

**A TRANSFORMATIONAL APPROACH
TO DATABASE SYSTEM IMPLEMENTATION**

Leonidas Fegaras
Computer Science Department
University of Massachusetts
Amherst, MA 01003

CMPSCI Technical Report 92-68

September 29, 1992

**A TRANSFORMATIONAL APPROACH
TO DATABASE SYSTEM IMPLEMENTATION**

A Dissertation Presented

by

LEONIDAS FEGARAS

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1993

Department of Computer Science

© Copyright by Leonidas Fegaras 1993

All Rights Reserved

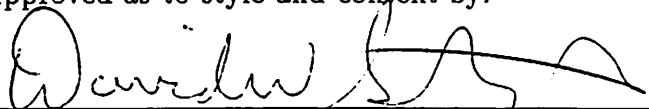
A TRANSFORMATIONAL APPROACH
TO DATABASE SYSTEM IMPLEMENTATION

A Dissertation Presented

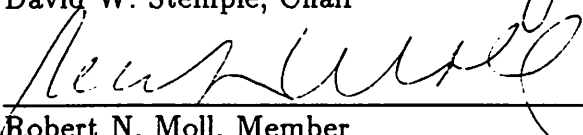
by

LEONIDAS FEGARAS

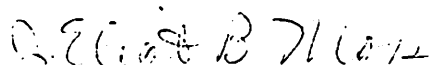
Approved as to style and content by:



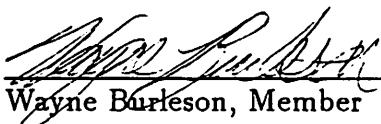
David W. Stemple, Chair



Robert N. Moll, Member



J. Eliot B. Moss, Member



Wayne Burleson, Member

Arnold L. Rosenberg, Acting Chair
Computer Science Department

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Professor David Stemple, for the support and the encouragement he gave me. I wish to thank the member of my committee Professor Eliot Moss for his guidance and his painstaking proof-reading. I am also grateful to the other members of my committee, Professors Robbie Moll and Wayne Burleson, for their valuable remarks which helped me to improve the quality of this work.

Many of the ideas presented in this thesis are influenced by the work of Professor Tim Sheard. The analysis of the expressiveness of the uniform traversal combinator algebra is done by Sushant Patnaik. Special thanks to Joydip Kundu and to Bennet Vance who proof-read parts of this thesis.

I wish to thank Professor Subhasish Mazumdar for his friendship and for the long and interesting discussions we had on many aspects of computer science.

Finally, I would like to thank my parents, Eleni and Iakovos, my brother, George, and my friends, Helene Hinis, Marina Stinga, Olga Kikou, Kimie Sekene, Thymios Delis, and Phoebe Jackson, for giving me the emotional support that I needed during my long journey that ended in this thesis. Without their love and understanding, it would not have been possible.

ABSTRACT

A TRANSFORMATIONAL APPROACH TO DATABASE SYSTEM IMPLEMENTATION

FEBRUARY 1993

LEONIDAS FEGARAS

B.E.E., NATIONAL TECHNICAL UNIVERSITY OF ATHENS, GREECE

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS

Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor David W. Stemple

The compilation of data intensive application programs involving persistent data into efficient implementations needs to consider multiple execution schemes. A clean separation of an application specification and its implementation can increase the number of implementation choices for a single specification. While current database systems offer data independence, their ability to capture complex specifications and computations is very limited. This affects the application performance as it forces incompatible specifications to be shaped into forms that can be captured by the few primitives these languages support. Most programming languages on the other hand offer satisfactory computational power and a large variety of data objects to choose from but support limited persistence. Here concrete implementations of abstract expressions are expanded hierarchically through layers of abstraction rather than generated from alternatives.

This dissertation bridges these two approaches by giving the database designer the ability to specify how the abstract objects defined in a program are to be mapped into the storage structures provided by the database, leaving the translation and optimization of the abstract operations to the compiler. We have developed a formal model that describes this process, called the type transformation model, based

on Darlington's work on transformational programming. The type transformation model provides a method of translating any abstract operation that manipulates abstract objects into an operation that manipulates concrete objects. The translation is based on the dependencies between the abstract and concrete objects, stated by an implementation designer. We present a new algebra that facilitates this task, called the uniform traversal combinator algebra. Database queries on bulk data types, such as on lists and sets, are captured as well-formed recursive functions, called traversal combinators, whose definition is derived from the inductive properties of the structure being traversed. One important contribution of our approach is the treatment of the class of traversal combinators resulting from restricting their input functions to be themselves traversal combinators. This introduces a very disciplined and uniform treatment of programs where any program is a traversal or some other very simple primitive. This uniformity simplifies the query translation and optimization process and facilitates the verification of the resulting translation.

TABLE OF CONTENTS

	<u>Page</u>
ACKNOWLEDGMENTS	iv
ABSTRACT	v
LIST OF FIGURES	x
CHAPTER	
1. INTRODUCTION	1
1.1 Our Methodology	5
1.2 The Technology	8
1.3 Thesis Organization	11
2. RELATED WORK	12
2.1 Program Transformation Systems	12
2.2 Type Transformation Systems	15
2.3 Reductions and Stereotyped Recursions	16
2.4 Query Algebras	19
2.5 Query Optimization	22
2.6 Flexible Database Management Systems and Translators	23
3. THE TYPE TRANSFORMATION MODEL	26
3.1 The ADABTPL Language	26
3.1.1 Function Types	27
3.1.2 Structure and Singleton Types	27
3.1.3 Union Types	28
3.1.4 Parameterization	28
3.1.5 Recursion	29
3.1.6 Finite Sets	30
3.1.7 Restriction	30
3.1.8 Bounded Parameterization	32
3.2 Some Generic Manipulators of Structures	33
3.3 Mapping Sets into Lists	36
3.4 The Formal Framework	38

3.5	Schema Evolution and Data Restructuring	44
3.6	Implementation of the Set Union	46
3.7	Undecidability of the Translation Problem	50
4.	UNIFORM TRAVERSAL COMBINATORS	52
4.1	Formal Definitions	55
4.1.1	The List Traversal Combinator	56
4.1.2	The General Form of Traversal Combinators	57
4.1.3	The Integer Traversal Combinator	58
4.1.4	The Boolean Traversal Combinator	59
4.1.5	The Tuple Traversal Combinator	59
4.2	Properties of Traversal Combinators	59
4.3	Structural Equality	61
4.4	Uniform Traversal Combinators	63
4.5	Composition of Uniform Traversal Combinators	67
4.6	Using the Composition Algorithm for Program Optimization	71
4.7	Unrestricted Uniform Traversal Combinators	73
4.8	Model Extensions	75
4.8.1	Capturing all Recursively Defined Types	76
4.8.2	Finite Sets	81
4.8.3	Vectors and Strings	87
4.8.4	Objects with Object Identity	88
4.8.5	Second-order Functions	95
4.9	Use of Uniform Traversal Combinators	96
5.	THEOREM PROVING AND PROGRAM SYNTHESIS	98
5.1	Equality of Uniform Traversal Combinators	99
5.2	Theorem Proving	103
5.3	Proving Set and Vector Theorems	105
5.4	Proving Theorems about Restricted Types	106
5.5	Proving Second-order Theorems and Synthesizing Programs	108
5.6	Proving Meta-theorems	113
5.7	Using the Theorem Prover for Program Translation	115
6.	PROGRAM TRANSLATION AND OPTIMIZATION	118
6.1	The Transformation Specification Language	120
6.2	Database Implementation	124
6.3	Cost Model	127
6.4	The Optimization Algorithm	135
6.5	Translating Encapsulated Functions Incrementally	141
6.6	The Concrete Layer	144
6.7	Conclusion	148

7. CONCLUSION	149
7.1 Summary	149
7.2 Contributions	152
7.3 Future Research	152

APPENDICES

A. THE USER-LEVEL LANGUAGE	156
B. THE TRAVERSAL COMBINATORS IN CATEGORICAL TERMS	165
BIBLIOGRAPHY	168

LIST OF FIGURES

Figure	Page
1.1 The translation process	6
3.1 Mapping a set(person) into an ordered list	37
3.2 Translation of the abstract operation union	38
3.3 Mapping an abstract function f into a concrete operation F	42
4.1 The composition algorithm	68
4.2 Composition of set traversals	84
4.3 Composition of vector traversals	88
4.4 Composition of object traversals	94
4.5 Composition of second-order traversals	95
5.1 Equality of uniform traversal combinators	100
5.2 Solution of $\mathcal{E}(\mathcal{F}_T(c, r_1, \dots, r_n)(f), g, \rho, \beta)$	111
6.1 The constructor tree of set(int)	128
6.2 The constructor tree of the part-subpart database	130
6.3 Definition of the cost function $\text{cost} = \mathcal{H}_C(\phi_1, \phi_2, \phi_3, \phi_4)$	132
6.4 Optimization of $\mathcal{F}_T(c, r_1, \dots, r_n)(f) = g$	138
6.5 A best-first search solution of $\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow F(x_1, \dots, x_n) = g$	140
A.1 Translation of UTCL expressions	157
A.2 Translation of UTCL imperative statements	160
B.1 The list traversal combinator $h = \text{tc.list}(f_1, f_2)$	167
B.2 The traversal combinator $h = \mathcal{H}_T(f_1, \dots, f_n)$	167

CHAPTER 1

INTRODUCTION

Current database management systems, such as systems based on the relational model, lack the expressive power to support complex queries in a high level language and typically provide limited variety in their data structures. The limitation on the expressive power of the high level languages in these systems is commonly circumvented by embedding query languages in more expressive conventional programming languages. The disadvantage of this approach is that it forces the database designer to express his/her conceptual model in terms of two different semantic models. Additionally, for programming language objects to become database objects (or vice versa) they often need to be restructured into the data model of the database system being used. Such restructuring incurs costs both in conceptual terms for programmers and in resources used when performed. Examples of this problem occur in applications such as rule-based systems and computer-aided design. In these applications complex structures are often used in programming the algorithms that access rules and designs. These programming language structures need to be "flattened" in order to be stored in relational structures, for example. The cost of restructuring can make it difficult to meet the efficiency requirements of these applications. In addition, whenever hierarchical or graph objects are mapped into flat structures some semantic information available from the graph specification is lost. Such information could substantially facilitate the query optimization process as it specifies explicitly all valid access paths to data, and thus avoids considering impossible paths.

One important advantage of commercial database systems is that they offer data independence, whereby abstract objects and the operations upon them can be significantly independent of their implementations. In these systems, database queries are expressed in an abstract language, usually in a declarative form, that does not indicate how the query is to be computed but only what properties the answer should satisfy. This offers many opportunities for optimization and, consequently more efficient implementations, as there are many programs to choose from that compute the same query. In addition, the database designer can intervene in the program translation and optimization process by specifying the storage structures that implement the abstract objects. In a relational database system, for example, a database designer may choose the implementation of a database table from a number of possible alternatives, such as attaching extra indexes to a table. This decision does not affect how queries are expressed in the database language but only how they are compiled and optimized. Data independence facilitates user productivity as changes to implementations do not require changes to the queries. Furthermore, some of these systems provide a restructuring mechanism to change the implementation of parts of the database or to modify the database schema itself without losing any stored data.

On the other hand, most programming languages offer satisfactory computational power and a large variety of data objects to choose from but support limited persistence, usually in the form of files. Some of these languages provide an abstract data type mechanism to support a layered model of programming, where abstract objects and the operations upon them are defined in terms of other, more implementation oriented, operations. This layering provides a type of data independence because, as in the database systems, it hides the implementation details from the user of the abstract data type. The difference is that an abstract data type typically has its operations decomposed into their implementations in a

predefined, rigid way, while a database system offers a number of alternative methods of execution. For example, there may be more than one access path to retrieve the same piece of information from a database, such as accessing an index in a relational system to retrieve pieces of information from a table or accessing the table itself. This redundancy of access paths makes it possible for a database system to perform query optimization, in which concrete implementations of abstract expressions are generated from alternatives, rather than expanded hierarchically through layers of abstraction. A rigid refinement style of implementation does not meet the needs of large data-intensive applications. It does not offer sufficient flexibility to solve the problems of evolution of such systems and it does not support optimization techniques exemplified by query optimization in database systems. This rigid refinement of implementation introduces another big disadvantage. There are times when a sequence of function calls does redundant computation in which the same data is computed twice, once in each function. Relational database systems suffer from a similar problem that can be solved by materializing intermediate relations between queries or by performing a kind of global query optimization, where sets of queries are considered and optimized together. To achieve this type of optimization in a regular programming language, one needs to unfold the body of each such function definition, put it in place of the call (as inline functions do), rearrange the resulting operations, and store some of the intermediate results in order to avoid the recomputation overhead. This optimization is difficult to perform using current compiler technology.

There has been some efforts to bridge database management technology with programming language philosophy to form persistent and database programming languages [2]. Some of these efforts are based on the object-oriented model [12, 1, 3]. Objects defined in such languages can have sophisticated structures, like those found in modern programming languages, and can be persistent across a wide variety

of types. Persistent and non-persistent objects are treated more uniformly than in ordinary languages, hiding the difference in persistence from the programmer of the operations that work on these objects. Most of these systems suffer from the same problem that non-persistent programming languages have: all abstract objects and operations are implemented via rigid translations. For data intensive applications, where most of the data need to be persistent, this problem becomes more important because the database accesses are relatively slower than the primary memory accesses. This may result in a system that is unnecessarily slow, because the programmer does not have enough flexibility in choosing how abstract objects are stored and the translator does not examine alternative ways to perform the queries using different access paths and algorithms.

In summary, we are concerned with data intensive application programs that involve persistent data. The compilation of these programs into efficient implementations needs to consider multiple execution schemes. For example, considering different paths for accessing the same piece of data increases the opportunities for optimization. A clean separation of an application specification and its implementation can increase the number of implementation choices for a single specification. While current database systems offer data independence, their ability to capture complex specifications and computations is very limited. This affects the application performance as it forces incompatible specifications to be shaped into forms that can be captured by the few primitives these languages support. This problem can be solved by extending the expressiveness of these database languages to capture more complex data structures and computations. There are already many efforts in that direction. People involved in such efforts tend to believe that the more you extend the expressiveness of a language the more difficult the program optimization task becomes. This thesis will demonstrate that this is not necessarily true by presenting a language that is expressive enough to capture most polynomial time functions but

still facilitates optimization. We believe that the undecidability of optimization is not managed by reducing the expressiveness of the language but by reducing the number of possible program schemes that compute a function. The search for an optimal solution is more efficient if there is a smaller number of program schemes to consider. A provision must be taken, though, that the optimal solution is within this search space. This restriction of the number of program schemes is achieved by requiring that all structure traversals be expressed in terms of a very small number of stereotyped generic recursions.

This thesis presents a framework for translating abstract database specifications into concrete implementations in a way that supports data independence. Both the specification and implementation are expressed in a language rich enough to capture very complex data structures and computations but restricted enough to facilitate program translation and optimization.

1.1 Our Methodology

This thesis explores the possibility of achieving a level of data independence similar to that found in database systems but in the context of a high-level database programming language, such as ADABTPL [26]. We believe that the best approach to this problem is to give the database designer the ability to specify how the abstract objects defined in a program are to be mapped into the storage structures provided by the database, as well as to specify the implementation of some of the abstract operations, but to leave the compiler to translate and optimize the rest. A database storage structure in this approach is not just an abstract data type that implements an abstract object. Here the designer must give explicitly the *dependency* between the abstract and the storage objects. A well-suited means of expressing this dependency is the use of a representation function to map the concrete object into the abstract object. The compiler will use these dependencies to translate the operations

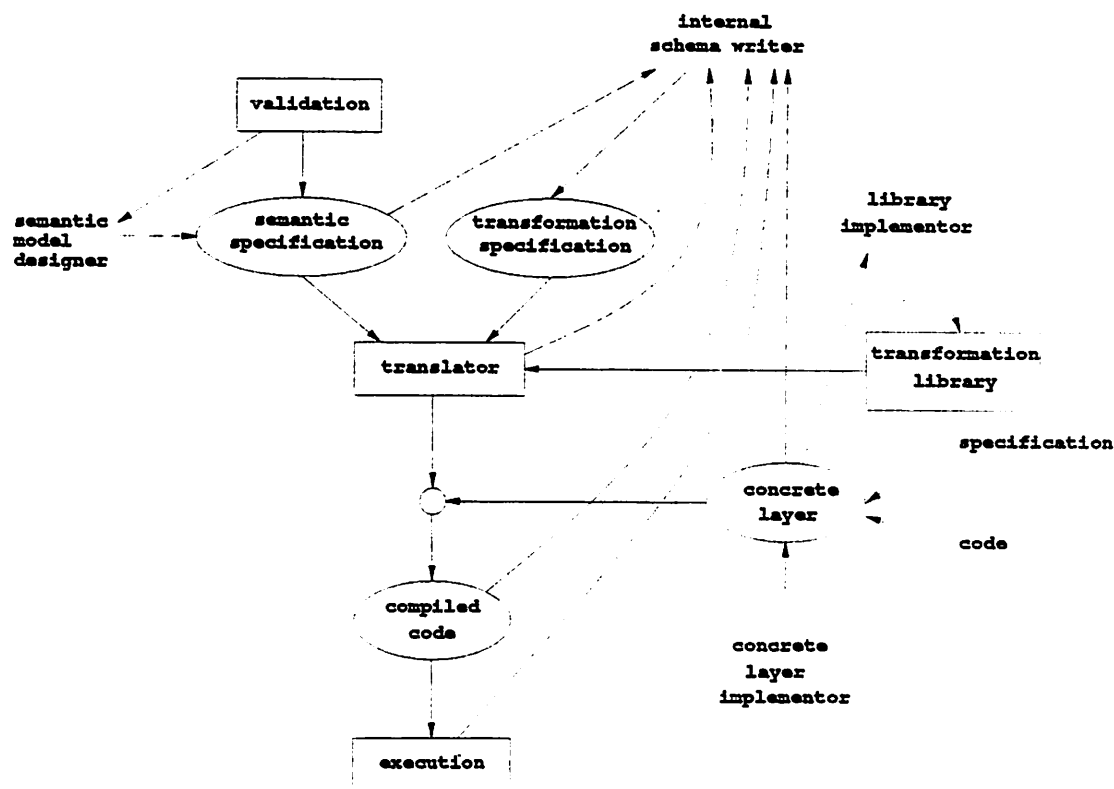


Figure 1.1 The translation process

and to assure the correctness of the translation. We have developed a formal model that describes this process, called the **type transformation model** [28], based on Darlington's work on transformational programming [23, 16, 46]. The dependencies between abstract objects, which can be expressed as integrity constraints, as well as the dependencies between the storage objects suggest that there may be more than one access path to retrieve the same data, since there is redundant information. The query translation process is a search over these alternatives guided by cost functions.

The necessary separation of specification from implementation requires that the translation be done in stages by people with different expertise. Figure 1.1 shows the necessary stages for compiling an abstract program. The ovals represent forms of specifications from abstract to concrete, the rectangles processors, and the unboxed

text describes people in various modes. The **semantic model designer** is the person who writes the abstract application program (the *semantic specification*) using our abstract specification language. This person should not be concerned with assigning storage structures to the abstract objects to achieve efficient execution. The main concern here is to write a functionally correct specification satisfying all the design requirements. The translator type-checks the specification and uses a theorem prover to find inconsistencies in the specification and to suggest corrections to overcome them. The semantic model designer can compile the specification into a non-persistent prototype using a simple translation, not shown in the figure, in order to check the behavior of the specified system. The whole design process can be repeated until the specification is consistent and the specified system behaves as intended.

After the program is proved to be syntactically and semantically correct, a second stage is initiated by the **internal schema writer** who assigns storage structures to all the abstract objects by defining explicitly the type transformations that will achieve the object translations. All these type transformations will constitute the *transformation specification* of the abstract program. To make this task easier, a rich library of type transformations, called the *transformation library*, will be provided. This library contains the transformations that are commonly used, such as the mappings of abstract sets into B^+ trees, attaching indexes to a set, etc. The task of writing this library is assigned to the **library implementor**. The internal schema writer is left with the task of selecting mappings from the library that suit this specific application, and defining new mappings when none of the mappings in the library are satisfactory. The code that manipulates the database storage structures constitutes the *concrete layer* and it is written by the **concrete layer implementor**. The internal schema writer as well as the library implementor need to know about this layer in order to write valid mappings. If the concrete layer is

changed, usually by the introduction of new storage structures, this will affect both the transformation library and the transformation specification.

The task of the compiler is to accept the semantic specification along with the transformation specification and produce the interface to the concrete layer. The theorem prover is used again to verify the correctness of the translation by proving whether the translation of each abstract operation manipulates the storage structures according to the defined object mappings. After this, the produced object code must be linked with the concrete layer library to yield executable modules. The internal schema writer will evaluate the performance of some of the functions and of the whole system to decide if there is an alternative way of implementing some of the abstract objects to improve the system performance. The process of the transformation specification is repeated until the system performance is acceptable.

It is very common for either the system specification or implementation to change, because the needs of most applications evolve through time. Decisions about schema evolution are made by the semantic model designer, while the internal schema writer is responsible for the changes in the implementation. It is desirable to restructure the database efficiently to reflect these changes, instead of destroying and recreating it from scratch. This task is performed by the internal schema writer who writes the required mappings to transform the old abstract objects to the new abstract objects. These mappings are similar to the mappings from abstract objects to concrete. The compiler uses this information to generate and optimize a program that restructures the database.

1.2 The Technology

The type transformation model provides a method of translating any abstract operation that manipulates abstract objects into an operation that manipulates concrete objects, according to the dependencies between the abstract and concrete

objects. This concrete program is equivalent to a program that translates the input concrete objects, performs the abstract operation, and then translates the resulting abstract object into a concrete object. The program optimizer needs to transform this program in such a way that it does not perform any unnecessary object translation and, more importantly, it is more efficient than other equivalent programs that compute the same function. Avoiding the object translation overhead is very important because we want our implementations to be expressed in terms of concrete primitives only, not in terms of mixed abstract and concrete computations. Therefore a necessary criterion for a successful application of the type transformation model is to specify a language that guarantees, and facilitates in a systematic way, that this default translation can always be transformed into a program consisting of concrete primitives only.

