan University, Taipei, Taiwan
7/1986 - 8/1987	Associate Engineer, R&D Division of Information Industry Institute, Taiwan, R.O.C.
10/1988 - 6/1993	Teaching Assistant and Post Graduate Research Assistant, UCLA Computer Science Department Los Angeles, California

Publications

Chin-Chi Hsu and Shen-Tzay Huang, Concurrent Prolog : An Interpretation under Distributed Model, International Computer Symposium 1986, Tainan, Taiwan, R.O.C.

Acknowledgments

I am especially grateful to my advisor, D. Stott Parker, for his valuable technical guidance and unfailing enthusiasm in pressing for the frontier. I would also like to thank the members of my committee, Professor Martin, Muntz and Baker for their participation and comments.

I also wish to acknowledge National Science Foundation, which provided partial support during my graduate studies at UCLA (contract IRI-8917907).

I would like to thank my parents for their constant love and understanding to their son in this long pursue of higher education. The special thank goes to my dear sisters, Li-Chiung and Yu-Chuang, and my brother, Teng-Ian, without their support, the overseas study would be impossible to start, to proceed, not to mention, to reach the end, for the eldest son of this family. They also give me so much warmth and encouragement in the process. I would also like to thank Irene Chen, with her care and warm company, I can move fast and steady in the last few steps toward the final accomplishment.

Last but not least, I thank all friends in computer science department and in UCLA Taiwanese Student Association. With them, I share and enjoy experiences in United States as a foreign student, a memory for the whole life.

List of Tables

List of Figures

Table of Contents

To My Parents

Swei-Wen Huang and Gin-Tsui Liu

The dissertation of Shen-Tzay Huang is approved.

Kirby Baker

Richard Muntz

David Martin

D. Stott Parker, Committee Chair

University of California, Los Angeles

1993

UNIVERSITY OF CALIFORNIA

Los Angeles

Dual $\mathcal{M}$-Sets : The Theoretical Framework and Applications

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Shen-Tzay Huang

1993