This thesis introduces a new algebra, called the **uniform traversal combinator algebra** [27], that fulfills the requirement of the previous paragraph. Furthermore, it facilitates program optimization and validation. Bulk data structures are defined inductively by a small number of type constructors. There is only one primitive in this algebra for traversing structures: *the traversal combinator*, which is defined inductively according to the recursive type definition of the structure being traversed. No explicit recursion is permitted here as it is hidden in these primitives. Each input function of a traversal combinator is associated with an object constructor of the type being traversed and it is restricted to be an operation in our algebra as well, such as another traversal. This introduces a very disciplined form of programs where everything is a traversal or some other very simple primitive. Programs are restricted to include traversals that traverse variables only, not the results of other computations such as other traversals. Consequently, having the default translation (derived from the type transformation model) in such form means that this program is expressed in terms of concrete primitives only, as

no nested traversals are permitted. Despite this limitation, our algebra is closed under composition. This basically means that traversals over traversals can be reduced to traversals over variables. Traversals over traversals appear very often, especially when we compose two functions that traverse their inputs. We have developed the composition algorithm that performs this reduction. This reduction algorithm can be used to transform the default translation into a program with no object translation overhead expressed in concrete primitives only. Furthermore, this algorithm is the basis of a theorem prover that can prove or disprove any first-order theorem expressed in our algebra in a very systematic and efficient way. The restriction that only variables that are not results of other traversals are allowed to be traversed offers very few ways of expressing functions as programs in our algebra. This substantially limits the search space of the optimization, making the program translation and optimization process very effective and efficient, even though some alternative program schemes may not be considered.

Our uniform traversal combinator algebra and the composition algorithm can be used in many domains other than database system implementation and theorem proving. The second-order equality algorithm, which is a part of the theorem prover and the program optimizer, can be used by itself as a second-order unification algorithm and also in program synthesis. In addition, the composition algorithm is a very powerful reduction algorithm that can be used for optimizing functional programs. More specifically, this algorithm completely automates the unfold-simplify-fold method [16], which is used for deriving optimal expressions from composite terms. It also completely automates deforestation [80], because the programs derived by this algorithm produce fewer intermediate results than the composite initial programs.

1.3 Thesis Organization

This thesis is organized into seven chapters and two appendices:

- Chapter 2 presents the related work that influenced our approach.
- Chapter 3 describes some of the features of the ADABTPL specification language and presents the type transformation model.
- Chapter 4 presents the uniform traversal combinator algebra and its extension to capture other complex structures such as sets and vectors.
- Chapter 5 describes a method of using the uniform traversal combinator algebra to prove theorems and to synthesize programs.
- Chapter 6 describes a method of using the type transformation model and the uniform traversal combinator algebra to translate and optimize database programs.
- Chapter 7 summarizes the thesis and presents its contribution and our future research plans.
- Appendix A presents a query language based on the uniform traversal combinator algebra, but more user-friendly.
- Finally, Appendix B explains the definition and some of the theorems on traversal combinators using basic category theory.

CHAPTER 2

RELATED WORK

The related work falls into these categories: Section 2.1 presents the work on program transformation and on program development by stepwise refinement. Section 2.2 presents the related work on type transformation. Section 2.3 presents the work on reductions and other stereotyped recursions similar to our traversal combinators. Section 2.4 presents some of the query algebras that use primitive operations over their bulk structures that are similar to our traversal combinators. Section 2.5 reviews some of the query translation and optimization systems that influenced our approach. Finally, Section 2.6 presents some systems that have the same motivation with our approach in achieving flexibility and adaptability.

2.1 Program Transformation Systems

Most semantic query optimization systems use program transformation methods to refine and simplify queries. There is a lot of work on transformational programming and program synthesis [65, 61]. The first such work was by Burstall and Darlington [16, 21, 20, 22, 23]. The basis of a transformation system is a set of *correctness-preserving transformation rules* that map a program scheme to another preserving the intended meaning of the original program. A *transformation system* does program construction by successively applying these transformation rules. If the rules preserve correctness, then this system guarantees that the final version of the program will still satisfy the initial specification. Most transformation systems

use three forms of transformation rules: *unfold* is the replacement of a call by its body with appropriate substitutions; *rewrite* is the application of a rewrite rule to a program scheme that matches the pattern of the rule head returning the rule body with the rule variables properly instantiated; *fold* is the replacement of some piece of code by an equivalent function call. Fold is the reverse operation of unfold. Both fold and unfold can be seen as special types of rewrite rules. Typically program transformation systems are semi-automatic programs that require user intervention whenever decisions need to be made, by selecting which rule to apply next. Most of the early work on transformational programming was on recursive programs [10, 25, 49], especially on replacing simple forms of recursion by iteration.

The Boyer-Moore theorem prover [14] is a transformation system that proves theorems by transforming them to the 'true' expression. It uses lemmas and meta-lemmas as rewrite rules, along with some very powerful heuristics such as a general form of structural induction, to reduce a theorem into a simpler one. Another application of transformational programming is program synthesis. The goal of program synthesis is to use the formal description of a program, usually in a form of pre- and post-conditions that this program satisfies, to generate code that computes this program. One of the first works in this area was by Manna and Waldinger for the DEDALUS system [52, 53, 54]. Program synthesis is related to second-order unification [41]. This is the unification between second-order terms, that is, terms that contain variables that are bound to functions (second-order variables). For example, the second-order pattern $f(A(g(B)), C)$, where A , B and C are constants and f and g are second-order variables, matches the program scheme 'while C do $A(B)$ ', by using the second-order substitution list $\sigma = \{f \leftarrow \lambda u \lambda v. \text{while } v \text{ do } u, g \leftarrow \lambda w. w\}$. One matching algorithm for second-order terms is described in [41] and it is used for formalizing a general program transformation technique that includes recursion removal. Second-order and higher-order unification are undecidable in general. In [73]

higher-order unification consists of four transformation rules: *term decomposition* where two calls to the same function are unifiable if their associated arguments are unifiable; *removal of trivial pairs* such as equal terms; *variable elimination* when we have solved the system for some variable; and the *imitation rule* that applies whenever a variable v is unifiable to a function call $a(\bar{u})$: if v is a first-order variable then it generates the binding $v \leftarrow a(\bar{y})$, where each y_i is an unknown first-order variable, otherwise it generates the projection $v \leftarrow \lambda\bar{x}.x_i$ or the binding $v \leftarrow \lambda\bar{x}.a(Y_1(\bar{x}), \dots, Y_n(\bar{x}))$, where each Y_i is an unknown higher-order variable. In the latter case the other terms are simplified by the new bindings using β -reductions.

Systems for developing programs by stepwise refinement [65, 57, 82] have the same motivation as the program transformation systems, but they are domain specific. Program refinement is the process of refining an abstract specification, which is not necessarily executable, to a concrete program, and proving the validity of each refinement step. In each refinement step a small choice is made about the form of a program that need to be added to satisfy the specifications. That way the undecidability of the program synthesis problem is managed by splitting the problem into simple subtasks. If a decision turns out to be invalid then it is withdrawn and the process is backtracked to a point where it can be resumed again. The abstract specification is usually represented as a pair of assertions, where the first, the input assertion, defines the set of admissible input states, while the second, the output assertion, defines the set of correct output states for each input state. The refinement steps are supplied by the user and they usually need to be proved correct before they are inserted, so that they do not need to be validated when they are instantiated and applied as refinement steps. One such method of validating rules before they are used is by proving that they satisfy the Church-Rosser and the termination properties. This guarantees that the rewrite system is a reduction system that always yields the same solution regardless of the order in which the

steps are applied. Two such systems are OBJ2 [34] and the Vienna Development Method [44, 11].

2.2 Type Transformation Systems

Our type transformation model is based on Darlington's work [23, 16]. He proposes a method for 'flattening' programs, that is, generating programs from abstract functions in terms of the lowest level language primitives. The user needs to specify the representation function that maps each lower data type into a higher one. This function, which is in general many-to-one, is used, along with some manual program transformation methods, to compile an abstract operation. His model is very close to the model described in Chapter 3. A similar approach was adopted by Kapur and Srivas [46], based on the theory of term rewriting systems. It uses the Knuth-Bendix completion procedure to test whether the insertion of a rule introduces inconsistencies (contradictions) to the rule database.

The Polya project [39, 38], undertaken by David Gries and other researchers at Cornell University, introduces a new programming language construct, called the *transform*, for expressing coordinate transformations. A coordinate transformation is very similar to a data type transformation. Each transform describes one coordinate transformation from a set of abstract variables to a set of concrete variables. It also includes explicit transformation rules to map each individual expression or statement that works on abstract variables into the corresponding one that works on concrete variables. Each part of an abstract program must match exactly with one of these patterns in order to be compiled. Furthermore, Polya uses coupling invariants as constraints between abstract and concrete variables to be used for proving the soundness of the transformation process. The Polya type transformation model is more general than ours, because it supports partial implementation of types, where only some of the operations upon a type are implemented leaving the rest

unspecified. In [39] they claimed that the approach of using a representation function from a concrete domain to an abstract domain is too restricted and unsuitable, because this function in some cases need not exist, in that several abstract values are represented by a single concrete one. This is not true for ADABTPL where dependencies between parts of an abstract object must be expressed as integrity constraints. Therefore, if two different abstract objects that are instances of the same type have the same implementation, there is a hidden dependency in this type that should be expressed as a constraint. In addition, we believe that even though there are cases where we need partial implementation of an abstract type, where only a subset of operations is supported, these cases are not too many to justify adopting such a general programming paradigm. Since it may be not obvious to a user whether a given operation matches any of these patterns, such a type transformer lacks the ability to define a priori the limits of its applicability to operations. Another difficulty coming from this approach is that operation translation is rigid, insensitive to cost, and does not depend on context. This could make optimization of operations with redundant computation very difficult.

2.3 Reductions and Stereotyped Recursions

Our algebra is motivated by the intention of capturing a large family of type-safe stereotyped recursions with well-understood properties that facilitate theorem proving, program synthesis, and program translation and optimization. There is some research lately on analyzing the properties of some highly stereotyped recursions similar to our uniform traversal combinators, but there is no work reported on defining a systematic and completely mechanized method for applying these properties to theorem proving and program optimization. This section reviews these related approaches.

Our work is highly influenced by Sheard's work on *reductions* [70, 69]. Reductions are convenient abstractions for expressing manipulations of bulk data types represented by recursive structures. They accumulate results as they traverse a structure and can be used for more computations than are expressible using a mapping traversal. Reductions over lists and finite sets are expressive enough to directly simulate all primitive recursive functions [42]. Reductions tailored to particular recursive types can be generated automatically by a compiler by examining the type details. The most notable contribution of Sheard's work was using compile-time linguistic reflection [72] to generate user-defined stereotyped recursions for any type by expressing the program generators in a strongly-typed meta-language which is the same as the language it represents. Our program generators that compute uniform traversal combinators are expressed in this language.

The most influential work on realizing the importance of capturing recursions into a few powerful patterns was by Richard Bird [10]. Even though his work was focused on specific types, such as lists, it suggested ways of extending these methods to other types. He used the idea of *promotion* for solving various transformation problems, such as deriving an efficient solution for the problem of detecting cycles in a graph. The basic idea of promotion is very simple: suppose that $H(\text{nil})$ is some constant and $H(\text{cons}(a,x))=h(a,H(x))$. We want to put $g(x)=f(H(x))$ into an inductive form similar to that of H . A sufficient condition for this is when there exists some function h' such that $f(h(a,y))=h'(a,f(y))$ (also called continuity condition). This is called *promotion*, as it pushes f to the right of the expression g in order to take advantage of the recursive properties of H . If f does not satisfy the above condition then we can introduce a generalization f' of function f that has one extra parameter. That is, $f'(c,x)=f(x)$ for some constant c . Then the generalization g' of g (where $g'(c,x)=g(x)$) is $g'(s,x)=f'(s,H(x))$ and the condition becomes $f'(s,h(a,y))=h'(a,f'(u(a,s),y))$, for some g' and u . In that case $g'(s,\text{cons}(a,x))=h'(a,g'(u(a,s),x))$, that is, g' is defined

inductively. This method of generalization is called *parameter accumulation* and is used as an optimization technique for making promotion effective. Note that g' does not have its first parameter smaller at each recursive step any more, a property required in our approach.

Mike Spivey adopted a categorical approach for expressing list recursions and their properties [74]. He defined list operations in terms of six primitives: list *map*; the *flatten* function, which concatenates the members of a list of lists; *singleton*, which takes a value and creates a singleton list; list *concatenation*; the *empty list*; and list *reduction*, where the accumulator is an associative operation. This type of reduction, where the accumulator is associative, forms a *monoid* in category theory. We will encounter very often this kind of restriction in various object algebras as it satisfies some nice properties.

The work of Grant Malcolm on *homomorphisms* [51] generalized these methods for all types. His paper introduced a generic form of homomorphisms, very similar to our traversal combinators, stated the promotion theorem for any homomorphism, and used it for proving some general properties of recursive types. Other works with similar definitions of recursive patterns, called *iterative functions*, were by Bohm and Berarducci [13] and by Pfenning and Paulin-Mohring [62], based on the second order lambda calculus. The categorical programming language Charity [18] introduces the *fold* operator, which is very similar to our traversal combinator but more generalized to capture lambda currying explicitly and uniformly. This language can also be used for computations in categories other than the category of types (which is what a programming language normally does), such as expressing derivations in the proof theory.

The most complete work on analyzing the properties of stereotyped recursions was by Meijer, Fokkinga, and Paterson [56]. They presented four classes of generic functions: *catamorphisms* similar to Malcolm's homomorphisms; *anamorphisms*

that can be seen as the inverses of catamorphisms; *hylomorphisms* a more general form of anamorphisms; and *paramorphisms* similar to our traversal combinators. They proved a large number of generic theorems for each class, such as the fusion law (similar to our promotion theorem) and the uniqueness property (for proving equalities of two functions).

Another approach on stereotyped recursions is using *monads* and *monad comprehensions* [81]. A monad in category theory is a triple (map, unit, join) that satisfies some properties, such as $\text{map}(g \circ f) = \text{map}(g) \circ \text{map}(f)$. For example, for lists, unit could be like Spivey's singleton, map like list map, and join like flatten. By specifying this triple for a specific type we can define a monad comprehension as $[t|q]$, where t is a term and q is a qualifier. $[t|q]$ is defined by the following rules: $[t|] = \text{unit}(t)$; $[t|x \leftarrow u] = \text{map}(\lambda x.t, u)$; and $[t|(p;q)] = \text{join}([t|q]|p)$. For example, $[(x,y)|x \leftarrow [1,2]; y \leftarrow [3,4]]$ is equivalent to $[(1,3), (1,4), (2,3), (2,4)]$. The query algebra described in [76] is based on comprehensions over bulk data structures.

2.4 Query Algebras

Our uniform traversal combinator algebra is influenced by the *set-reduce algebra (SRL)* [71]. The most important operation in this algebra is the set reduction $\text{set-reduce}(s, \text{app}, \text{acc}, \text{base}, \text{extra})$ that applies function app to every element of the set s and accumulates the results using the binary accumulator acc starting with base . Parameter extra contains all extra parameters needed by the app function, normally done implicitly using currying in other algebras. That is, if $s = \{a_1, \dots, a_n\}$, then the previous reduction is $\text{acc}(\text{appl}(a_1, \text{extra}), \text{acc}(\text{appl}(a_2, \text{extra}), \dots, \text{acc}(\text{appl}(a_n, \text{extra}), \text{base})))$. For example, the set intersection of r and s is computed by:

```
set-reduce(r, [x,y] → if member(x,y) then insert(x,emptyset) else emptyset,
          union,emptyset,s)
```

Here the application function `appl` has an extra parameter `y` bound to `extra`, which here is `s`. Therefore, for each element `x` in the set `r` we return `{x}` if `x ∈ s` or the empty set otherwise. These results are unioned together using the accumulator `union` starting with the empty set. In the SRL algebra all parameters to `set-reduce` are SRL expressions, which can be `set-reduce` calls, set constructions (that is, `emptyset` or `insert`), tuple constructions, tuple selections, equalities, or operations over primitive types, such as booleans and integers. This algebra captures all polynomial time functions [42].

Another approach is exemplified by the polymorphic functional programming language Machiavelli [59, 15], enhanced with a set data type and database operations, such as `join` and `projection`. A set type `{τ}` in Machiavelli can be defined for any type `τ` that supports equality, such as `τ` being another set type. There are four set primitives defined: the empty set `{}`, the singleton constructor `{x}`, set union, and the *hom* operator similar to `set-reduce`: $\text{hom}(f, op, z, \{\}) = z$ and $\text{hom}(f, op, z, \{x_1, \dots, x_n\}) = op(f(x_1), \dots, op(f(x_n), z))$. A *hom* operation is proper when `op` is associative and commutative. In [15] a new algebra was presented based on two equivalent programming constructs for doing structural recursion over sets: 1) $h = \Phi(e, f, u)$ that satisfies $h(\emptyset) = e$, $h(\{x\}) = f(x)$, and $h(S_1 \cap S_2) = u(h(S_1), h(S_2))$, and 2) $g = \Psi(e, i)$, a restricted version of *hom*, that satisfies $g(\emptyset) = e$ and $g(\text{Insert}(x, S)) = i(x, g(S))$ where i is a commutative-idempotent monoid, that is, it satisfies $i(x, i(y, S)) = i(y, i(x, S))$ and $i(x, i(x, S)) = i(x, S)$. A certain sublanguage of this language can simulate all operators from relational algebra and it can even capture transitive closure.

The set-oriented language FAD developed at MCC [77, 78] supports nested sets. It is based on the filter expression `filter(f, s1, ..., sn)` that returns a set composed by the results of applying the `n`-ary filter function `f` to each element of the cartesian product of the sets `si`. The optimization process performs algebraic transformations

to the filter calls and assigns annotations to them, indicating how the filter is actually implemented.

The object-oriented data model and language described in [8] use the reduction operation pump, which is similar to Spivey's reduce operator as it requires an associative and commutative accumulator. In addition, this algebra supports the apply-to-all operator, similar to map, and the cartesian product operator. Expressions in this algebra can be optimized using some powerful generic rules, some of which reflect familiar heuristics for relational query optimization. One is vertical loop fusion that combines two loops by creating a single loop whose body is the composition of the bodies of the two original loops. Another is horizontal loop fusion that combines two independent loops over the same value into a single one that executes both the bodies of the loops.

Bennet Vance uses only one bulk primitive in his algebra: the fold operator [79], which is similar to Spivey's reduction operator, that is, the accumulator is required to be associative:

$$\begin{aligned} [x_1, x_2, \dots, x_n] \text{fold}(u, f, \oplus) &= f(x_1) \oplus f(x_2) \oplus \dots \oplus f(x_n) \\ [] \text{fold}(u, f, \oplus) &= u \end{aligned}$$

Like Spivey's language, each type here is defined by three primitives: empty, singleton, and concatenation, an associative operation. For example, concatenation for lists is append, for multisets union, and for arrays array concatenation. By expressing all bulk operations as folds and by using some very powerful generic laws about fold expressions one can achieve very effective semantic query optimization. However, the requirement that the accumulator be associative restricts the expressiveness of the language.

2.5 Query Optimization

Much work has been done on query optimization for commercial database management systems [43]. Most of these projects address the problem of selecting the best evaluation strategy (a query evaluation plan) for a complex query, expressed as a join in a relational database system. The most influential work in this area was done for System R [67]. They introduced a *selectivity factor* for the restriction predicate of a join to be an estimate of the number of tuples that satisfy this predicate. The query optimizer needs to retrieve statistics from the database, such as index cardinalities, to compute these factors. For an n-join query the optimizer uses heuristics to reduce the join order permutations to be considered. For each such permutation, and for each join access method, it generates a candidate access plan. The cost of each such plan is predicted by using the selectivity factors to estimate the size of the intermediate composite relations. The optimizer selects the candidate access path with the minimum total cost.

The optimization process is very complex and often evolves through time, as new implementations for the database objects are introduced. Rule-based optimizers offer enough flexibility to solve these problems. For these systems the optimization problem is reduced to a search over all transformation rules applied to a specific query with the ultimate goal of minimizing the execution cost of the query. The rule-based systems are very suitable for semantic query optimization, such as the simplification of the boolean expression that forms the selection predicate of a relational query. Freytag's work on ruled-based optimization [32, 33, 31, 30, 29] is an example of such approach, but it does not use cost functions to guide the search. The best work on flexible rule-based optimizers is the EXODUS optimizer generator [35, 37, 36]. The input to the optimizer generator is a model description file, where the database implementor lists the set of operators of the data model, the set of methods to be considered when building and comparing access plans, the

transformation rules that define legal transformations on the query trees, and the implementation rules that associate methods with operators. Each transformation rule is associated with an expected cost factor, which is derived automatically by the optimizer, by learning from its past experience. The optimizing process is a hill climbing method guided by these cost factors. We believe that assigning cost values to rules instead of cost functions that depend on the data, is too restricted, since it does not let a rule be applied in many different situations. More recently Graefe introduced dynamic query evaluation plans [37] to solve this problem. Here most of the optimization task is done at compile time, and a guard is generated for each access plan that retrieves statistics from the database. If a guard of an access plan evaluates to true then the plan is executed, otherwise the optimizer is called again to redo optimization.

Global query optimization methods [47, 68], where a set of queries are considered together for optimization, also seem very promising. Here a set of queries that use the same relations is analyzed to detect possible redundant computations, such as sorting a relation before a merge join operation, by materializing intermediate relations in a way that can be used by more than one query.

2.6 Flexible Database Management Systems and Translators

Our work is motivated and influenced by the Genesis extensible database management system developed at the University of Texas at Austin [4, 7, 5, 6]. Genesis introduced a technology which enables customized database management systems to be developed rapidly, using user-defined modules as building blocks. The Genesis concrete layer, which forms the physical database, is a collection of internal files and links, a generalization of the network semantic model. A transformation model is used to map abstract models to concrete implementations. This is done with possibly more than one level of conceptual to internal mappings, transferring

abstract models to more implementation-oriented ones, until a primitive layer is reached. These mappings are a sequence of database definitions that are progressively more implementation-oriented. The programs that translate abstract schemas into concrete schemas, called *type transformers*, are written by the Genesis database implementor, whose role is similar to the role of an internal schema writer, using a language that has some carefully selected compile-time reflective capabilities. The database implementor is also responsible for writing the program transformers for each type transformer, called the *operation expanders*, that translate every operation on an abstract type to a sequence of operations on the concrete type. The Genesis system supports reflective operations to make the operation expanders able to decide how to translate the operations according to the structure of the abstract schema. This provides compile-time query processing and optimization even though it does not depend on the distribution of the data in the database. This model cannot handle the problem of redundant computations between queries, since optimization decisions need to be made over a set of queries instead of just one, property beyond the capabilities of the Genesis system. Genesis can support most current database management systems (or similar systems), but the Genesis researchers claim that this system can support a database system of any complexity. We agree that the Genesis framework is sufficient general to support this position, but we think that the only way to achieve this is by extending the reflective capabilities of the language to capture more complex type transformers.

The SETL project, at New York University [24, 66], has more ambitious goals than ours. There algorithms are coded without specifying any nonabstract data structures at all. New program objects are introduced, called *bases*, which are used as universal sets of program values. Programs are written using only this set theoretic algebra. All the objects appearing in a program are dynamically assigned appropriate abstract data types from among the basic data types supported by

the language. The SETL translator uses an automatic data structure selection algorithm to perform a global analysis of the way program objects are used and related to each other to select among alternative data structures. Relations and advanced index mechanisms are provided to implement sets efficiently. The SAIL system, developed by James Low [50], is influenced by the SETL project. Here a rich library of implementations is provided to implement the various abstract types and the primitive operations. The SAIL compiler uses static flow analysis to help the implementation selection algorithm minimize the cost. Two other approaches are similar to the SETL system: PARTS by Robert Paige [60] and LIBRA by Elaine Kant [45]. All these systems have the drawback that they need a sophisticated selection algorithm to select implementations for the abstract objects. These selections depend on the type of queries performed on these objects but ignore their frequencies. None of these systems give the user the ability to change or extend the implementation library or to change the actual selection algorithm, since they do not support reflection.

CHAPTER 3

THE TYPE TRANSFORMATION MODEL

This chapter presents the type transformation model which is an extension of the work done by Darlington and others [23, 16, 46]. Their model is enhanced to include parametric types and second-order polymorphic functions, and its definition of coding functions is expanded to cover a wider class of mappings. This model is used for transforming abstract operations into expressions containing only concrete operations, in conformance with the storage structures assigned to the abstract objects manipulated by the abstract operations.

The chapter is organized as follows. Sections 3.1 and 3.2 give a brief description of the ADABTPL type system along with some generic functions for manipulating structures. Then Section 3.3 introduces the type transformation model by giving an example of implementing sets as lists. Section 3.4 presents a formal framework for the translation process. Section 3.5 uses the type transformation model for restructuring databases in accordance with schema or implementation changes. Section 3.6 uses the type transformation framework to translate the set union function. Finally, Section 3.7 explains the need for a new algebra for expressing our computations and transformations.

3.1 The ADABTPL Language

The language used in this and subsequent sections is ADABTPL [26], which is a strongly-typed functional programming language [40]. Primitive types in ADABTPL include `int`, `boolean`, and `string`. They support the usual built-in operations.

In addition to these types, ADABTPL supports type definitions, where symbols are defined to represent type expressions and type constructors for building complex type expressions. The following summarizes some of the type constructors in ADABTPL.

3.1.1 Function Types

Functions are first-class objects. They can be passed as parameters to other functions or stored as data like any other value. A function type is constructed using a pair of brackets and a right arrow. For example, `[int,int]→int` is a function type whose instances are functions that take two integers as input and return an integer as output. One instance of such a type is the integer plus function.

3.1.2 Structure and Singleton Types

A structure (or tuple) type is the cartesian product of its constituent types. One example of a tuple type definition is:

```
person = struct make_person ( name: string, ssn: int, address: string );
```

The tuple value constructor here is `make_person` of type `[string,int,string]→person`, which takes a string, an integer, and a string as input and returns a new instance of type `person` as output. The names `name`, `ssn`, and `address`, called selectors, are used to select a component of a structure of type `person`. For example, if `x` is an instance of type `person` then `x.name` or `name(x)` returns the name of `x`. Tuples with no components are defined by the singleton type constructor. For example:

```
nothing = singleton none;
```

has only one instance: the constant `none` (a nullary constructor function).

3.1.3 Union Types

A union type is defined as a set of alternatives. Every instance of a union type has the properties of only one of these alternatives. For example:

```
point = union
  ( cartesian: struct make_cartesian ( x: int, y: int );
    polar:    struct make_polar ( radius: int, angle: int ) );
```

says that a point is defined either by its cartesian coordinates x and y or by its polar coordinates $radius$ and $angle$. Every point is constructed either by the constructor `make_cartesian` of type $[int,int] \rightarrow point$ or by the constructor `make_polar` of type $[int,int] \rightarrow point$. Instances of union types are manipulated mainly by pattern-based case statements. For example, the following returns the distance of a point x from its origin:

```
case x
{ make_cartesian(m,n) → m*m+n*n;
  make_polar(r,a)   → r }
```

This expression pattern-matches the value of x with the patterns `make_cartesian(m,n)` and `make_polar(r,a)`. If x is a cartesian point then x matches the first choice and variables m and n are bound to the x and y components of x . Otherwise, variables r and a are bound to $radius$ and $angle$.

3.1.4 Parameterization

A parametric type definition introduces a new name for a set of types. It can be instantiated to a type whenever its parameters (the type variables) are instantiated to types. For example:

```
pair(alpha,beta) = struct pairup ( first: alpha, second: beta);
```


3.1.6 Finite Sets

Finite sets cannot be captured as regular recursive types because they must satisfy some special properties, such as, being independent of the order in which elements are inserted. All the type constructors described so far form a free algebra, in which there is a unique way of constructing an instance of a type. Finite sets are one exception to this rule. In Sections 4.8.3 and 4.8.4 we will encounter two more exceptions: fixed-size vectors and mutable objects. In languages that describe order-sorted algebras [19], such as OBJ2 [34], types are defined explicitly by a set of axioms, making the definition of sets and vectors possible. We will not adopt such a general approach as it will increase the complexity of our method of translating and optimizing programs.

Finite sets are expressed as instances of a special built-in type $\text{set}(\alpha)$. The set type has two constructors: emptyset that returns an empty set and $\text{insert}(e,s)$ that constructs a new set by inserting the element e into the set s (the type of insert is $[\alpha, \text{set}(\alpha)] \rightarrow \text{set}(\alpha)$). Note that $\text{insert}(a, \text{insert}(b,s)) = \text{insert}(b, \text{insert}(a,s))$ and that if e is already in s then $\text{insert}(e,s) = s$. The selector function $\text{choose}(s)$ returns an arbitrary element of its input set s , while the selector function $\text{rest}(s)$ returns a set equal to s with the chosen element removed.

3.1.7 Restriction

Each type is associated with a set of values, the instances of the type, that share common properties, such as common operations. In ADABTPL we can restrict the set of instances of a type T further by using the where type constructor:

$$T' = T \text{ where}(x) p(x);$$

This defines a new type T' whose instances are all the instances x of type T that satisfy the predicate $p(x)$, where p is an ADABTPL expression of type $[T] \rightarrow \text{boolean}$. Type T' is a subtype of type T , that is, the set of instances of type T' is a subset

of that of T . Any function that operates on T values can also operate on T' values, but the reverse is not always true: the T values passed to an T' operation must be tested at compile or run time to insure that they satisfy the predicate p . The where clause is called a **restriction** in ADABTPL. The following are examples of restrictions:

```

rang1 = int where(x) x ≥ 10 and x ≤ 20;
nset(alpha) = set(alpha) where(x) x ≠ emptyset;
slist(alpha) = struct msl ( info: list(alpha), size: int )
                where(x) x.size=length(x.info);

```

The first type definition defines `rang1` to be integer in the range between 10 and 20; the second one defines `nset` to be a non-empty set; while the last keeps the length of a list as redundant information attached to the list. Redundant information is very important to optimization as it offers alternative methods of execution to choose from that may have different costs. For example, it is cheaper to access `x.size` from the `slist` `x` to derive the length of `x` than it is to compute `length(x.info)`.

In addition to type parameters, type definitions can be parameterized by values that are used in the where clauses of these type definitions. This is called a *parameterized restriction*:

```

T'[x1 : t1, ..., xn : tn] = T where(x) p(x, x1, ..., xn);

```

This defines a new type T' whose instances are all instances x of the type T that satisfy the predicate $p(x, x_1, \dots, x_n)$. That is, this restriction is parameterized and it is instantiated whenever the parameters x_i are instantiated to values during compile time. For example:

```

bounded(alpha)[low:alpha,high:alpha,less:[alpha,alpha] →boolean] =
    alpha where(x) less(low,x) and less(x,high);

```

If $\text{bounded}(t)[l,h,p]$ is an instantiation of this type, where t is any type, l and h are constant expressions of type t , and p is a function of type $[t,t] \rightarrow \text{boolean}$, then this is equivalent to the type expression:

$t \text{ where}(x) p(l,x) \text{ and } p(x,h)$

The following are some valid type definitions that refer to the type bounded :

$\text{range}[\text{low}:\text{int},\text{high}:\text{int}] = \text{bounded}(\text{int})[\text{low},\text{high},\leq];$

$\text{range1} = \text{range}[10,20];$

Another example, that we will encounter often in this thesis is ordered lists:

$\text{ordered_list}(\alpha)[\text{before}: [\alpha, \alpha] \rightarrow \text{boolean}] =$
 $\text{list}(\alpha) \text{ where}(x) \text{ ordered}(x, \text{before});$

where $\text{ordered}(x, \text{before})$ guarantees that list x is sorted by the function before .

3.1.8 Bounded Parameterization

So far we have introduced various type constructors for building complex types from simple ones. In this section we describe a language primitive for decomposing a type into its components in a way that they can be used for building other types. This is a limited form of compile-time reflection [72] and it is related to bounded universal quantification [17].

Type definitions have the following form:

$$\text{name}(a_1, \dots, a_r)[x_1 : t_1, \dots, x_n : t_n] = T(a_1, \dots, a_r);$$

where each a_i is a name of a type parameter and each $x_i : t_i$ is a value declaration to be used for parameterizing restrictions. Both the list of type parameters and value declarations are optional. Here we extend type parameterization to contain both type parameter names and type definitions. If a_i is a type parameter then it can be

instantiated to any type; if it is a type definition it can only be instantiated to a type expression that matches the right side of the definition. Type parameters and value declarations in this type definition play the role of variables that are instantiated to type expressions and values after the matching.

For example, in the following type definition

$$\text{tt}(m(\alpha, \beta) = \text{pair}(\alpha, \beta)) = \text{pair}(\text{list}(\alpha), \beta);$$

$m(\alpha, \beta) = \text{pair}(\alpha, \beta)$ is a bounded type parameter with name m . For example, $\text{tt}(\text{pair}(\text{int}, \text{int}))$ is equivalent to $\text{pair}(\text{list}(\text{int}), \text{int})$ (because $m(\alpha, \beta)$ is bound to $\text{pair}(\text{int}, \text{int})$ and both α and β are bound to int). Any type expression that does not match this type definition, such as $\text{tt}(\text{int})$, will cause a compile-time error.

Bounded parameterization is very useful when it is combined with parameterized restrictions. For example,

$$\text{tr}(m(\alpha)[f: [\alpha] \rightarrow \text{boolean}] = \alpha \text{ where}(x) f(x))[g: [\text{boolean}] \rightarrow \text{boolean}] = \\ \alpha \text{ where}(x) g(f(x));$$

For example, $\text{tr}(\text{int where}(x) x < 10)[[x] \rightarrow \text{not}(x)]$ is equivalent to the type expression $\text{int where}(x) \text{not}(x < 10)$.

3.2 Some Generic Manipulators of Structures

In the remainder of this chapter we will use reduction functions [70] over lists and sets to express operations and transformations. This does not unduly limit expressiveness, since all primitive recursive functions over lists and sets can be coded using reduction functions [42]. In Chapter 4 we will introduce a more restricted, well-behaved, class of reductions that facilitate the transformation process.

Function `list_reduce` is a reduction over lists:

```
function(alpha,beta)
  list_reduce ( x: list(alpha), acc: [alpha,beta] $\rightarrow$ beta, base: beta ) : beta;
case x
{ nil  $\rightarrow$  base;
  cons(a,r)  $\rightarrow$  acc(a,list_reduce(r,acc,base)) };
```

This defines the polymorphic function `list_reduce` that has two type parameters: *alpha* and *beta*. It has three input parameters: *x* of type `list(alpha)`, `acc` of type `[alpha,beta] \rightarrow beta`, and `base` of type *beta*. The output type of this function is *beta*. That is, the type of `list_reduce` is `[list(alpha); [alpha,beta] \rightarrow beta,beta] \rightarrow beta`. If the value of *x* is `cons(x1,cons(x2,... ,cons(xn,nil)))` then `list_reduce(x,acc,base)` computes `acc(x1,acc(x2,... ,acc(xn,base)))`. An example use of `list_reduce` is given by the length function computed by `list_reduce(x,[?,r] \rightarrow r+1,0)`, where `[x1,... ,xn] \rightarrow exp` is a lambda abstraction (anonymous function) with variables *x*₁,... ,*x*_{*n*} and body exp (expressed as $\lambda x_1 \dots \lambda x_n.$ exp in lambda calculus) and ? is a don't-care parameter (an unnamed variable). Here the base is 0 of type `int` and therefore type *beta* is instantiated to `int`. The type of `[?,r] \rightarrow r+1` is `[alpha,int] \rightarrow int`, an instantiation of type `[alpha,beta] \rightarrow beta`.

The set reduction function is `set_reduce`, very similar to `list_reduce`¹:

```
function(alpha,beta)
  set_reduce ( s: set(alpha), acc: [alpha,beta] $\rightarrow$ beta, base: beta ) : beta;
if s=emptyset
  then base
  else acc(choose(s),set_reduce(rest(s),acc,base));
```

¹We use if-then-else statements instead of case statements to manipulate set objects because there is no unique way of matching the value of a non-empty set with the pattern `insert(a,r)`. We could let '*a*' be the choose element of the set and '*r*' the rest of the set but this could possibly cause some confusion.

For example, the set union of two sets s_1 and s_2 is $\text{union}(s_1, s_2)$, computed by $\text{set_reduce}(s_1, [a, r] \rightarrow \text{insert}(a, r), s_2)$. That is, we insert every element of s_1 into s_2 .

Set equality of two sets x and y is computed by the following:

```
set_reduce(x,
    [ex,r] → member(ex,y) and r,
    set_reduce(y,
        [ey,s] → member(ey,x) and s,
        true))
```

where $\text{member}(e, s)$ is computed by $\text{set_reduce}(s, [a, r] \rightarrow (a=e) \text{ or } r, \text{false})$.

Another example, which is a second-order function, is the generic join function (it is generic enough to have selection and projection embedded in its input functions):

```
function(alpha, beta, gamma)
    join ( x: set(alpha), y: set(beta),
        match: [alpha, beta] → boolean,
        concat: [alpha, beta] → gamma) : set(gamma);
set_reduce(x,
    [ex,r] → set_reduce(y,
        [ey,s] → if match(ex,ey)
            then insert(concat(ex,ey),s)
            else s,
        r),
    emptyset);
```

$\text{join}(x, y, \text{match}, \text{concat})$ takes every combination of elements from the sets x and y , checks whether these elements satisfy the match predicate, and if so it returns a new element by applying the concat function to form a new element of the result set. An example call to this join retrieves all employees working in the CS department:

```

join(employees,departments,
     [emp,dept]→(emp.dno=dept.dno and dept.name="CS"),
     [emp,dept]→emp)

```

3.3 Mapping Sets into Lists

In this section we present an example of implementing set objects. Suppose that we have an instance of the type `set(person)` and we want to implement it as an ordered list (a list with no duplicate elements, sorted by some total order). That is, we want to translate our abstract programs that manipulate these sets, such as the union of two sets of persons, into programs that manipulate ordered lists. Ordered lists are convenient implementations of sets because there is an isomorphism between set objects and ordered lists objects. This mapping preserves all the properties of sets, such as $\text{insert}(a, \text{insert}(b, s)) = \text{insert}(b, \text{insert}(a, s))$, because there is a unique way of mapping instances of such sets into ordered lists, independently of the way they were constructed. Ordered lists are not the only valid implementations of sets. Any mapping that preserves the set properties is valid. Isomorphisms always preserve these properties but this is not a necessary condition.

Let x be a set of persons. Before mapping x into an ordered list we need to specify the ordering. Suppose that the `ssn` and the `name` of a person are unique. Then two possible orderings are the functions $[x, y] \rightarrow x.\text{ssn} < y.\text{ssn}$ and $[x, y] \rightarrow x.\text{name} < y.\text{name}$ (see Figure 3.1a). In both cases the inverse mapping for implementing sets as ordered lists is `rep`:

$$x = \text{rep}(y) = \text{list_reduce}(y, [a, r] \rightarrow \text{insert}(a, r), \text{emptyset})$$

We call function `rep` the **representation function** that interprets ordered lists as sets. The mapping from `set(person)` to a list ordered by `ssn` is `Cssn`:

$$y = \text{Cssn}(x) = \text{set_reduce}(x, [a, r] \rightarrow \text{ordered_cons}(a, r, [x, y] \rightarrow x.\text{ssn} \leq y.\text{ssn}), \text{nil})$$

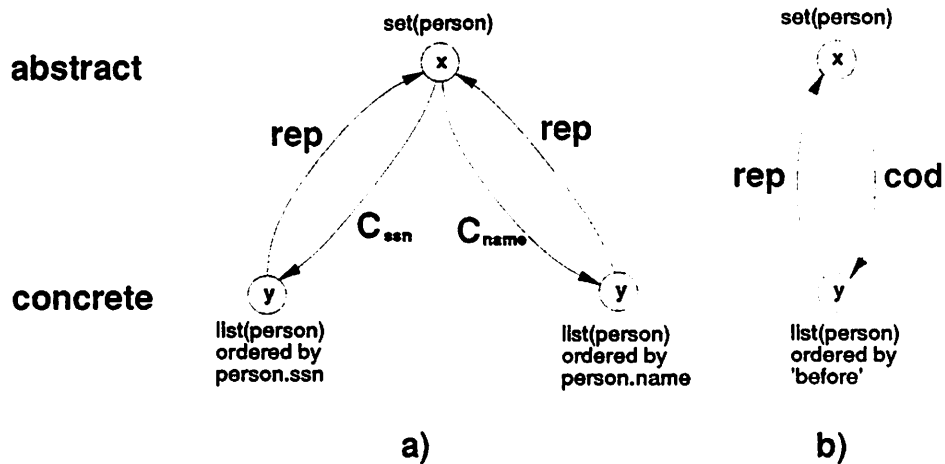


Figure 3.1 Mapping a `set(person)` into an ordered list

where `ordered_cons(e,x,before)` inserts e into the ordered list x :

```
function(alpha) ordered_cons
  ( e: alpha, x: list(alpha), before: [alpha,alpha] → boolean ) : list(alpha);
case x
{ nil → cons(e,nil);
  cons(a,r) → if before(e,a)
                then if before(a,e) then x else cons(e,x)
                else cons(a,ordered_cons(e,r,before)) }
```

while the mapping to a list ordered by name is C_{name} :

$$y = C_{name}(x) = \text{set_reduce}(x, [a,r] \rightarrow \text{ordered_cons}(a,r,[x,y] \rightarrow x.name \leq y.name), \text{nil})$$

We can capture both these isomorphisms, as well as any other similar isomorphism from `set(person)` to an ordered list, as (see Figure 3.1b):

$$y = \text{cod}(x,\text{before}) = \text{set_reduce}(x, [a,r] \rightarrow \text{ordered_cons}(a,r,\text{before}), \text{nil})$$

We call function `cod` the **coding function** of this mapping. Function `before` is an extra parameter to the coding function that determines how the resulting sequence

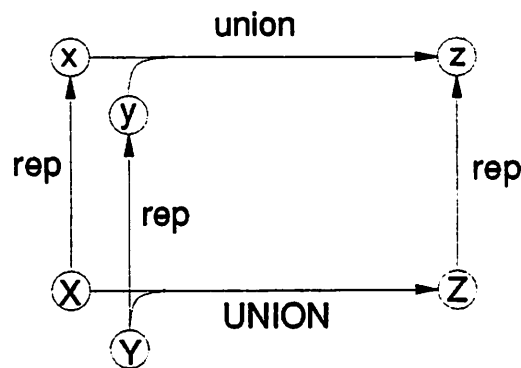


Figure 3.2 Translation of the abstract operation union

is ordered. Therefore, to map sets into ordered lists one must choose an ordering function before that will determine the implementation of a set. That way the mapping becomes isomorphic. Note that for any total ordering before we have $\text{rep}(\text{cod}(x, \text{before})) = x$.

We now examine how these mappings can help us translate set operations. Consider for example the union of sets defined as $\text{set_reduce}(x, [a, r] \rightarrow \text{insert}(a, r), y)$. Then UNION, some concrete implementation of union, must satisfy the following condition for all X and Y (from Figure 3.2):

$$\text{rep}(\text{UNION}(X, Y)) = \text{union}(\text{rep}(X), \text{rep}(Y))$$

One possible solution for UNION is:

$$\text{UNION}(X, Y) = \text{cod}(\text{union}(\text{rep}(X), \text{rep}(Y)), \text{before})$$

as can be easily proved if we substitute this expression to the previous condition and use the equation $\text{rep}(\text{cod}(x, \text{before})) = x$. The following section specifies this translation process formally.

3.4 The Formal Framework

In the type transformation framework we transform abstract operations into expressions consisting only of concrete operations, in conformance with the storage

structures assigned to the abstract objects manipulated by these abstract operations. This relationship between abstract objects and storage structures can be expressed as mapping dependencies:

Definition 3.1: (Mapping Dependency) A mapping dependency m between two types t and t' is a predicate of type $[t, t'] \rightarrow \text{boolean}$.

In the context of this mapping, t is referred to as the **abstract type** and t' as the **concrete type**.

Definition 3.2: (Implementation) A function $F : [t'_1, \dots, t'_n] \rightarrow t'_0$ is an implementation of a function $f : [t_1, \dots, t_n] \rightarrow t_0$ via the mapping dependencies $m_i : [t_i, t'_i] \rightarrow \text{boolean}$ if the following equation is true for all x_i and y_i :

$$m_1(x_1, y_1) \wedge \dots \wedge m_n(x_n, y_n) \Rightarrow m_0(f(x_1, \dots, x_n), F(y_1, \dots, y_n)) \quad (3.1)$$

In the context of this mapping, function f is referred to as the **abstract operation** while F as the **concrete operation**.

Equation 3.1 is a second-order theorem in which function F is unknown. Proving second-order theorems and constructing programs from proofs is a very difficult problem in general. In Section 5.5 we will present a complete algorithm for proving such theorems for a restricted class of functions. If in addition we are concerned with deriving not just any implementation F of f but only the optimal one (according to some cost criteria), then we may need to generate all possible programs F and choose the one with the minimal cost. The number of alternative functions F that satisfy Equation 3.1 can be substantially diminished if we restrict all mapping dependencies to be of the form:

$$m(x, y) = (y = r(x)) \wedge (x = c(y))$$

that is, m is expressed in terms of a representation function r and a coding function c . Even though we would prefer to consider as many alternatives as possible for

constructing good implementations we will adopt this restriction in order to avoid the undecidability of the general translation problem.

For purposes that will be apparent in the next chapter we will define both representation and coding functions as combinators. A combinator is a function that accepts functions as input and return a new function as output. If $g(f_1, \dots, f_n)$ is such a combinator then applying the resulting function to some values x_1, \dots, x_m is expressed as $g(f_1, \dots, f_n)(x_1, \dots, x_m)$.

In our formal notation we use lower case letters for type expressions and upper case letters for type names. Names starting with the letters α or β denote type parameters. We denote sequences of zero or more type parameters, such as $\alpha_1, \dots, \alpha_n$, as $\bar{\alpha}$ or $\bar{\beta}$.

Definition 3.3: (Representation Function) Let $t_1(\bar{\alpha})$ and $t_2(\bar{\beta})$ be two parametric types. A representation function R_{t_2, t_1} that returns a mapping of the type $t_2(\bar{\beta})$ into the type $t_1(\bar{\alpha})$ is a combinator of type $((\beta_1 \rightarrow \alpha_1) \times \dots \times (\beta_n \rightarrow \alpha_n)) \rightarrow (t_2(\bar{\beta}) \rightarrow t_1(\bar{\alpha}))$.

In the context of this mapping, t_1 is referred to as the abstract type and t_2 as the concrete type. If y is an instance of the concrete type t_2 , then there is an instance x of the abstract type t_1 such that $x = R_{t_2, t_1}(r_1, \dots, r_n)(y)$. Each r_i is a representation function for mapping β_i into α_i .

For example, the representation function that interprets lists as sets is:

```
function(alpha, beta) rep ( r1: [alpha] → beta ) : [list(beta)] → set(alpha);
[y] → list_reduce(y, [a, r] → insert(r1(a), r), emptyset);
```

that is, it returns a function that maps an instance y of type $\text{list}(beta)$ into an instance of type $\text{set}(alpha)$. The reason why we do not map $\text{list}(beta)$ into $\text{set}(beta)$ (and in general $t_2(\bar{\beta})$ into $t_1(\bar{\beta})$) is flexibility: we capture more mappings by mapping both constructors and parameters.

Isomorphic mappings are of special interest. In isomorphisms there is a one-to-one correspondence between the objects of the abstract and concrete types. In this case, there is an inverse function of the representation function, called the coding function. More specifically, for the mapping with representation function $R_{t_2,t_1}(r_1, \dots, r_n)$ the coding function is its inverse $C_{t_1,t_2}(c_1, \dots, c_n)$, where each c_i is the inverse of r_i .

We can generalize this definition of coding function to include non-isomorphic mappings by adding extra parameters e_1, \dots, e_m to C_{t_1,t_2} . More specifically:

Definition 3.4: (General Coding Function) $C_{t_1,t_2}(c_1, \dots, c_n, e_1, \dots, e_m)$ is a coding function associated with the representation function $R_{t_2,t_1}(r_1, \dots, r_n)$, if each c_i is a coding function associated with the representation function r_i and for all e_1, \dots, e_m the composition of $C_{t_1,t_2}(c_1, \dots, c_n, e_1, \dots, e_m)$ with $R_{t_2,t_1}(r_1, \dots, r_n)$ is the identity function:

$$\forall e_1 \dots \forall e_m \forall x : R_{t_2,t_1}(r_1, \dots, r_n)(C_{t_1,t_2}(c_1, \dots, c_n, e_1, \dots, e_m)(x)) = x \quad (3.2)$$

For example, the coding function for mapping sets into lists is:

`function(alpha,beta) cod`

```
( c1: [alpha] → beta, before: [beta,beta] → boolean ) : [set(alpha)] → list(beta);
[x] → set_reduce(x, [a,r] → ordered_cons(c1(a),r,before),nil);
```

where `c1` is the coding function from *alpha* to *beta* and function `before` is the extra parameter e_1 of C_{t_1,t_2} .

Each homomorphic mapping defined by a representation function and a coding function with some extra parameters can be considered as a parameterized isomorphism, that is, it becomes an isomorphism whenever these parameters are instantiated. This is very convenient, as it captures many possible implementations of an abstract type and uses a theory similar to that of isomorphisms, *making the analysis easier*.

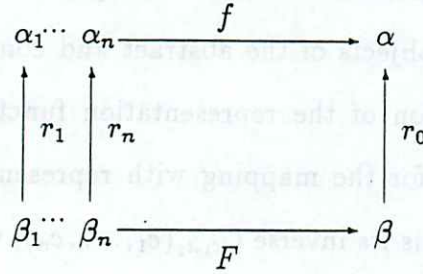


Figure 3.3 Mapping an abstract function f into a concrete operation F

We consider now the case of mapping functions into functions (Figure 3.3). Let $t_1(\alpha_1 \dots \alpha_n, \alpha) = \alpha_1 \times \dots \times \alpha_n \rightarrow \alpha$ and $t_2(\beta_1 \dots \beta_n, \beta) = \beta_1 \times \dots \times \beta_n \rightarrow \beta$. That is, function F of type $\beta_1 \times \dots \times \beta_n \rightarrow \beta$ is mapped into a function f of type $\alpha_1 \times \dots \times \alpha_n \rightarrow \alpha$, using the representation functions $r_i : \beta_i \rightarrow \alpha_i$ and $r_0 : \beta \rightarrow \alpha$. If in addition we require that all mappings are homomorphic then the above diagram commutes. That is, for all x_1, \dots, x_n we have

$$r_0(F(x_1, \dots, x_n)) = f(r_1(x_1), \dots, r_n(x_n)) \quad (3.3)$$

If c_0 is a coding function for the mapping of α into β , then one solution for F is:

$$F(x_1, \dots, x_n) = c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n))) \quad (3.4)$$

where e_1, \dots, e_m are extra parameters that need to be provided to the coding function c_0 . There are no extra parameters for isomorphic mappings.

Equation 3.4 can be used to compute F by translating the input concrete objects into abstract objects, performing the abstract operation f upon them, and translating back the resulting abstract object into a concrete one. The objective of the translation is to express the concrete implementation in terms of the lower-level primitives exclusively, avoiding the object translation overhead.

It is obvious from Equation 3.4 that isomorphic mappings produce only one function F that implements f , even though there are different programs, some more

costly than others, that are equivalent to F . In contrast, non-isomorphic mappings express function F in terms of some extra arguments to be provided before the translation, as stated in Equation 3.4. When these arguments are provided, they pin down the implementation of f . These extra parameters must not be seen as a problem but as a source of alternative methods of implementing f and, therefore, as increasing opportunities for optimization. Choosing the right values for the extra parameters can significantly improve the performance of the implementation.

Homomorphisms with no coding function are difficult to work with. Function F cannot be deduced directly from Equation 3.3, as r_0 is not a one-to-one function. For that reason, we require that all mappings be defined in terms of both a representation and a coding function (in its general form with extra parameters). A theorem prover can be used to prove whether this pair of functions satisfy Equation 3.2.

The translation process of an abstract program starts by assigning concrete types to all abstract types. This is done by specifying both the representation and the coding function for each mapping. Typically, the extra parameters of the coding function are also specified, especially when we map persistent objects, but occasionally we may leave some of these parameters unspecified, such as in the case of temporary variables where we do not want to specify a detailed mapping. The compiler delays a commitment to a decision for these extra values as long as possible, to reduce the possibility of backtracking. We may have more than one concrete type assigned to the same abstract type. This is permitted because when we are programming using a high-level language we sometimes work on the same type abstractions even when we intend to use them in different contexts. Most mappings are parametric and, therefore, can be packed into reusable modules. The compiler uses all these mappings to express the implementation F of an abstract operation f by using Equation 3.4.

3.5 Schema Evolution and Data Restructuring

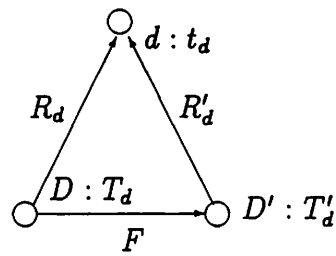
Most database application programs evolve during their lifetime to catch up with the users' needs and with current trends of technology. In our model these changes can be divided into two categories:

- Implementation evolution: changes to the implementation of the abstract model.
- Schema evolution: changes to the abstract model, including the abstract types and operations.

In either case, the actual data stored in the database must be modified to reflect the new schema. Our transformation framework can be directly applied to this situation, as it is an adequate model for mapping types to types.

When we change the implementation of the database, it will be very convenient to have a program that gets the old concrete database as input and generates the new database of the new implementation. That way the new database does not have to be rebuilt from scratch. This is the problem of data restructuring in a database environment. The compiler can synthesize such a program using the type transformation model. The benefit of this is that these programs can be translated and optimized like any other operation.

Suppose that the abstract database object d of type t_d is assigned a concrete implementation D of type T_d via the representation function R_d . If we change the implementation of the database, we need to specify a new representation function R'_d that maps the database into its new implementation. We can specify R'_d in terms of R_d by changing the representation function of a few abstract objects that are part of the database, keeping the rest the same. That way R'_d will be the same as R_d , except for these modified parts.

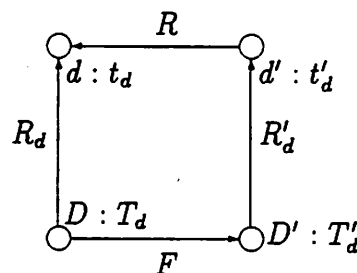


Function F is the function that rebuilds the database:

$$R'_d(F(D)) = R_d(D) \Rightarrow F(D) = C'_d(R_d(D), \dots)$$

where C'_d is a coding function associated with R'_d , possibly needing some extra parameters.

If we change the database abstract schema t_d to t'_d via the representation function R , then the new abstract database object $d' : t'_d$ will have a different concrete implementation $D' : T'_d$ via the new representation function R'_d . Modifying the abstract database schema t_d into a new schema t'_d is not an abstract to concrete mapping but it can be made to fit in the same framework. This mapping is usually expressed using the coding function C instead of the representation function R . Note also that mapping an abstract database schema usually means making a small change to one of the types that constitute the database schema, such as adding a new component to a tuple. In that case, we need to define the mapping for this type only, leaving the rest as identity mappings.



Function F , the function that rebuilds the database, satisfies:

$$R(R'_d(F(D))) = R_d(D) \Rightarrow F(D) = C'_d(C(R_d(D), \dots), \dots)$$

where C'_d and C are coding functions associated with R'_d and R .

3.6 Implementation of the Set Union

In this section we present an example of using the type transformation model for translating an abstract operation and of using an ad-hoc method for transforming the resulting translation into a program expressed in concrete primitives only.

Set union can be computed by the following program:

```
function(alpha) union ( x: set(alpha), y: set(alpha) ) : set(alpha);
set_reduce(x, [a,r] → insert(a,r),y);
```

Suppose that we implement $\text{set}(\alpha)$ as $\text{coded_set}(\beta)$ using a coding function associated with the representation function r . Then UNION, some concrete implementation of union, has the type signature:

```
function(beta) UNION ( X: coded_set(beta), Y: coded_set(beta) ) : coded_set(beta);
```

and satisfies the following instantiation of Equation 3.3:

$$r(\text{ra})(\text{UNION}(X,Y)) = \text{union}(r(\text{ra})(X), r(\text{ra})(Y))$$

where r has been substituted for r_0 , r_1 and r_2 since they are all the same and ra is the representation function for mapping β into α . For example, if we implement $\text{set}(\alpha)$ as an ordered list, then ra is $[z] \rightarrow z$ since β is equal to α . In this case UNION can be derived from Equation 3.4 after replacing c_0 with cod and r_1 and r_2 with rep , where cod and rep were introduced in Sections 3.3 and 3.4 as the coding and representation functions for implementing sets as ordered lists:

$$\text{UNION}(X,Y) = \text{cod}([z] \rightarrow z, \text{before})(\text{union}(\text{rep}([z] \rightarrow z)(X), \text{rep}([z] \rightarrow z)(Y)))$$

Even though this solution has the correct functionality it does not fulfill our objective of translating union into expressions of concrete primitives only. In addition, we would like to avoid the object translation overhead of mapping the input and output of union before and after the execution of the function. One way to do this is to use standard program transformation techniques to simplify the above solution. One such technique is the unfold-simplify-fold method [16]. In Chapter 4 we will present a complete method of solving this problem.

Let us first compose $F1(X,Y) = \text{union}(\text{rep}([z] \rightarrow z)(X), \text{rep}([z] \rightarrow z)(Y))$. We unfold the definitions of union and rep:

$$F1(X,Y) = \text{set_reduce}(\text{list_reduce}(X, [a,r] \rightarrow \text{insert}(a,r), \text{emptyset}), \\ [a,r] \rightarrow \text{insert}(a,r), \\ \text{list_reduce}(Y, [a,r] \rightarrow \text{insert}(a,r), \text{emptyset}))$$

unfold the first list_reduce once:

```
set_reduce(case X
  { nil → emptyset;
    cons(a,r) → insert(a, list_reduce(r, [a,r] → insert(a,r), emptyset)) },
  [a,r] → insert(a,r),
  list_reduce(Y, [a,r] → insert(a,r), emptyset))
```

pull the case out of set_reduce:

```
case X
{ nil → set_reduce(emptyset,
  [a,r] → insert(a,r),
  list_reduce(Y, [a,r] → insert(a,r), emptyset));
  cons(a,r) → set_reduce(insert(a, list_reduce(r, [a,r] → insert(a,r), emptyset)),
  [a,r] → insert(a,r),
  list_reduce(Y, [a,r] → insert(a,r), emptyset)) }
```

but the nil case can be simplified into $\text{list_reduce}(Y, [a,r] \rightarrow \text{insert}(a,r), \text{emptyset})$. Also $\text{set_reduce}(\text{insert}(a,S), [a,r] \rightarrow \text{insert}(a,r), B) = \text{insert}(a, \text{set_reduce}(S, [a,r] \rightarrow \text{insert}(a,r), B))$. This statement is not obvious. It can be proved by induction. In Section 4.8.2 we will see that a sufficient condition for $\text{set_reduce}(\text{insert}(a,S), \text{acc}, B)$ to be equal to $\text{acc}(a, \text{set_reduce}(S, \text{acc}, B))$ is acc being a commutative-idempotent monoid, that is, acc behaving like insert . The previous expression now becomes:

```

case X
{ nil → list_reduce(Y, [a,r] → insert(a,r), emptyset);
  cons(a,r) → insert(a, set_reduce(list_reduce(r, [a,r] → insert(a,r), emptyset),
                                   [a,r] → insert(a,r),
                                   list_reduce(Y, [a,r] → insert(a,r), emptyset))) }

```

But the set_reduce call can be folded into $F1(r, Y)$:

```

case X
{ nil → list_reduce(Y, [a,r] → insert(a,r), emptyset);
  cons(a,r) → insert(a, F1(r, Y)) }

```

or equivalently:

```

F1(X, Y) =
  list_reduce(X, [a,r] → insert(a,r), list_reduce(Y, [a,r] → insert(a,r), emptyset))

```

We now compose $\text{UNION}(X, Y) = \text{cod}([z] \rightarrow z, \text{before})(F1(X, Y))$:

```

set_reduce(list_reduce(X, [a,r] → insert(a,r),
                      list_reduce(Y, [a,r] → insert(a,r), emptyset)),
           [a,r] → ordered_cons(a,r,before), nil)

```

we unfold list_reduce once:

```

set_reduce(case X
  { nil → list_reduce(Y, [a,r] → insert(a,r),emptyset);
    cons(a,r) → insert(a,F1(r,Y)) },
  [a,r] → ordered_cons(a,r,before), nil)

```

and pull the case statement out:

```

case X
{ nil → set_reduce(list_reduce(Y, [a,r] → insert(a,r),emptyset),
  [a,r] → ordered_cons(a,r,before), nil);
  cons(a,r) → set_reduce(insert(a,F1(r,Y)),
  [a,r] → ordered_cons(a,r,before), nil) }

```

The nil case can be simplified using the unfold-simplify-fold method into:

```
list_reduce(Y, [a,r] → ordered_cons(a,r,before),nil)
```

and the set_reduce(insert... pulls out the ordered_cons:

```

case X
{ nil → list_reduce(Y, [a,r] → ordered_cons(a,r,before),nil),
  cons(a,r) → ordered_cons(a,set_reduce(F1(r,Y),
  [a,r] → ordered_cons(a,r,before),
  nil),before) }

```

The inner set_reduce is equal to UNION(r,Y):

```

case X
{ nil → list_reduce(Y, [a,r] → ordered_cons(a,r,before),nil),
  cons(a,r) → ordered_cons(a,UNION(r,Y),before) }

```

This can be folded into a list_reduce:

$$\text{UNION}(X,Y) =$$

$$\text{list_reduce}(X, [a,r] \rightarrow \text{ordered_cons}(a,r,\text{before}),$$

$$\text{list_reduce}(Y, [a,r] \rightarrow \text{ordered_cons}(a,r,\text{before}), \text{nil}))$$

We can see that the last solution of UNION is expressed in terms of concrete primitives only (list_reduce and ordered_cons). Note also that if we had the restriction that the ordered list Y is ordered by the same function before, that is, if:

$$Y = \text{list_reduce}(Y, [a,r] \rightarrow \text{ordered_cons}(a,r,\text{before}), \text{nil})$$

then the previous solution can be simplified further into:

$$\text{UNION}(X,Y) = \text{list_reduce}(X, [a,r] \rightarrow \text{ordered_cons}(a,r,\text{before}), Y)$$

In Chapter 6 we will present a formal framework of using restrictions ('where' clauses) attached to types to optimize expressions.

3.7 Undecidability of the Translation Problem

For any function f in the abstract domain there is always a function F in the concrete domain that satisfies Equation 3.3. This is true because there is a one-to-one mapping between the objects in the abstract and concrete domain. But not all functions are expressible as programs in a language. There is no guarantee that for any program in the abstract domain there is a concrete program (expressed in terms of concrete primitives only) that satisfies this equation. But even if such a program always exists, there is no complete method known of synthesizing it.

In the previous example of translating the union function we succeeded by applying the unfold-simplify-fold method to achieve our goal. If programs were expressed in imperative style or data types were pointer-based, this task could be more difficult. If we take a closer look to the previous example we can see a pattern for translating expressions involving reductions only: whenever we have

nested reductions we unfold the inner reduction, pull the case out, simplify each case using the same method, and then fold back to a new reduction. This suggests that an algebra consisting of reductions exclusively could make the translation problem tractable and the whole process mechanized. This algebra is the topic of the next chapter.

CHAPTER 4

UNIFORM TRAVERSAL COMBINATORS

In Section 3.1 we saw how recursive structures, such as lists and trees, can be defined inductively in ADAPTPL and how they can be manipulated by recursive programs. Transforming such programs and proving theorems about them can be made easier by requiring that functions be expressed in stereotyped ways. One kind of stereotyping is the systematic use of higher order functions that carry out all the traversal of recursive structures. Such traversal functions can capture common patterns of recursion that occur often during programming and, therefore, minimize the explicit use of recursion, which now becomes encapsulated by these functions. The well-known map function that applies a function to each element of a list is an example of a higher order function that encapsulates a traversal. By proving theorems about such traversal functions, some properties of functions using them can be proven by shallower reasoning than would be required if the traversals were not “pre-analyzed” in isolation. In particular, the use of induction proofs can be substantially diminished.

In Section 3.2 we introduced some generic manipulations of recursive structures, called reductions. They accumulate results as they traverse a structure and can be used for more computations than are expressible using a mapping traversal. *Reductions* tailored to particular recursive types can be generated automatically by a compiler by examining the type details [70, 69]. Reductions over lists and finite sets are *expressive enough* to directly simulate all primitive recursive functions [42]. The work reported in this chapter extends these reductions to cover a larger set of

recursion patterns and is motivated by the desire to use them as an aid in program transformation, theorem proving, and program synthesis.

In this chapter we explore a broad class of traversal functions and prove their fundamental properties. We introduce a family of generic programs, called **traversal combinators** [27], that capture a large family of type-safe primitive recursive functions¹. Most functions expressed as recursive programs where only one parameter becomes smaller at each recursive call are members of this family. This restriction excludes the computation of some valid functions, such as structural equalities and ordering, because they require that their two input structures be traversed simultaneously.

Our generic programs are combinators as each takes functions as inputs and returns a new function as output (the one that performs the actual reduction). One very important contribution of our approach is the treatment of the class of traversal combinators resulting from restricting their input functions to be themselves traversal combinators. We call these functions **uniform traversal combinators**. This offers a disciplined and uniform treatment of functions. This uniformity introduces some nice properties, such as these combinators being closed under composition, that aid in type transformation and theorem proving. In order to prove equality theorems it was necessary to extend our language to include structural equality as a special primitive. Programs expressed in this algebra can be tested for functional equivalence in a systematic and complete way, based on the fact that there is a unique way of expressing a function as a traversal combinator.

¹Note that we use the term *function* in two different cases: to refer to a mapping from a set of values to a set of values and to refer to an ADAPTPL function. When we refer to computation classes, such as primitive recursive functions or polynomial time functions, then we obviously mean the first choice. Otherwise, to avoid confusion we use the first meaning only when we say that a program computes or captures a certain function or that a function is expressed as a certain program.

There are two ways of extending our algebra. One way is to relax some of the restrictions posed to the uniform traversal combinators in order to capture more computations. One important restriction is the requirement that no result of a traversal is traversed again. This basically means that the only thing that we are permitted to do with these results is to pass them as whole values to constructions or to compare them with other values for equality. Removing such a restriction has some unwanted drawbacks: the simplification algorithms, such as the one that composes combinators, are not complete any more. Another way of extending our algebra is to capture data structures that cannot be defined inductively, such as finite sets, vectors, and mutable objects with sharing and cycles. It is very undesirable to have a different algebra for any such structure because we would need different simplification rules for each case. It is more preferable to define the uniform traversal combinators for each such extension in such a way that all the theorems proved for the regular inductive structures hold also in these cases with minor modifications.

The algebra presented in this chapter can capture most polynomial time functions, even though we failed to find programs that compute some interesting functions from this class, such as transitive closure and integer exponentiation. Nevertheless, our system can express complex computations and prove complicated theorems.

This chapter is organized as follows. Section 4.1 introduces the traversal combinators and gives some examples of their use. Section 4.2 presents the promotion theorem and the uniqueness property that are used throughout this chapter for proving properties of traversal combinators. Section 4.3 defines the equality combinator which cannot be captured as a traversal combinator. Section 4.4 is the most important section of this chapter. It defines the algebra of uniform traversal combinators. Section 4.5 proves that uniform traversal combinators are closed under composition and presents some composition examples. Section 4.6 explains how the

composition algorithm can be used to optimize programs. This algorithm is a very important reduction algorithm that captures most database optimization rules, such as pushing a selection inside a join. We will use it for proving theorems and for synthesizing and optimizing programs. Section 4.7 explores the restriction posed on uniform traversal combinators that the result of traversals cannot be traversed and suggests ways to overcome it. Section 4.8 extends the algebra to cover all recursively defined types, sets, vectors, objects, and second-order functions. Finally, Section 4.9 concludes this chapter by presenting how this algebra can help us manage the intractability of the translation and verification process.

4.1 Formal Definitions

All types described in this chapter are **canonical types**. The set of canonical types is a restricted subset of all the types that can be expressed in ADABTPL. They are constructed exclusively using parameterization, recursion, the singleton type constructor, the tuple type constructor, and the union type constructor (as defined in Section 3.1). Canonical types do not include primitive types. They are defined explicitly using canonical type constructors. For purposes of explanation, in the definition of a type T we restrict recursion to be a direct reference to type T , not to any type expression $t(T)$ that depends on T . For example, we do not permit the definition of the part-subpart tree structure where a subpart consists of a list of parts (Section 3.1.5). In Section 4.8 we will remove this restriction to allow any recursively defined type.

In general, any canonical type T has a number of constructors C_1, \dots, C_n . More specifically, a tuple type has only one constructor whose input types are not recursive. Each alternative of a union type has one constructor and therefore this union type has all these constructors. We assume that any union type has at least one constructor with no recursive input types. To make our notation simpler, we

assume that each constructor C_i has the variables of type T (the recursive references) separated from the other variables: we write $C_i(\bar{x}_i, \bar{y}_i)$ to indicate that the variables $\bar{x}_i = x_1^i, \dots, x_{i_r}^i$ are of any type other than T and the variables $\bar{y}_i = y_1^i, \dots, y_{i_s}^i$ are of type T .

4.1.1 The List Traversal Combinator

Before presenting the formal definition of traversal combinators, we give one example. In Section 3.2 we defined the `list_reduce` function. Another generic function over lists, very similar to `list_reduce`, is the traversal combinator `tc_list`:

function(*alpha*, *beta*)

`tc_list (fnil: [] → beta,`

`fcons: [alpha, list(alpha), beta] → beta): [list(alpha)] → beta;`

`[x] → case x`

`{ nil → fnil();`

`cons(a, r) → fcons(a, r, tc_list(fnil, fcons)(r)) };`

Note that the body of `tc_list` is a lambda abstraction, that is, this function returns a closure. In other words, since the inputs are functions this is a combinator. For this reason, applying `tc_list` of the functions `f1` and `f2` to a list `l` is written as `tc_list(f1, f2)(l)`.

Any `list_reduce` call can be expressed as a `tc_list` call:

`list_reduce(x, acc, base) = tc_list([] → base, [a, ?, s] → acc(a, s))(x)`

For example, the list length function is computed by `tc_list([] → 0, [?, ?, i] → i + 1)`. The

list append function `append(x, y)` is computed by `tc_list([] → y, [a, ?, l] → cons(a, l))(x)`,

while the list reverse is computed by `tc_list([] → nil, [a, ?, l] → append(l, cons(a, nil)))`.

The reason for letting `fcons` access `r`, the tail of the list `x`, is that this allows us to express order. For example, `length(r)` provides the distance of element `a` from the end of the list, information which could not be directly available otherwise².

²Note that we could access `r` from the accumulated result: `tc_list([] → f1, [a, r, l] → f2(a, r, l))(x) = tc_list([] → pairup(nil, f1), [a, ?, s] → pairup(cons(a, s.first), f2(a, s.first, s.second)))(x).second`, that is,

4.1.2 The General Form of Traversal Combinators

Reductions can be generalized to cover all canonical types. We call these generalized reductions *traversal combinators*. They are combinators because they accept functions as inputs, such as `fnil` and `fcons` in `tc_list`, and return a new function as output. The ‘traversal’ part of the name is justified because the output function of the combinator traverses the hierarchical structure of its input object.

Definition 4.1: (Traversal combinator) Let T be a canonical type with constructors $C_i(\bar{x}_i, \bar{y}_i)$. A traversal combinator $\mathcal{H}_T(f_1, \dots, f_n) : T \rightarrow b$, where b is any type, is defined as follows:

$$\begin{aligned}
 [x] \rightarrow & \text{case } x \\
 & \{ \dots \\
 & \quad C_i(\bar{x}_i, \bar{y}_i) \rightarrow f_i(\bar{x}_i, \bar{y}_i, \mathcal{H}_T(f_1, \dots, f_n)(y_1^i), \dots, \mathcal{H}_T(f_1, \dots, f_n)(y_r^i)); \\
 & \quad \dots \\
 & \}
 \end{aligned}$$

Variables z_k^i in $f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i)$, where $z_k^i = \mathcal{H}_T(f_1, \dots, f_n)(y_k^i)$, are called **accumulative result variables** because they accumulate the results of the recursive calls to $\mathcal{H}_T(f_1, \dots, f_n)$, while variables from \bar{x}_i and \bar{y}_i are called **non-accumulative result variables**. Note that if $C_i(\bar{x}_i, \bar{y}_i)$ has k variables of type other than T (these are the \bar{x}_i variables) and m variables of type T (these are the \bar{y}_i variables), then f_i has $k+2m$ variables: $k+m$ non-accumulative and m accumulative result variables. For example, in the expression `tc_list([], y, [a, r, s] → cons(a, s))(x)` variable `s` is accumulative while `a`, `r`, `x`, and `y` are not.

the result is the pair of the rest of the list `r` and the result of the computation. In Section 4.4 we will exclude any such use of the accumulative result variables.

From Definition 4.1 we can see that if f is the traversal combinator $\mathcal{H}_T(f_1, \dots, f_n)$ then:

$$f(C_i(\bar{x}_i, \bar{y}_i)) = f_i(\bar{x}_i, \bar{y}_i, f(y_1^i), \dots, f(y_{i_r}^i))$$

In addition, if $\forall i : f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i) = C_i(\bar{x}_i, \bar{z}_i)$ then $f = \lambda x.x$; that is, f is the identity function for T .

In ADABTPL, a traversal combinator \mathcal{H}_T is written as `tc.T`. For example, `tc.list` is the traversal combinator for the list type.

4.1.3 The Integer Traversal Combinator

The integer type `int` has two constructors: `zero: int` and `succ: [int] → int`. The traversal combinator over integers is `tc_int(fz, fs)`:

```
function(beta) tc_int ( fz: [] → beta, fs: [int, beta] → beta ): [int] → beta;
[x] → case x
  { zero → fz();
    succ(i) → fs(i, tc_int(fz, fs)(i)) };
```

If type *beta* is `int` then this combinator can simulate all primitive recursive functions for integers [64]. For example, `tc_int([] → y, [?, r] → succ(r))(x)` computes $x + y$;
`tc_int([] → zero, [?, r] → r + y)(x)` computes $x * y$;
`tc_int([] → succ(zero), [?, r] → x * r)(y)` computes x^y ;
`tc_int([] → succ(zero), [i, r] → succ(i) * r)(x)` computes $x!$;
`tc_int([] → true, [?, r] → (r = false))(x)` computes the predicate *even*(x);
`tc_int([] → false, [i, r] → (i = y) or r)(x)` computes $x > y$;
`tc_int([] → x = zero, [i, r] → (succ(i) = x) or r)(y)` computes $x \leq y$;
`tc_int([] → zero, [i, r] → succ(if (i = y) then zero else r))(x)` computes $x - y$;
and `tc_int([] → succ, [?, f] → tc_int([] → f(succ(zero)), [?, i] → f(i)))(m)(n)` computes the Ackermann function $\text{Ack}(m, n)$ [13] (type parameter *beta* in both `tc_int` is bound to `[int] → int`, yielding high order traversals).

4.1.4 The Boolean Traversal Combinator

The boolean type is just the union of the singletons true and false. The boolean traversal combinator `tc_boolean` is the thinly disguised if-then-else function:

```
function(beta) tc_boolean ( ftrue: [] → beta, ffalse: [] → beta ) : [boolean] → beta;
[x] → case x { true → ftrue(); false → ffalse() };
```

For example, `x or y = tc_boolean([] → true, [] → y)(x)`. Sometimes we will write `if x then f else g` as an alternative expression for `tc_boolean([] → f, [] → g)(x)`.

4.1.5 The Tuple Traversal Combinator

For a tuple type definition, such as `person`, we have:

```
function(beta) tc_person ( f: [string,int,string] → beta ) : [person] → beta;
[x] → case x { make_person(name,ssn,address) → f(name,ssn,address) };
```

which can also be expressed as `[x] → f(x.name,x.ssn,x.address)`.

For example: `x.name = tc_person([n,?,?] → n)(x)`.

4.2 Properties of Traversal Combinators

A theorem very useful for proving properties about traversal combinators is the promotion theorem [51] (or fusion law [56]). It states a sufficient condition for the composition of a function with a traversal combinator to be a traversal combinator. It says that the composition of a function g with a traversal combinator $\mathcal{H}_T(f_1, \dots, f_n)$ is also a traversal combinator if the composition of g with each f_i promotes the g call only to the accumulative result variables of f_i . This theorem is used in Section 4.4 as a reduction rule for composing traversal combinators.

Theorem 4.1: (Promotion Theorem)

if $\forall i \forall \bar{x}_i \forall \bar{y}_i \forall \bar{z}_i : g(f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i)) = \phi_i(\bar{x}_i, \bar{y}_i, g(z_1^i), \dots, g(z_{i,r}^i))$ then

$$g \circ \mathcal{H}_T(f_1, \dots, f_n) = \mathcal{H}_T(\phi_1, \dots, \phi_n)$$

Proof by structural induction: We will prove that $\forall x : g(f(x)) = \phi(x)$, where $f = \mathcal{H}_T(f_1, \dots, f_n)$ and $\phi = \mathcal{H}_T(\phi_1, \dots, \phi_n)$. If x is a construction $C_i(\bar{x}_i)$ (that is, C_i has no arguments of type T), then $g(f(C_i(\bar{x}_i))) = g(f_i(\bar{x}_i)) = \phi_i(\bar{x}_i) = \phi(C_i(\bar{x}_i))$. Let $x = C_i(\bar{x}_i, \bar{y}_i)$. We assume the theorem is true for all $y : T$ that are subtrees of the tree x . Then $g(f(x)) = g(f_i(\bar{x}_i, \bar{y}_i, f(y_1^i), \dots, f(y_{i_r}^i))) = \phi_i(\bar{x}_i, \bar{y}_i, g(f(y_1^i)), \dots, g(f(y_{i_r}^i)))$. But each y_k^i is a subtree of x and thus $g(f(y_k^i)) = \phi(y_k^i)$. Therefore, $\phi_i(\bar{x}_i, \bar{y}_i, g(f(y_1^i)), \dots, g(f(y_{i_r}^i))) = \phi_i(\bar{x}_i, \bar{y}_i, \phi(y_1^i), \dots, \phi(y_{i_r}^i)) = \phi(C_i(\bar{x}_i, \bar{y}_i)) = \phi(x)$. \square

For example, the promotion theorem for lists is:

$$\left. \begin{array}{l} g(f_1()) = \phi_1() \\ \forall a \forall l \forall s : g(f_2(a, l, s)) = \phi_2(a, l, g(s)) \end{array} \right\} \Rightarrow g \circ \text{tc_list}(f_1, f_2) = \text{tc_list}(\phi_1, \phi_2)$$

The promotion theorem for integers is:

$$\left. \begin{array}{l} g(f_1()) = \phi_1() \\ \forall i \forall r : g(f_2(i, r)) = \phi_2(i, g(r)) \end{array} \right\} \Rightarrow g \circ \text{tc_int}(f_1, f_2) = \text{tc_int}(\phi_1, \phi_2)$$

Another way of seeing the promotion theorem is the following. The composition $g \circ \mathcal{H}_T(f_1, \dots, f_n)$ is another traversal combinator $\mathcal{H}_T(\phi_1, \dots, \phi_n)$, where each ϕ_i satisfies the equation:

$$\phi_i(\bar{x}_i, \bar{y}_i, g(z_1^i), \dots, g(z_{i_r}^i)) = g(f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i))$$

This equation does not yield a solution for ϕ_i directly. All the terms $g(z_j^i)$ in ϕ_i must be eliminated, that is, if we generalize $g(z_j^i)$ as w_j^i in both sides of equation, where w_j^i is a new variable name, then the right side will not contain any reference to the variable z_j^i . Note that if f_i is another traversal combinator then we can apply the promotion theorem again. In Section 4.4 we will restrict every f_i in a traversal combinator $\mathcal{H}_T(\phi_1, \dots, \phi_n)$ to be another traversal combinator or some other very simple primitive. In that case we apply the promotion theorem repeatedly until we reach a simple primitive. Then eliminating $g(z_j^i)$ from both sides of the equation is

simplier. That way we get a complete algorithm for composing any function with a traversal combinator.

The following corollary says that there is a unique way of expressing a function as a traversal combinator [56]. It is used in Section 5.1 for testing the functional equality of two traversal combinators:

Corollary 4.1: (Uniqueness Property)

$$\forall i \forall \bar{x}_i \forall \bar{y}_i : g(C_i(\bar{x}_i, \bar{y}_i)) = \phi_i(\bar{x}_i, \bar{y}_i, g(y_1^i), \dots, g(y_{i,r}^i)) \Leftrightarrow g = \mathcal{H}_T(\phi_1, \dots, \phi_n)$$

Proof: \Rightarrow : From the promotion theorem with $f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i) = C_i(\bar{x}_i, \bar{z}_i) = f_i(\bar{x}_i, \bar{z}_i, \bar{z}_i)$.

\Leftarrow : From Definition 4.1. \square

Appendix B explains the definition of traversal combinators and proves the promotion theorem using basic category theory.

4.3 Structural Equality

Structural equalities for all but the trivial canonical types cannot be captured as traversal combinators. We need to define a special combinator instead. In the following definition of structural equality we do not separate the arguments of C_i of type T from the others in order to make the notation more readable:

Definition 4.2: (Structural Equality) Let $T(\bar{\alpha})$ be a canonical type, where $\bar{\alpha}$ is a sequence of type parameters $\alpha_1, \dots, \alpha_r$, $r \geq 0$. A structural equality $\mathcal{EQ}_T(\varepsilon_1, \dots, \varepsilon_r) : T(\bar{\alpha}) \times T(\bar{\alpha}) \rightarrow \text{boolean}$, where $\varepsilon_i : \alpha_i \times \alpha_i \rightarrow \text{boolean}$, over the canonical type $T(\bar{\alpha})$ is defined as:

$[x, y] \rightarrow \text{case } x, y$

{ ...

$$C_i(x_1, \dots, x_{k_i}), C_i(y_1, \dots, y_{k_i}) \rightarrow \bigwedge_{j=1}^{k_i} \mathcal{R}_{i,j}(x_j, y_j);$$

...


```

    other → false
  }

```

where each x_j and y_j are of type $T_{i,j}(\bar{\alpha})$ and

$$\mathcal{R}_{i,j} = \begin{cases} \varepsilon_s & \text{if } T_{i,j}(\bar{\alpha}) = \alpha_s \text{ for some } s \\ \mathcal{EQ}_{T_{i,j}}(\varepsilon_1, \dots, \varepsilon_r) & \text{otherwise} \end{cases}$$

In ADABTPL, a structural equality \mathcal{EQ}_T is written as `equal_T`. For example, the list equality has only one parameter `ea` that corresponds to the type parameter *alpha*:

function(*alpha*) `equal_list`

```

  ( ea: [alpha, alpha] → boolean ) : [list(alpha), list(alpha)] → boolean;

```

```

[x,y] → case x, y

```

```

  { nil, nil → true;

```

```

    cons(a,l), cons(b,r) → ea(a,b) and equal_list(ea)(l,r);

```

```

    other → false };

```

We will now enhance the definition of structural equality \mathcal{EQ}_T in such a way that the composition algorithm in the next section is sound. This is done by taking the two inputs of type T of the output function of \mathcal{EQ}_T , such as the two lists of `equal_list`, as nullary functions and by adding a continuation that maps the result of the equality to a type β (typically this continuation is expressed as a `tc_boolean` combinator, that is, as an if-then-else). The reason behind this enhancement is that when we compose a traversal combinator with an equality combinator we want to yield another equality combinator so that our language is closed under composition. This is achieved by the extra continuation ϕ .

Definition 4.3: (Equality Combinator) Let T be a canonical type with structural equality \mathcal{EQ}_T , f and h functions of type $() \rightarrow T$, and ϕ a function of type `boolean` $\rightarrow \beta$. The equality combinator for T is $\mathcal{E}_T : () \rightarrow \beta$, defined as:

$$\mathcal{E}_T(f, h, \phi)() = \phi(\mathcal{EQ}_T(\varepsilon_1, \dots, \varepsilon_r)(f(), h()))$$

Parameters ε_k are instantiated to equalities whenever the type parameters of T are instantiated to types. From now on we will ignore them because they do not affect our analysis and will denote $\mathcal{EQ}_T(\varepsilon_1, \dots, \varepsilon_r)(f, h)$ as $f() == h()$. Typically, ϕ is `tc_boolean(ϕ_1, ϕ_2)`. In that case: $\mathcal{E}_T(f, h, \phi) = (\text{if } f() == h() \text{ then } \phi_1() \text{ else } \phi_2())$.

The promotion theorem for equality combinators is simply:

$$\forall g : g \circ \mathcal{E}_T(f, h, \phi) = \mathcal{E}_T(f, h, g \circ \phi)$$

In ADABTPL, an equality combinator \mathcal{E}_T is written as `eq_T`. For example:

```
function(beta) eq_list
  ( f: [] → list(int), h: [] → list(int), c: [boolean] → beta ) : [] → beta;
  [] → c(equal_list(equal_int)(f(), h()));
```

4.4 Uniform Traversal Combinators

A very interesting class of traversal combinators results from restricting all functions f_i in $\mathcal{H}_T(f_1, \dots, f_n)$ to be traversal combinators themselves. This restriction is very important because any theorem or algorithm that refers to such combinators can also work on each f_i recursively. This strict discipline of the function form offers us a more uniform treatment of functions. Functions expressed strictly as traversal combinators have a tree-like form, where each traversal combinator $\mathcal{H}_T(f_1, \dots, f_n)$ is a node and each f_i is a child. This structured view of functions is aimed at simplifying type transformation and theorem proving and facilitating program synthesis.

Definition 4.4: (Uniform Traversal combinator) A uniform traversal combinator from T to b , where T and b are canonical types and all variables z and \bar{z} are bound variables, is one of the following:

- **projection:** a lambda abstraction $\lambda \bar{z}. z$, where z is a variable of type b ;
- **construction:** an expression $\lambda \bar{z}. C(h_1(\bar{z}), \dots, h_s(\bar{z}))$ where C is a constructor of b and each h_i is a uniform traversal combinator;

- traversal**: a traversal combinator $\lambda\bar{x}. \mathcal{H}_T(f_1, \dots, f_n)(z)$ from T to b where each f_i is a uniform traversal combinator and variable z is a non-accumulative result variable;
- equality**: an equality combinator $\lambda\bar{x}. \mathcal{E}_T(f, h, \phi)()$, where f , h , and ϕ are uniform traversal combinators³.

For example, multiplication is computed by the uniform traversal combinator:

$$[x, y] \rightarrow \text{tc_int}([\] \rightarrow \text{zero}, [?, i] \rightarrow \text{tc_int}([\] \rightarrow i, [?, j] \rightarrow \text{succ}(j))(y))(x)$$

The function `reverse(x)` computed by:

$$[x] \rightarrow \text{tc_list}([\] \rightarrow \text{nil}, [a, ?, l] \rightarrow \text{tc_list}([\] \rightarrow \text{cons}(a, \text{nil}), \\ [c, ?, s] \rightarrow \text{cons}(c, s))(l))(x)$$

is not a uniform traversal combinator, because the inner `tc_list` is on `l` which is an accumulative result variable. The following is a uniform traversal combinator that returns a list of the lengths of the lists contained in `x`, a list of lists:

$$[x] \rightarrow \text{tc_list}([\] \rightarrow \text{nil}, [a, ?, r] \rightarrow \text{cons}(\text{tc_list}([\] \rightarrow \text{zero}, \\ [?, ?, i] \rightarrow \text{succ}(i))(a, r))(x)$$

Here the inner `tc_list` is over `a`, which is not an accumulative result variable. Integer subtraction `x-y` can be computed by the following uniform traversal combinator:

$$[x, y] \rightarrow \text{tc_int}([\] \rightarrow \text{zero}, \\ [i, r] \rightarrow \text{succ}(\text{eq_int}([\] \rightarrow i, [\] \rightarrow y, \\ [z] \rightarrow \text{tc_boolean}([\] \rightarrow \text{zero}, [\] \rightarrow r)(z))))(x)$$

³Typically $\mathcal{E}_T(f, h, \phi)$ is equal to $\mathcal{E}_T(f, h, \text{tc_boolean}(\phi_1, \phi_2))$ which can be expressed as `if f() == h() then ϕ_1 else ϕ_2 .`

Note that there is no explicit use of selector functions, such as the integer predecessor and the list head and tail, as are found in other languages, such as in [9]. The reason for this is found in the use of patterns in the `case` construct which effectively embeds the selector functions in the traversals as discussed above. For example, the tail of a list `x` is equivalent to the following traversal expression in our algebra:

```
tc_list([], nil, [?, r, ?] → r)(x)
```

Two points need to be clear in the definition of uniform traversal combinators. First, traversals are defined to be over variables (that is, over projections), not over any other form of uniform traversal combinators. In addition, this definition does not state that the composition of two traversals is defined to be a uniform traversal combinator. But we will prove next that this is true for any such composition. That way a traversal over another traversal is reduced to a traversal over a variable. Second, the variable `z` of a traversal must not be an accumulative result variable. This is a necessary condition for having these combinators closed under composition. The intuition behind this is that we cannot traverse the values that are accumulated during the traversal of a structure (but we can pass them as whole values to constructions or to equalities). This restriction is very important because it substantially limits the expressiveness of our algebra (it makes our language at most polynomial time [9]). In Section 4.7 we will explore more the consequences of this restriction and ways of removing it.

Even though the previous program that computes the reverse function is not in our language, there is an alternative program that has the same functionality and it is in the language:

```
tc_list([], nil,
         [?, l, r] → cons(nth(x, length(l)), r))(x)
```

where $\text{nth}(x,n)$ is computed by:

```
tc_list( [] → ?,
  [a,s,u] → if succ(length(s))+n=length(x)
              then a
              else u)(x)
```

$\text{nth}(x,n)$ returns the n th element of the list x . Here n is $\text{length}(l)$, the length of the tail of the list during the outer traversal of x . The composition $\text{succ}(\text{length}(s))+\text{length}(l)$ can be expressed as a uniform traversal combinator (as we will see in the next section) and, therefore, this computation of reverse is also a uniform traversal combinator.

Even though reverse can be captured in our algebra, we have yet to find a way to express some interesting functions, such as the transitive closure and the integer exponentiation⁴. For example, suppose that a graph is represented by a list of pairs, where each pair $\text{pairup}(\text{from},\text{to})$ indicates that there is a graph edge from node from to node to . The transitive closure of this graph is the set of all pairs $\text{pairup}(a,b)$ such that there is a path in the graph from node a to node b . The following algorithm that computes the transitive closure of the graph x works as follows: for each graph edge $\text{pairup}(a_1,a_2)$ and for each path $\text{pairup}(b_1,b_2)$ that we have already found using this algorithm we check if $a_2=b_1$. If this is true we insert the path $\text{pairup}(a_1,b_2)$. Then we check if $b_2=a_1$. If this is true we insert the path $\text{pairup}(b_1,a_2)$:

```
[x] → tc_list( [] → emptyset,
  [a,?,r] → tc_list( [] → insert(a,emptyset),
    [b,?,s] → tcext(a,b,s))(r))(x)
```

where $\text{tcext}(a,b,s)$ is:

⁴This does not necessarily mean that such programs do not exist.

```

tc_pair([a1,a2] → tc_pair([b1,b2] →
  insert(b,if a2=b1
    then if b2=a1 then insert(pairup(b1,a2),insert(pairup(a1,b2),s))
    else insert(pairup(a1,b2),s)
  else if b2=a1 then insert(pairup(b1,a2),s) else s))

```

(b))(a)

We can see that this program is not a uniform traversal combinator because the inner `tc_list` traverses `r`, an accumulative result variable.

4.5 Composition of Uniform Traversal Combinators

Suppose that we have the composition $g(h(\bar{x}))$, where g and h are uniform traversal combinators. We can synthesize the traversal ϕ equal to $g \circ h$ by applying the promotion theorem to break $g \circ h$ down into simpler cases. These cases are also made simpler by applying the promotion theorem again. The following constructive proof composes any uniform traversal combinators.

Theorem 4.2: (Composition of Uniform Traversal Combinators)

The composition of uniform traversal combinators is also a uniform traversal combinator.

Constructive proof: First we will present the algorithm for composing combinators and then we will prove that the algorithm is correct. We use $\Phi(g(h_1, \dots, h_r), \rho, \sigma)$ to denote the composition of the program g with the programs $h_1 \dots h_r$, where all g, h_1, \dots, h_r are uniform traversal combinators. Variable ρ contains bindings from combinators to variables while σ from variables to combinators. To extract this composition, the g call is promoted inside the expressions h_i by applying the promotion theorem. The promotion theorem says that for $f = \mathcal{H}_T(f_1, \dots, f_n)$ we have if $g(f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i)) = \phi_i(\bar{x}_i, \bar{y}_i, g(z_1^i), \dots, g(z_{i_r}^i))$ then $g \circ f = \mathcal{H}_T(\phi_1, \dots, \phi_n)$. From Definition 4.1 we have $z_k^i = f(y_k^i)$. If $w_k^i = g(z_k^i) = g(f(y_k^i))$ then $\phi_i(\bar{x}_i, \bar{y}_i, w_1^i, \dots, w_{i_r}^i)$

$$\begin{array}{l}
1) \quad \Phi(\lambda\bar{x}.z_i, \rho, \sigma) \longrightarrow \text{if } \sigma \vdash z_i/f(y_i) \text{ then } \lambda\bar{x}.f(y_i) \text{ else } \lambda\bar{x}.z_i \\
2) \quad \Phi(\lambda\bar{x}.g(z_i), \rho, \sigma) \longrightarrow \begin{cases} \text{if } \rho \vdash g(z_i)/w_i \text{ then } \lambda\bar{x}.w_i \\ \text{else if } \sigma \vdash z_i/f(y_i) \text{ then } \lambda\bar{x}.\Phi(g(f(y_i)), \rho, \sigma) \\ \text{else } \lambda\bar{x}.g(z_i) \end{cases} \\
3) \quad \Phi(\lambda\bar{x}.C(\dots, e_i(\bar{x}), \dots), \rho, \sigma) \longrightarrow \lambda\bar{x}.C(\dots, \Phi(e_i, \rho, \sigma), \dots) \\
4) \quad \Phi(\lambda\bar{x}.\mathcal{E}_T(f, h, \phi), \rho, \sigma) \longrightarrow \lambda\bar{x}.\mathcal{E}_T(\Phi(f, \rho, \sigma), \Phi(h, \rho, \sigma), \Phi(\phi, \rho, \sigma)) \\
5) \quad \Phi(g(\lambda\bar{x}.\mathcal{E}_T(f, h, \phi)), \rho, \sigma) \longrightarrow \lambda\bar{x}.\mathcal{E}_T(\Phi(f, \rho, \sigma), \Phi(h, \rho, \sigma), \Phi(g(\phi), \rho, \sigma)) \\
6) \quad \Phi(\lambda\bar{x}.\mathcal{H}_T(\dots, f_i, \dots)(C_i(\bar{u}, \bar{w})), \rho, \sigma) \\
\quad \quad \quad \longrightarrow \Phi(f_i(\bar{u}, \bar{w}, \dots, \Phi(\lambda\bar{x}.\mathcal{H}_T(\dots, f_i, \dots)(w_k), \rho, \sigma), \dots), \rho, \sigma) \\
7) \quad \Phi(g(\lambda\bar{x}.\mathcal{H}_T(\dots, f_i, \dots)(z)), \rho, \sigma) \\
\quad \quad \quad \longrightarrow \begin{cases} \lambda\bar{x}.\mathcal{H}_T(\dots, \Phi(g(\lambda\bar{x}_i\lambda\bar{y}_i\lambda\bar{w}_i.f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i)), \rho', \sigma'), \dots)(z) \\ \text{where } \rho' = \rho[\dots, g(z_k^i)/w_k^i, \dots] \\ \text{and } \sigma' = \sigma[\dots, z_k^i/\mathcal{H}_T(\dots, f_i, \dots)(y_k^i), \dots] \end{cases}
\end{array}$$

Figure 4.1 The composition algorithm

is the composition $g \circ f_i$, provided that all references to z_k^i are eliminated (as will be proved below). For that reason we pass a binding list ρ to Φ that contains the bindings $g(z_k^i) = w_k^i$ and a binding list σ that contains $z_k^i = f(y_k^i)$. The algorithm is the following:

Algorithm 4.1: (Composition Algorithm) The rules of this algorithm are shown in Figure 4.1. Expression $\rho[u/w]$ extends ρ with the binding from u to w , while expression $\rho \vdash u/w$ returns true if there is a binding in ρ from u to w . The last rule comes from the promotion theorem. It renames the variables of f_i from \bar{w}_i to \bar{z}_i but it binds each w_k^i in \bar{w}_i to $g(z_k^i)$. So if later a call $g(z_k^i)$ is found, it is replaced by w_k^i . For example, the last rule for `tc_list` is:

$$\Phi(g(\lambda\bar{x}.\text{tc_list}(f_1, f_2)(z)), \rho, \sigma) \longrightarrow \begin{cases} \lambda\bar{x}.\text{tc_list}(\Phi(g(f_1()), \rho, \sigma), \\ \quad \Phi([a, l, r] \rightarrow g(f_2(a, l, s))), \\ \quad \rho[g(s)/r], \\ \quad \sigma[s/\text{tc_list}(f_1, f_2)(l)])(z) \end{cases}$$

Note that in the last rule if z is used by any of f_i then this z is set to $C_i(\bar{x}_i, \bar{y}_i)$.

We can see that the reduction rules in Figure 4.1 are correct (from the promotion theorem). Even though the first parameter of Φ becomes smaller in each recursive call, the algorithm might not terminate because of the second rule: if $\sigma \vdash z_i/f(y_i)$ then we perform the composition $\Phi(g(f(y_i)), \rho, \sigma)$ which may have been encountered before, leading to infinite recursion. But we can prove that this is impossible. The last rule applies whenever g is a traversal. In that case Φ is called recursively with g as the first argument. This is true for the fourth and sixth rule too. This recursion terminates whenever we find a call $\Phi(g(h), \rho, \sigma)$ that does not match these recursive rules. Therefore, this call must match the second rule. Then, if x_i in $\lambda \bar{x}.x_i$ is one of the z_k^i in ρ then $g(x_i)$ is replaced with the variable w_k^i . If z_k^i does not appear in a call to g , then z_k^i is replaced with $f(y_k^i)$, where $f = \mathcal{H}_T(\dots, f_i, \dots)$. Therefore, $g(f(y_i))$ has not been encountered before and, thus, we do not fall into an infinite loop. The only other place (besides the second rule) that z_k^i could appear is in z in the last rule. But this is not permitted because in uniform traversal combinators traversals cannot be performed over accumulative result variables. \square

To use the synthesis of combinators in proving theorems, different expressions are reduced to their traversal form, over the same variable, and checked for identity. We now use this technique to prove that $\text{length}(x) + \text{length}(y)$ is equal to $\text{length}(\text{append}(x,y))$. First, we will find a traversal combinator $\text{tc_list}(h1,h2)(x)$ equivalent to the composition $\text{length}(\text{append}(x,y))$, where append and length are:

$$\text{append}(x,y) = \text{tc_list}([\] \rightarrow y, [a,?,s] \rightarrow \text{cons}(a,s))(x)$$

$$\text{length}(x) = \text{tc_list}([\] \rightarrow \text{zero}, [?,?,i] \rightarrow \text{succ}(i))(x)$$

We apply the promotion theorem with $g = \text{length}$:

$$1) h1() = \text{length}(y) = \text{tc_list}([\] \rightarrow \text{zero}, [?,?,i] \rightarrow \text{succ}(i))(y)$$

$$2) h2(a,?,\text{length}(s)) = \text{length}(\text{cons}(a,s)) = \text{succ}(\text{length}(s))$$

$$\Rightarrow h2(?,?,u) = \text{succ}(u) \quad \text{where } u = \text{length}(s)$$

Therefore, the composition $\text{length}(\text{append}(x,y))$ is:

$$\text{tc_list}([\] \rightarrow \text{tc_list}([\] \rightarrow \text{zero}, [?, ?, i] \rightarrow \text{succ}(i))(y), \\ [?, ?, u] \rightarrow \text{succ}(u))(x)$$

Let $\text{length}(x) + \text{length}(y)$ be equal to $\text{tc_list}(h1, h2)(x)$, where

$$x + y = \text{tc_int}([\] \rightarrow y, [?, j] \rightarrow \text{succ}(j))(x)$$

We apply the promotion theorem for lists with

$$g(x) = \text{tc_int}([\] \rightarrow \text{tc_list}([\] \rightarrow \text{zero}, [?, ?, i] \rightarrow \text{succ}(i))(y), [?, j] \rightarrow \text{succ}(j))(x)$$

to compose $g(\text{length}(x)) = \text{tc_list}(h1, h2)(x)$:

$$1) h1() = g(\text{zero}) = \text{tc_list}([\] \rightarrow \text{zero}, [?, ?, i] \rightarrow \text{succ}(i))(y)$$

$$2) h2(?, ?, g(i)) = g(\text{succ}(i)) = \text{succ}(g(i)) \\ \Rightarrow h2(?, ?, w) = \text{succ}(w) \quad \text{where } w = g(i)$$

Therefore, $\text{length}(x) + \text{length}(y)$ is:

$$\text{tc_list}([\] \rightarrow \text{tc_list}([\] \rightarrow \text{zero}, [?, ?, i] \rightarrow \text{succ}(i))(y), \\ [?, ?, w] \rightarrow \text{succ}(w))(x)$$

From these two examples we can see that

$$\text{length}(\text{append}(x,y)) = \text{length}(x) + \text{length}(y)$$

This is an example of a theorem proved without using induction explicitly. Here testing the equality of $\text{length}(\text{append}(x,y))$ and $\text{length}(x) + \text{length}(y)$ was trivial. In Section 5.1 we will present a complete method for testing functional equalities.

When we compose a function with a nested traversal we need to apply the promotion theorem multiple times: once for each traversal. The following example is more complex than the previous example and is presented exclusively for illustrating such a case. It composes $g(f(x))$, where:

$$f(x) = tc_list([\rightarrow nil, \\ [a,?,r] \rightarrow cons(tc_list([\rightarrow zero, \\ [?,?,i] \rightarrow succ(i))(a,r)))(x)$$

$$g(y) = tc_list([\rightarrow zero, \\ [b,?,j] \rightarrow tc_int([\rightarrow j, [?,k] \rightarrow succ(k))(b))(y)$$

(if x is a list of lists then $f(x)$ returns the list of lengths of x and if y is a list of integers then $g(y)$ returns the sum of all these integers). The promotion theorem for $g(f(x))$ equal to $tc_list(f1,f2)(x)$ gives:

$$\begin{aligned} 1) f1() &= g(nil) = zero \\ 2) f2(a,?,g(r)) &= g(cons(tc_list([\rightarrow zero, [?,?,i] \rightarrow succ(i))(a,r)) \\ &= tc_int([\rightarrow g(r), [?,k] \rightarrow succ(k)) \\ &\quad (tc_list([\rightarrow zero, [?,?,i] \rightarrow succ(i))(a))) \end{aligned}$$

Let $h(x) = tc_int([\rightarrow u, [?,k] \rightarrow succ(k))(x)$, where $u=g(r)$, and $h(tc_list([\rightarrow zero, [?,?,i] \rightarrow succ(i))(a)) = tc_list(h1,h2)(a)$, then:

$$\begin{aligned} 1) h1() &= u \\ 2) h2(?,?,h(i)) &= succ(h(i)) \Rightarrow h2(?,?,w) = succ(w) \end{aligned}$$

Therefore, $g(f(x))$ is

$$tc_list([\rightarrow zero, \\ [a,?,u] \rightarrow tc_list([\rightarrow u, [?,?,w] \rightarrow succ(w))(a))(x)$$

4.6 Using the Composition Algorithm for Program Optimization

Algorithm 4.1 that composes two uniform traversal combinators is a very powerful reduction algorithm that optimizes programs. This algorithm basically reduces traversals over traversals into traversals over single variables. This is done by pushing the outer traversal of the composition into the components of the inner traversal.

This does not imply that a traversal performed after another traversal becomes a nested traversal, because the outer traversal will be pushed inside the inner traversal until it is eliminated by the generalizations introduced in Algorithm 4.1. That is, the result will be a single traversal. For example, let

$$\text{sum}(x) = \text{tc_list}([\] \rightarrow 0, [a, ?, r] \rightarrow a+r)(x)$$

$$\text{upto}(n) = \text{tc_int}([\] \rightarrow \text{nil}, [i, s] \rightarrow \text{cons}(i, s))(n)$$

that is, $\text{sum}(x)$ adds all elements in the list x and $\text{upto}(n)$ creates a list with elements from zero to $n-1$. The expression $\text{sum}(\text{upto}(n))$ is simplified using the composition algorithm into:

$$\text{sum}(\text{upto}(n)) = \text{tc_int}([\] \rightarrow 0, [i, s] \rightarrow i+s)(n)$$

The composition algorithm facilitates program optimization in three major ways:

- complete automation of the unfold-simplify-fold method [20]. ‘Eureka’ steps are not necessary here during folding to a function call as they are hidden in the generalization part of the composition algorithm (when new variable names are assigned to the accumulation results).
- complete automation of deforestation [80]. Deforestation is the removal of unnecessary intermediate data structures produced during nested function calls. For example, upto in $\text{sum}(\text{upto}(n))$ creates a new list of numbers which is not really necessary as we can see from $\text{tc_int}([\] \rightarrow 0, [i, s] \rightarrow i+s)(n)$, the simplification of $\text{sum}(\text{upto}(n))$.
- complete constant folding. Whenever we have an expression $f(C(\dots))$, where $C(\dots)$ is a construction, such as applying a traversal over a construction, it is simplified using the sixth rule of Algorithm 4.1.

In Section 4.8.2, where we define the set traversal combinator, we will see that most database optimization techniques, such as pushing a selection inside a join, are captured by the composition algorithm.

4.7 Unrestricted Uniform Traversal Combinators

In the definition of uniform traversal combinators we posed the restriction that traversals cannot traverse any accumulative result variable. This excludes programs such as the reverse computed by:

$$\text{tc_list}([\] \rightarrow \text{nil}, [a,?,l] \rightarrow \text{tc_list}([\] \rightarrow \text{cons}(a,\text{nil}), [c,?,s] \rightarrow \text{cons}(c,s))(l))$$

There are some cases where this restriction excludes many programs, such as, the traversals over trees with more than one child. For example, the program $\text{tc_tree}([\] \rightarrow 0, [?,?,l,r] \rightarrow l+r+1)$ that computes the number of nodes in a binary tree (see page 29) is not in our language.

If we allow traversals over accumulative result variables then we call these combinators **unrestricted uniform traversal combinators**. The following corollary comes from the proof of Theorem 4.2:

Corollary 4.2: The composition of an unrestricted uniform traversal combinator with a uniform traversal combinator is an unrestricted uniform traversal combinator.

The following analysis partially solves the case where we compose a uniform traversal combinator with any unrestricted uniform traversal combinator.

Let us see how we can compose $\text{length}(\text{reverse}(x))$, where

$$\text{length} = \text{tc_list}([\] \rightarrow \text{zero}, [?,?,i] \rightarrow \text{succ}(i))$$

If this is equal to $\text{tc_list}(f1,f2)$ then the promotion theorem says:

- 1) $f1() = \text{length}(\text{nil}) = \text{zero}$
- 2) $f2(a,l,\text{length}(r)) = \text{length}(\text{tc_list}([\] \rightarrow \text{cons}(a,\text{nil}), [b,?,s] \rightarrow \text{cons}(b,s))(r))$

We set the last composition equal to $\text{tc_list}(h1,h2)(r)$ and apply the promotion theorem again:

$$1) h1() = \text{length}(\text{cons}(a, \text{nil})) = \text{succ}(\text{zero})$$

$$2) h2(b, ?, \text{length}(s)) = \text{length}(\text{cons}(b, s)) = \text{succ}(\text{length}(s)) \Rightarrow h2(?, ?, j) = \text{succ}(j)$$

The equation $f2(a, l, \text{length}(r)) = \text{tc_list}([\] \rightarrow \text{succ}(\text{zero}), [?, ?, j] \rightarrow \text{succ}(j))(r)$ cannot be solved directly to yield a solution for $f2$. If we substitute $r = \text{reverse}(l)$ in the right side of the equation then Algorithm 4.1 on page 68 will fall into an infinite loop. Luckily, the right side of the equation can be transformed into:

$$\text{succ}(\text{tc_list}([\] \rightarrow \text{zero}, [?, ?, j] \rightarrow \text{succ}(j))(l)) = \text{succ}(\text{length}(l))$$

and therefore $\text{length}(\text{reverse}(x))$ is $\text{tc_list}([\] \rightarrow \text{zero}, [?, ?, u] \rightarrow \text{succ}(u))(x)$.

In general, let z be an accumulative result variable and $g(\mathcal{H}_T(f_1, \dots, f_n)(z))$ the composition of g with a traversal over z . This is reduced by Algorithm 4.1 to some $\mathcal{H}_T(\phi_1, \dots, \phi_n)(z)$. But we know that $\rho \vdash g(z)/w$. Therefore, we need to express $\mathcal{H}_T(\phi_1, \dots, \phi_n)(z)$ in terms of $g(z)$ only (so that $g(z)$ can be replaced by w). That is, we need to find a uniform traversal combinator h such that:

$$h(g(z)) = \mathcal{H}_T(\phi_1, \dots, \phi_n)(z)$$

where g and $\mathcal{H}_T(\phi_1, \dots, \phi_n)$ are known traversals. This is a second-order theorem in which h is unknown. In Section 5.5 we will present a complete way of solving such theorems and yielding uniform traversal combinators h from the proof of the theorem. Note that there may be none, one, more than one, or even an infinite number of combinators h that satisfy this theorem. Because there may be no solution for h , unrestricted uniform traversal combinators, where traversals are allowed to traverse the accumulative result variables, are not closed under composition.

Definition 4.5: (Irreducible Compositions) A form $f \circ g$, where f and g are unrestricted uniform traversal combinators, is **irreducible** if there is no unrestricted uniform traversal combinator h , derived by the previous synthesis algorithm, such as $h = f \circ g$.

Irreducible compositions are left in that form, because there are no other simple ways of improving them further. Therefore, in that case the definition of unrestricted uniform traversal combinators should contain another item to cover the case of irreducible forms.

For example, suppose that we want to compose $\text{append}(\text{reverse}(x),y)$. We apply the promotion theorem for $g(\text{reverse}(x))=\text{tc_list}(f1,f2)(x)$, where

$$g(x) = \text{append}(x,y) = \text{tc_list}([\] \rightarrow y, [a,?,r] \rightarrow \text{cons}(a,r))(x)$$

$$1) f1() = g(\text{nil}) = y$$

$$2) f2(a,?,g(r)) = g(\text{tc_list}([\] \rightarrow \text{cons}(a,\text{nil}), [c,?,s] \rightarrow \text{cons}(c,s))(r))$$

The last composition is $\text{tc_list}(h1,h2)(r)$, derived by:

$$1) h1() = g(\text{cons}(a,\text{nil})) = \text{cons}(a,g(\text{nil})) = \text{cons}(a,y)$$

$$2) h2(c,?,g(s)) = g(\text{cons}(c,s)) = \text{cons}(c,g(s))$$

Therefore, $f2(a,?,g(r))=\text{tc_list}([\] \rightarrow \text{cons}(a,y), [c,?,u] \rightarrow \text{cons}(c,u))(r)$. We want to find a uniform traversal combinator h that satisfies:

$$h(\text{tc_list}([\] \rightarrow y, [a,?,r] \rightarrow \text{cons}(a,r))(r)) = \text{tc_list}([\] \rightarrow \text{cons}(a,y), [c,?,u] \rightarrow \text{cons}(c,u))(r)$$

that is, $h(\text{append}(r,y))=\text{append}(r,\text{cons}(a,y))$. This equation cannot be solved for arbitrary r and y . Therefore, $\text{append}(\text{reverse}(x),y)$ cannot be expressed as a uniform traversal combinator using our previous analysis and, thus, it is an irreducible form.

4.8 Model Extensions

In the previous chapters we have defined an algebra that captures most polynomial time functions over any domain of canonical types. There are database applications though that require more complex types, such as sets and vectors. This section extends the definition of canonical types to capture more complex types, their traversal combinators, and enhancements to the composition algorithm to capture these cases.

4.8.1 Capturing all Recursively Defined Types

In the beginning of Section 4.1 we restricted the definition of canonical types by excluding non-direct recursions. In this section we extend the definition of uniform traversal combinators to cover all recursively defined types. We will show here that this extension does not complicate our earlier analysis. After this analysis whenever we refer to canonical types we will always mean the set of all recursively defined types.

Before introducing the general form of traversal combinators, we will define the **generic map** over any recursively defined type T [70]. This is needed for traversing a value component of a type T that refers to T in a complex way, such as a subpart of a part is a set of parts. If a type T has a constructor $C(\dots, x_i, \dots)$, where x_i is of type $t(T)$, that is a type expression that refers to T , then instead of calling \mathcal{H}_T recursively in the Definition 4.1 of \mathcal{H}_T (on page 57) we call $\mathcal{M}_{T'}(\mathcal{H}_T)$, where $T'(\alpha) = t(\alpha)$, α is a new type parameter, and $\mathcal{M}_{T'}$ is the generic map over T' . A generic map of a canonical type $T(\bar{\alpha}) = T(\alpha_1, \dots, \alpha_r)$ is the function $\mathcal{M}_T(m_1, \dots, m_r)$ that assigns a mapping m_i to each type parameter $\alpha_i \in \bar{\alpha}$:

Definition 4.6: (Generic map) A generic map $\mathcal{M}_T(m_1, \dots, m_r) : T(\bar{\alpha}) \rightarrow T(\bar{\beta})$, where $m_i : \alpha_i \rightarrow \beta_i$, over the recursively defined type $T(\bar{\alpha})$ is:

[x] \rightarrow case x

$$\left\{ \begin{array}{l} \dots \\ C_i(x_1, \dots, x_{k_i}) \rightarrow C_i(\mathcal{P}_{i,1}(x_1), \dots, \mathcal{P}_{i,k_i}(x_{k_i})); \\ \dots \end{array} \right\}$$

where each x_j (the j th parameter of C_i) is of type $T_{i,j}(\bar{\alpha}) \equiv t_{i,j}(\bar{\alpha}, T(\bar{\alpha}))$ and

$$\mathcal{P}_{i,j} = \begin{cases} m_s & \text{if } T_{i,j}(\bar{\alpha}) = \alpha_s \text{ for some } s \\ \mathcal{M}_{T_{i,j}}(m_1, \dots, m_r) & \text{if } T_{i,j}(\bar{\alpha}) \text{ contains at least one of } \bar{\alpha} \\ \lambda x.x & \text{otherwise} \end{cases}$$

In ADABTPL, a generic map \mathcal{M}_T is written as `map_T`. For example, the generic map for lists is:

```
function(alpha,beta) map_list ( f: [alpha]→beta ) : [list(alpha)]→list(beta);
[x]→case x
  { nil → nil;
    cons(a,r) → cons(f(a),map_list(f)(r)) };
```

The generic map for the type `term(alpha)`, defined as:

```
term(alpha) = union
  ( variable: struct var ( name: string ),
    base_element: struct element ( value: alpha ),
    application: struct apply ( fun: term(alpha),
                                arguments: list(term(alpha)) ),
    abstraction: struct abstract ( variables: list(string),
                                   body: term(alpha) ) );
```

is `map_term` defined as:

```
function(alpha,beta)
  map_term ( f: [alpha]→beta ) : [term(alpha)]→term(beta);
[x]→case x
  { var(v) → var(v);
    element(v) → element(f(v));
    apply(fun,args) → apply(map_term(f)(fun),
                              map_list(map_term(f))(args));
    abstract(vars,body) → abstract(vars,map_term(f)(body)) };
```

Note that the call `map_list(map_term(f))` is an instantiation of the middle case of $\mathcal{P}_{i,j}$ in the Definition 4.6. Here $T_{i,j}$ is equal to the type of argument, which is `list(term(alpha))`. Therefore, $\mathcal{M}_{T_{i,j}}$ is `map_list`. The call `map_list(map_term(f))(args)` maps function `f` to all the terms in the list `args`.

Theorem 4.3: (Composition of Generic Maps) Generic maps are closed under composition:

$$\mathcal{M}_T(m'_1, \dots, m'_r) \circ \mathcal{M}_T(m_1, \dots, m_r) = \mathcal{M}_T(m'_1 \circ m_1, \dots, m'_r \circ m_r)$$

Proof: By structural induction. \square

The following definition of general traversal combinators extends Definition 4.1 on page 57. Again we assume that each constructor $C_i(\overline{x}_i, \overline{y}_i)$ of the recursively defined type $T(\overline{\alpha})$ separates the recursive variables $y_j^i \in \overline{y}_i$ (of type $t_{i,j}(\overline{\alpha}, T(\overline{\alpha}))$) that contain references to $T(\overline{\alpha})$ from the non-recursive variables in \overline{x}_i .

Definition 4.7: (General Traversal combinator) Let $T(\overline{\alpha})$ be a recursively defined type with constructors $C_i(\overline{x}_i, \overline{y}_i)$. A general traversal combinator $\mathcal{H}_T(f_1, \dots, f_n) : T(\overline{\alpha}) \rightarrow b$ is defined as follows:

$[x] \rightarrow \text{case } x$

$$\left\{ \begin{array}{l} \dots \\ C_i(\overline{x}_i, \overline{y}_i) \rightarrow f_i(\overline{x}_i, \overline{y}_i, \mathcal{P}_{i,1}(y_1^i), \dots, \mathcal{P}_{i,i_r}(y_{i_r}^i)); \\ \dots \end{array} \right\}$$

where

$$\mathcal{P}_{i,j} = \begin{cases} \mathcal{H}_T(f_1, \dots, f_n) & \text{if } t_{i,j}(\overline{\alpha}, T(\overline{\alpha})) = T(\overline{\alpha}) \\ \mathcal{M}_{T_{i,j}}(\mathcal{H}_T(f_1, \dots, f_n), \overline{\lambda x.x}) & \text{if } T(\overline{\alpha}) \text{ is part of } t_{i,j}(\overline{\alpha}, T(\overline{\alpha})) \\ & \text{where } T_{i,j}(\beta, \overline{\alpha}) \equiv t_{i,j}(\overline{\alpha}, \beta) \end{cases}$$

where $\overline{\lambda x.x}$ are identity mappings assigned to the type parameters $\overline{\alpha}$.

For example, the general traversal combinator for the type term is:

function(*alpha*, *beta*)

tc_term (fvar: [string] \rightarrow *beta*, fel: [*alpha*] \rightarrow *beta*,

fappl: [term(*alpha*), list(term(*alpha*)), *beta*, list(*beta*)] \rightarrow *beta*,

$\text{fabs: [list(string),term(alpha),beta] \rightarrow beta } : [\text{term}(alpha)] \rightarrow beta;$
 $[x] \rightarrow \text{case } x$
 $\{ \text{var}(v) \rightarrow \text{fvar}(v);$
 $\text{element}(v) \rightarrow \text{fel}(v);$
 $\text{apply}(\text{fun}, \text{args}) \rightarrow \text{fappl}(\text{fun}, \text{args}, \text{tc_term}(\text{fvar}, \text{fel}, \text{fappl}, \text{fabs})(\text{fun}),$
 $\text{map_list}(\text{tc_term}(\text{fvar}, \text{fel}, \text{fappl}, \text{fabs}))(\text{args}));$
 $\text{abstract}(\text{vars}, \text{body}) \rightarrow \text{fabs}(\text{vars}, \text{body}, \text{tc_term}(\text{fvar}, \text{fel}, \text{fappl}, \text{fabs})(\text{body})) \};$

For example, the following call returns a function that checks whether a term refers to a variable var:

$\text{tc_term}([v] \rightarrow (v = \text{var}), [?] \rightarrow \text{false},$
 $[?, ?, \text{fun}, \text{args}] \rightarrow (\text{fun or tc_list}([?] \rightarrow \text{false}, [a, ?, r] \rightarrow a \text{ or } r)(\text{args})),$
 $[\text{vars}, ?, \text{body}] \rightarrow (\text{body or tc_list}([?] \rightarrow \text{false}, [v, ?, s] \rightarrow (v = \text{var}) \text{ or } s)(\text{vars})))$

Theorem 4.4: (General Promotion Theorem)

If $\forall i \forall x_i \forall y_i \forall z_i : g(f_i(\overline{x}_i, \overline{y}_i, \overline{z}_i)) = \phi_i(\overline{x}_i, \overline{y}_i, \mathcal{R}_{i,1}(z_1), \dots, \mathcal{R}_{i,i_r}(z_{i_r}))$, where each x_j is of type $t_{i,j}(\overline{\alpha}, T(\overline{\alpha}))$ and

$$\mathcal{R}_{i,j} = \begin{cases} g & \text{if } t_{i,j}(\overline{\alpha}, T(\overline{\alpha})) = T(\overline{\alpha}) \\ \mathcal{M}_{T_{i,j}}(g, \overline{\lambda x.x}) & \text{if } T(\overline{\alpha}) \text{ is part of } t_{i,j}(\overline{\alpha}, T(\overline{\alpha})) \\ & \text{where } T_{i,j}(\beta, \overline{\alpha}) \equiv t_{i,j}(\overline{\alpha}, \beta) \end{cases}$$

then $g \circ \mathcal{H}_T(f_1, \dots, f_n) = \mathcal{H}_T(\phi_1, \dots, \phi_n)$.

Proof: The proof is similar to that of Theorem 4.1 on page 59 (the promotion theorem) and it uses Theorem 4.3. \square

For example, the general promotion theorem for the type term(*alpha*) is:

$$\begin{array}{lll} \forall v : & g(f_1(v)) & = \phi_1(v) \\ \forall v : & g(f_2(v)) & = \phi_2(v) \\ \forall n \forall a \forall r \forall s : & g(f_3(n, a, r, s)) & = \phi_3(n, a, g(r), \text{map_list}(g)(s)) \\ \forall v \forall b \forall s : & g(f_4(v, b, s)) & = \phi_4(v, b, g(s)) \end{array}$$

$$\Rightarrow \forall x : g(\text{tc_term}(f_1, f_2, f_3, f_4)(x)) = \text{tc_term}(\phi_1, \phi_2, \phi_3, \phi_4)(x)$$

We can modify Definition 4.4 on page 63 of uniform traversal combinators to include general traversal combinators as traversals. In that case the variables of type $T_{i,j}(\beta)$ returned from the mapping $\mathcal{M}_{T_{i,j}}(\mathcal{H}_T(f_1, \dots, f_n), \overline{\lambda x.x})$ in Definition 7 are accumulative result variables but here they can be partially traversed by another traversal. That is, only the β components of $T_{i,j}(\beta)$ are not allowed to be traversed. Consider for example the query on page 79 that checks whether a term refers to a variable `var`. Variable `args` is an accumulative result variable of type `list(boolean)` (that is, $\beta = \text{boolean}$) but it can be traversed by a `tc.list`. Both `a` and `r`, though, are not allowed to be traversed. In this example `a or r` can be expressed as `if a=true then true else r` which is a uniform traversal combinator. We will prove next that under the extension of general traversal combinators the composition algorithm needs minor modifications to be valid.

Lemma 4.1: Any generic map $\mathcal{M}_T(m_1, \dots, m_r)$, where each m_i is a uniform traversal combinator, is also a uniform traversal combinator.

Proof: A generic map $\mathcal{M}_T(m_1, \dots, m_r)$ is the traversal $\mathcal{H}_T(f_1, \dots, f_n)$, where each $f_i = \lambda \overline{x_i}. \lambda \overline{y_i}. \lambda \overline{z_i}. C_i(\mathcal{R}_{i,1}(x_1^i), \dots, \mathcal{R}_{i,i_r}(x_{i_r}^i), \overline{z_i})$ (the accumulated result variables $\overline{z_i}$ are separated from the others $\overline{x_i}$ and $\overline{y_i}$) and

$$\mathcal{R}_{i,j} = \begin{cases} m_s & \text{if } T_{i,j}(\overline{\alpha}) = \alpha_s \text{ for some } s \\ \mathcal{M}_{T_{i,j}}(m_1, \dots, m_r) & \text{if } T_{i,j}(\overline{\alpha}) \text{ contains at least one of } \overline{\alpha} \\ \lambda x.x & \text{otherwise} \end{cases}$$

where each x_j^i is of type $T_{i,j}(\overline{\alpha}) \equiv t_{i,j}(\overline{\alpha}, T(\overline{\alpha}))$. It can be easily proved by induction that $\mathcal{R}_{i,j}$ is a uniform traversal combinator. Therefore f_i is a construction of uniform traversal combinators and, thus, $\mathcal{M}_T(m_1, \dots, m_r)$ is also a uniform traversal combinator. \square

For example, `map_list(m)` is `tc.list([], nil, [a,?,r] → cons(m(a),r))` and `map_term(f)` is `tc.term([v] → var(v), [v] → element(f(v)), [?,?,n,a] → apply(n,a), [v,?,s] → abstract(v,s))`.

Theorem 4.5: Uniform traversal combinators are closed under composition.

Proof: Algorithm 4.1 on page 68 is modified to capture the new uniform traversal combinators where traversals are general traversal combinators. The last rule of the algorithm comes now from the general promotion theorem. Therefore, ρ will have bindings from $\mathcal{R}_{i,k}(z_k^i)$ to w_k^i , where $\mathcal{R}_{i,j}$ is defined in Theorem 4.4. $\mathcal{R}_{i,j}$ is a uniform traversal combinator because both g and $\mathcal{M}_{T_{i,j}}(g, \overline{\lambda x.x})$ are uniform traversal combinators (from Lemma 4.1). Therefore, the analysis we performed for the proof of Theorem 4.2 is still valid. \square

4.8.2 Finite Sets

Sets are not canonical types (Section 3.1.6) but the set traversal combinator can be defined in such a way that the theorems in Section 4.2 are still true. `tc_set` is similar to `set_reduce` (introduced in Section 3.2):

```
function(alpha,beta) tc_set
  ( f1: [] → beta, f2: [alpha,set(alpha),beta] → beta ) : [set(alpha)] → beta;
[x] → if x=emptyset
  then f1()
  else f2(choose(x),rest(x),tc_set(f1,f2)(rest(x)))
```

For example, `union(x,y)` can be expressed as `tc_set([] → y, [a,?,r] → insert(a,r))(x)`.

For example, the following defines a set type with a primary key `f`, that is, there are no elements `a` and `b` in the set such that `f(a)=f(b)`:

```
keyed_set(alpha,beta)[f:[alpha] → beta] =
  set(alpha) where(x) tc_set([] → true,
    [a,r,s] → tc_set([] → s, [b,?,u] → if f(a)=f(b)
      then false
      else u)(r))(x)
```

For example, $\text{keyed_set}(\text{person}, \text{int})[\text{[x]} \rightarrow \text{x.ssn}]$ defines a set of persons whose primary key is ssn.

We can see from the definition of tc_set that $\text{tc_set}(f1, f2)(\text{insert}(a, s))$ is equal to $f2(\text{choose}(\text{insert}(a, s)), \text{rest}(\text{insert}(a, s)), \text{tc_set}(f1, f2)(\text{rest}(\text{insert}(a, s))))$ which may not be equal to $f2(a, s, \text{tc_set}(f1, f2)(s))$. We define a special class of set traversal combinators in which $\text{tc_set}(f1, f2)(\text{insert}(a, s))$ has a simple form:

Definition 4.8: (Order-independent Set Traversal Combinator) An order-independent set traversal combinator is a set traversal combinator $\text{tc_set}(f1, f2)$ that for any a and s satisfies:

$$\text{tc_set}(f1, f2)(\text{insert}(a, s)) = \text{if } a \in s \text{ then } \text{tc_set}(f1, f2)(s) \text{ else } f2(a, s, \text{tc_set}(f1, f2)(s))$$

The previous condition can be expressed in combinator form as:

$$\begin{aligned} \text{tc_set}(f1, f2)(\text{insert}(a, s)) = \\ \text{tc_set}(\text{[]} \rightarrow f2(a, s, \text{tc_set}(f1, f2)(s)), \\ \text{[b, ?, r]} \rightarrow \text{if } a = b \text{ then } \text{tc_set}(f1, f2)(s) \text{ else } r)(s) \end{aligned}$$

For example, $\text{length}(\text{insert}(a, s))$, where length is $\text{tc_set}(\text{[]} \rightarrow \text{zero}, \text{[?, ?, r]} \rightarrow \text{succ}(r))$, is equal to $\text{length}(s)$ if a is in s , or to $\text{succ}(\text{length}(s))$ otherwise.

The order-independence condition for $a \in s$ comes from the definition of tc_set . Therefore, the following theorem needs to be tested for each set traversal $f = \text{tc_set}(f1, f2)$:

$$a \notin s \Rightarrow f2(a, s, f(s)) = f2(\text{choose}(\text{insert}(a, s)), \text{rest}(\text{insert}(a, s)), f(\text{rest}(\text{insert}(a, s))))$$

This can be tested by a theorem prover, such as the one described in Chapter 5.

From now on we will require that all traversal combinators be order-independent.

A sufficient condition for the order-independence test to be true is $f2(a, s, u) = g(a, u)$, where g is a commutative-idempotent monoid [15]:

Definition 4.9: (Commutative-idempotent Monoid) A function f is a commutative-idempotent monoid if:

$$\begin{aligned} \forall m \forall n \forall s : f(m, f(n, s)) &= f(n, f(m, s)) && \text{(commutativity)} \\ \forall n \forall s : f(n, f(n, s)) &= f(n, s) && \text{(idempotence)} \end{aligned}$$

For example, `insert` and `[x,y]→ordered_cons(x,y,before)` (defined in Section 3.3) are commutative-idempotent monoids, while `cons` is not.

Theorem 4.6: If a function f_2 is a commutative-idempotent monoid then $f(\text{insert}(a,s))=f_2(a,f(s))$, where $f=\text{tc_set}(f_1, [v,?,r] \rightarrow f_2(v,r))$.

Proof: According to the definition of `tc_set` we have that $f(\text{insert}(a,s))$ is equal to $f_2(\text{choose}(\text{insert}(a,s)), f(\text{rest}(\text{insert}(a,s))))$. If $s=\text{emptyset}$ then the theorem is true. If a is a member of s then $\text{insert}(a,s)=s$ and $f_2(a,f(s))=f(s)$ (idempotence property). Otherwise, if $\text{insert}(a,s)=\{b_1, \dots, a, \dots, b_n\}$, that is, the first chosen element is b_1 , the last b_n , and a is somewhere between, then $f(\text{insert}(a,s))=f_2(b_1, f_2(\dots, f_2(a, \dots, f_2(b_n, f_1(\dots))))))$ which is equal to $f_2(a, f_2(b_1, f_2(\dots, f_2(b_n, f_1(\dots))))))$ (commutativity property), which is equal to $f_2(a, f(s))$. \square

For example, $\text{tc_set}([\] \rightarrow \text{emptyset}, [a,?,r] \rightarrow \text{insert}(a,r))(\text{insert}(u,w))$ is equal to $\text{insert}(u, \text{tc_set}([\] \rightarrow \text{emptyset}, [a,?,r] \rightarrow \text{insert}(a,r))(w))$.

The promotion theorem for sets is:

Theorem 4.7: (Promotion Theorem for Sets)

$$\left. \begin{aligned} g(f_1()) &= \phi_1() \\ \forall a \forall r \forall s : g(f_2(a, r, s)) &= \phi_2(a, r, g(s)) \end{aligned} \right\} \Rightarrow g \circ \text{tc_set}(f_1, f_2) = \text{tc_set}(\phi_1, \phi_2)$$

and if $\text{tc_set}(f_1, f_2)$ is order-independent then $\text{tc_set}(\phi_1, \phi_2)$ is order-independent.

Proof: We will prove that $\forall x : g(\text{tc_set}(f_1, f_2)(x)) = \text{tc_set}(\phi_1, \phi_2)(x)$. The proof for $x = \text{emptyset}$ is trivial. If $x \neq \text{emptyset}$ then there are a and s such that $a = \text{choose}(x)$ and $s = \text{rest}(x)$. We assume that the theorem is true for s (which is smaller than x). Then $g(\text{tc_set}(f_1, f_2)(x)) = g(f_2(a, \text{tc_set}(f_1, f_2)(s))) = \phi_2(a, g(\text{tc_set}(f_1, f_2)(s)))$

$$\begin{array}{l}
\Phi(\text{tc_set}(f_1, f_2)(\text{emptyset}), \rho, \sigma) \longrightarrow \Phi(f_1(), \rho, \sigma) \\
\Phi(\text{tc_set}(f_1, f_2)(\text{insert}(a, s)), \rho, \sigma) \\
\longrightarrow \left\{ \begin{array}{ll}
\Phi(f_2(a, s, \text{tc_set}(f_1, f_2)(s)), \rho, \sigma) & \text{if } a \notin s \\
\text{tc_set}([\] \rightarrow \Phi(f_2(a, s, \text{tc_set}(f_1, f_2)(s)), \rho, \sigma), & \text{otherwise} \\
[b, ?, r] \rightarrow \text{if } a = b \text{ then } \text{tc_set}(f_1, f_2)(s) \text{ else } r)(s) &
\end{array} \right.
\end{array}$$

Figure 4.2 Composition of set traversals

$(s)) = \phi_2(a, \text{tc_set}(\phi_1, \phi_2)(s)) = \text{tc_set}(\phi_1, \phi_2)(x)$. If $\text{tc_set}(f_1, f_2)$ is an order independent set traversal combinator and $a \in r$ then $\text{tc_set}(\phi_1, \phi_2)(\text{insert}(a, r))$ is equal to $g(\text{tc_set}(f_1, f_2)(\text{insert}(a, r))) = g(\text{tc_set}(f_1, f_2)(r)) = \text{tc_set}(\phi_1, \phi_2)(r)$. If $a \notin r$ then $\text{tc_set}(\phi_1, \phi_2)(\text{insert}(a, r)) = g(f_2(a, r, \text{tc_set}(f_1, f_2)(r))) = \phi_2(a, r, g(\text{tc_set}(f_1, f_2)(r))) = \phi_2(a, r, \text{tc_set}(\phi_1, \phi_2)(r))$. Therefore, $\text{tc_set}(\phi_1, \phi_2)(r)$ is order-independent. \square

The uniqueness property for sets is the following:

Theorem 4.8: (Uniqueness Property for Sets)

$$\left. \begin{array}{l}
g(\text{emptyset}) = \phi_1() \\
\forall a \forall s : g(\text{insert}(a, s)) = \text{if } a \in s \text{ then } g(s) \\
\hspace{1.5cm} \text{else } \phi_2(a, s, g(s))
\end{array} \right\} \Leftrightarrow g = \text{tc_set}(\phi_1, \phi_2)$$

Proof: \Leftarrow : from Definition 4.8. \Rightarrow : from Theorem 4.7 with $f_1 = \text{emptyset}$ and $f_2(a, s) = \text{insert}(a, s)$ (that is, $\text{tc_set}(f_1, f_2)$ is an order-independent combinator): $g = \text{tc_set}(\phi_1, [a, r, s] \rightarrow \text{if } a \in r \text{ then } s \text{ else } \phi_2(a, r, s)) = \text{tc_set}(\phi_1, \phi_2)$ (because $a \notin r$). \square

The last rule of Algorithm 4.1 on page 68 can incorporate the case of sets, since the set promotion theorem does not differ from the one of canonical types:

$$\Phi(g(\lambda \bar{x}. \text{tc_set}(f_1, f_2)(z)), \rho, \sigma) \longrightarrow \left\{ \begin{array}{l}
\lambda \bar{x}. \text{tc_set}(\Phi(g(f_1()), \rho, \sigma), \\
\Phi([a, l, r] \rightarrow g(f_2(a, l, s)), \\
\rho[g(s)/r], \\
\sigma[s/\text{tc_set}(f_1, f_2)(l)]))(z)
\end{array} \right.$$

But Algorithm 4.1 needs to be extended to include the two rules in Figure 4.2. The second rule says that if we know that $a \notin s$ is true for all a and s then we do not have to expand the left form into a tc_set that tests membership. We can prove that

if we allow only order-independent set traversal combinators then the composition algorithm is still true, that is, the composition of any uniform traversal combinator g with $tc_set(f_1, f_2)$ is another order-independent set traversal combinator.

For example, we set the composition $size(union(x,y))$, where

$$size(x) = tc_set([\rightarrow zero, [?, ?, i] \rightarrow succ(i)](x)$$

$$union(x,y) = tc_set([\rightarrow y, [a, ?, s] \rightarrow insert(a,s)](x)$$

equal to $tc_set(f_1, f_2)(x)$ and apply the set promotion theorem:

$$1) f_1() = size(y) \Rightarrow f_1() = tc_set([\rightarrow zero, [?, ?, i] \rightarrow succ(i)](y)$$

$$2) f_2(a,l,size(s)) = size(insert(a,s))$$

$$= tc_set([\rightarrow succ(size(s)), [b, ?, r] \rightarrow if\ a=b\ then\ size(s)\ else\ r](s)$$

$$\Rightarrow f_2(a,l,u) = tc_set([\rightarrow succ(u), [b, ?, r] \rightarrow if\ a=b\ then\ u\ else\ r](s) = g(s)$$

But $s=union(l,y)$. Let $g(union(l,y))=tc_set(h_1, h_2)(l)$. Then:

$$1) h_1() = g(y) = tc_set([\rightarrow succ(u), [b, ?, r] \rightarrow if\ a=b\ then\ u\ else\ r](y)$$

$$2) h_2(b,w,g(r)) = g(insert(b,r))$$

$$= tc_set([\rightarrow if\ a=b\ then\ u\ else\ g(r),$$

$$[c, ?, v] \rightarrow if\ b=c\ then\ g(r)\ else\ v](w)$$

Therefore, $size(union(x,y))$ is:

$$tc_set([\rightarrow tc_set([\rightarrow zero, [?, ?, i] \rightarrow succ(i)](y),$$

$$[a, l, u] \rightarrow tc_set([\rightarrow tc_set([\rightarrow succ(u), [b, ?, r] \rightarrow if\ a=b\ then\ u\ else\ r](y),$$

$$[b, w, q] \rightarrow tc_set([\rightarrow if\ a=b\ then\ u\ else\ q,$$

$$[c, ?, v] \rightarrow if\ b=c\ then\ q\ else\ v)(w))(l))(x)$$

The composition algorithm, with its extension to include sets, can be used for simplifying set operations, such as nested selections. For example, suppose that we have the following relational schema:

That is, the selection is pushed inside the join loop.

4.8.3 Vectors and Strings

Fixed-size vectors, like finite sets, cannot be captured as regular recursive types. They can be defined by expressing the properties of vector constructors explicitly. The actual implementation of vector constructors could be non-functional but it is ignored as it does not affect their properties. Vectors here are not mutable objects⁵.

Vectors here have two constructors: `newvector(l,v)` that returns a vector of size `l` whose elements are equal to `v`, and `update(i,v,a)` that returns a new vector equal to `a` but with the `i`th element set to `v` (vectors here are zero based). The only selectors for vectors are: `size(a)` that returns the size of the vector `a` and `index(i,a)` that returns the `i`th element of the vector `a`. Constructor `update` satisfies:

$$\text{update}(i,v,\text{update}(i,w,a)) = \text{update}(i,v,a)$$

The vector traversal combinator is defined as follows:

function(*alpha*,*beta*) `tc_vector`

(`f1`: `[]` \rightarrow *beta*, `f2`: `[int, alpha, beta]` \rightarrow *beta*) : `[vector(alpha)]` \rightarrow *beta*;
`[x]` \rightarrow `tc_int`(`[]` \rightarrow `f1`(`[]`), `[i,r]` \rightarrow `f2`(`i`,`index`(`i`,`x`),`r`))(`size`(`x`));

For example, the sum of all vector elements is `tc_vector`(`[]` \rightarrow 0, `[?,v,s]` \rightarrow `s+v`), the size of a vector is `tc_vector`(`[]` \rightarrow 0, `[?,?,s]` \rightarrow `s+1`), and the identity function for vectors is `[x]` \rightarrow `tc_vector`(`[]` \rightarrow `newvector`(`size`(`x`),`?`), `[i,v,s]` \rightarrow `update`(`i`,`v`,`s`))(`x`).

Theorem 4.9: (Promotion Theorem for Vectors)

$$\left. \begin{array}{l} g(f_1()) = \phi_1() \\ \forall a \forall s : g(f_2(i, v, s)) = \phi_2(i, v, g(s)) \end{array} \right\} \Rightarrow \text{gotc_vector}(f_1, f_2) = \text{tc_vector}(\phi_1, \phi_2)$$

⁵A mutable object is an object that when updated is destructively modified, so that all objects that share this object will share the new modified object instead.

$$\begin{array}{l}
\Phi(\text{tc_vector}(f_1, f_2)(\text{newvector}(l, v)), \rho, \sigma) \\
\quad \longrightarrow \Phi(\text{tc_int}([\] \rightarrow f_1(), [i, s] \rightarrow f_2(i, v, s))(l), \rho, \sigma) \\
\Phi(\text{tc_vector}(f_1, f_2)(\text{update}(k, w, a)), \rho, \sigma) \\
\quad \longrightarrow \left\{ \begin{array}{l} \Phi(\text{tc_vector}([\] \rightarrow f_1(), [i, v, s] \rightarrow \text{if } i = k \text{ then } f_2(k, w, s) \\ \text{else } f_2(i, v, s))(a), \rho, \sigma \end{array} \right.
\end{array}$$

Figure 4.3 Composition of vector traversals

Proof: $g(\text{tc_vector}(f_1, f_2)(x)) = g(\text{tc_int}([\] \rightarrow f_1(), [i, r] \rightarrow f_2(i, \text{index}(i, x), r))(\text{size}(x)))$.

We apply the promotion theorem for integers: $\phi_1() = g(f_1())$ and $\phi_2(i, \text{index}(i, x), g(r)) = g(f_2(i, \text{index}(i, x), r))$, therefore, $g(\text{tc_vector}(f_1, f_2)(x))$ is equal to $\text{tc_int}([\] \rightarrow \phi_1(), [i, r] \rightarrow \phi_2(i, \text{index}(i, x), r))(\text{size}(x)) = \text{tc_vector}(\phi_1, \phi_2)(x)$. \square

Algorithm 4.1 on page 68 is extended to include the two rules in Figure 4.3 (the promotion theorem for vectors is incorporated into the last rule of Algorithm 4.1).

The string type can be defined as $\text{vector}(\text{range}[0,255])$. For example, the concatenation of the strings x and y is computed by:

$$\begin{array}{l}
\text{tc_vector}([\] \rightarrow \text{tc_vector}([\] \rightarrow \text{newvector}(\text{size}(x) + \text{size}(y), 0), \\
\quad [i, v, a] \rightarrow \text{update}(i + \text{size}(x), v, a))(y), \\
\quad [i, v, a] \rightarrow \text{update}(i, v, a))(x)
\end{array}$$

4.8.4 Objects with Object Identity

Recursive data types can capture data structures of a tree-like form. There are applications though that require more complex data structures, including shared objects and general graphs with cycles or mutable data structures. One such example is mutable vectors. Mutable vectors though are well behaved because the only kind of vector mutation is updating a vector element. In general, objects may have cycles in an unpredictable way that usually is not prespecified in the definition of an object. In order to capture objects in a functional setting based on our constructor algebra we need to find a method similar to that for sets and vectors:

Object types here have four constructors:

- `new(v)` of type $[T] \rightarrow T'$ that creates a new object with state v and with a unique OID. For example, `new(node(v,l,r))`;
- `modify(x,v)` of type $[T',T] \rightarrow T'$ that updates the state of the object x to be v . This is a destructive operation that modifies the object state but leaves the OID unchanged. `modify(x,v)` returns x . For example, if x has type `otree` then `modify(x,node(v,l,r))` updates the state of x . The `modify` constructor satisfies `modify(modify(x,v),w) = modify(x,w)`;
- `define(x,o)` of type $[string, T'] \rightarrow T'$ defines variable x to be the object o (that is, the OID of o). The scope of x is the expression o and all right sibling expressions of o (in the expression tree that contains o). The name x in `define(x,o)` must be unique (no nested scoping is permitted);
- `refer(x)` of type $[string] \rightarrow T'$ is the object o defined by a `define(x,o)` constructor. If there is no such definition it raises a compile-time error. For example, `define("x",new(node(1,refer("x"),refer("x"))))` creates an `otree` node whose left and right parts point to the node itself. Another example is:
`new(node(1,define("x",new(node(1,new(empty),new(empty))))),refer("x"))`
 that creates a node whose left and right children are the same object.

There is only one selector for objects: `deref(x)` of type $[T'] \rightarrow T$ that returns the state of the object x .

If $\mathcal{H}_T(f_1, \dots, f_n) : T \rightarrow \beta$ is a traversal combinator for T , where $f_i : \overline{T}_i \rightarrow \beta$, then the object traversal combinator $\mathcal{H}_{T'}(g_0, g_1, \dots, g_n) : T' \rightarrow \beta$ has component functions $g_0 : T' \times \beta \rightarrow \beta$ and $g_i : T' \times \overline{T}_i \rightarrow \beta$, where the first value of each g_i is bound to the object itself. Function g_0 is executed whenever we reach an object that has already been reached before by the same traversal. It has two parameters: the

first is the object itself and the second is the β value computed for this object when it was traversed for the first time. That way the recursion terminates on shared objects or on cycles.

For example, the following program will destructively modify the state of all tree nodes with info u to have info w :

```
tc_otree([o,?] → o, [o] → o,
         [o,v,?,?,l,r] → if v=u then o else modify(o,node(w,l,r)))
```

The following creates an exact copy of an otree:

```
tc_otree([?,n] → n, [?] → new(empty),
         [?,v,?,?,l,r] → new(node(v,l,r)))
```

The following returns the list of nodes reachable from the root:

```
tc_otree([?,?] → nil, [o] → o,
         [o,?,?,?,l,r] → cons(o,append(l,r)))
```

Another example that defines a double linked list is the following:

```
double_linked_list( $\alpha$ ) =
  otree( $\alpha$ ) where(x) tc_otree([?,?] → true, [?] → true,
                               [o,?,?,r,s,?] → s and o=tc_otree([?,?] → ?, [?] → x,
                                                                    [?,?,l,?,?,?] → l)(r))(x);
```

That is, a `double_linked_list` is an otree in which the right child of any node has the node itself as the left child (the right child of the root is empty)⁶. For example, the following program inserts the value `val` at the end of a `double_linked_list`:

⁶Remember that the `=` operation for objects is not structural equality but comparison between OIDs.

```

tc_otree( [?,?] →?, [?] →new(node(val,empty,empty)),
          [o,v,l,r,s,?] →if l=empty
                        then modify(o,node(v,new(node(val,empty,o)),r))
                        else s)

```

Not all g_0 functions are valid. For example, consider the following:

```

tc_otree( [?,n] →n, [?] →0, [?,?,?,?.l,r] →l+r+1)
(define("x",new(node(1,refer("x"),refer("x")))))

```

If we assume that we have `define` and `refer` for all canonical types then this expression is equal to `define("n",refer("n")+refer("n")+1)` which is the solution of $n=2 \times n+1$. Because we do not want to permit cycles and shared objects in our regular constructor algebra we allow the second parameter of g_0 (the accumulated value) to be used only when β is an object type, that is, when the traversal \mathcal{H}_T constructs objects.

Accessing the accumulated value v in $g_0(o, v)$ that was computed when the object o was reached the first time is somewhat tricky to implement for objects with cycles. For each traversal f and for each object x we have a set $S[x, f]$ of objects. If the object o has been reached before but its accumulated value has not been computed yet (that is, if we have a cycle) then n in $g_0(o, n)$ is set equal to the address of $S[o, f]$ while the address of n is inserted into $S[o, f]$. When we finish constructing the accumulated value of the object o we set all values whose addresses are in $S[o, f]$ to the new result. This will work because the only use we have for accumulated results is for testing equalities or for passing them to constructions. If accumulated values were allowed to be traversed then this method could not work.

For example, one possible implementation for `tc_otree` in pseudo-ADABTPL is the following (if x has been encountered before and its value has been constructed (such as in shared objects) then `VALUE(x, f)` returns the already computed $f(x)$):

function(α, β) tc_otree

(fstop: [otree(α), β] \rightarrow β , fempty: [otree(α)] \rightarrow β ,

fnode: [otree(α), α , otree(α), otree(α), β , β] \rightarrow β): [otree(α)] \rightarrow β ;

[x] \rightarrow if x has already been traversed by $f = \text{tc_otree}(\text{fstop}, \text{fempty}, \text{fnode})$

then if x is in a cycle

then { $S[x, f] := \{x\} + S[x, f]$; $\text{addr}(S[x, f])$ }

else fstop($x, \text{VALUE}(x, f)$)

else let $\text{res} := \text{case } x$

{ empty \rightarrow fempty(x);

node(v, l, r) \rightarrow fnode($x, v, l, r, \text{tc_otree}(\text{fstop}, \text{fempty}, \text{fnode})(l)$,

$\text{tc_otree}(\text{fstop}, \text{fempty}, \text{fnode})(r)$) } in

{ for all y in $S[x, f]$ do $\text{deref}(y) := \text{fstop}(\text{deref}(y), \text{res})$; res };

This method is inefficient but objects are intended to be used where regular recursive types fail: for shared objects and for graphs with cycles.

The promotion theorem for objects is:

Theorem 4.10: (Object Promotion Theorem) if $\forall o \forall v : g(f_0(o, v)) = \phi_0(o, g(v))$ and $\forall o \forall i \forall \bar{x}_i \forall \bar{y}_i \forall \bar{z}_i : g(f_i(o, \bar{x}_i, \bar{y}_i, \bar{z}_i)) = \phi_i(o, \bar{x}_i, \bar{y}_i, g(z_1^i), \dots, g(z_{i,r}^i))$
then $g \circ \mathcal{H}_T(f_0, f_1, \dots, f_n) = \mathcal{H}_T(\phi_0, \phi_1, \dots, \phi_n)$

Proof: We will prove that $\forall x : g(f(x)) = \phi(x)$, where $f = \mathcal{H}_T(f_0, f_1, \dots, f_n)$ and $\phi = \mathcal{H}_T(\phi_0, \phi_1, \dots, \phi_n)$. If $\text{deref}(x) = C_i(\bar{x}_i, \bar{y}_i)$ and x has not been found before then $g(f(x)) = g(f_i(x, \bar{x}_i, \bar{y}_i, f(y_1^i), \dots, f(y_{i,r}^i))) = \phi_i(x, \bar{x}_i, \bar{y}_i, g(f(y_1^i)), \dots, g(f(y_{i,r}^i))) = \phi_i(x, \bar{x}_i, \bar{y}_i, \phi(y_1^i), \dots, \phi(y_{i,r}^i))$ (from induction step) $= \phi(x)$. If x has been reached before then $g(f(x)) = g(f_0(x, v)) = \phi_0(x, g(v))$. But v is $f(x)$ because it is the result returned when x was traversed for the first time. Thus, $g(f(x)) = \phi_0(x, g(f(x))) = \phi_0(x, \phi(x)) = \phi(x)$. \square

Algorithm 4.1 on page 68 is extended to include the rules in Figure 4.4 (the promotion theorem for objects is incorporated into the last rule of Algorithm 4.1).

$$\begin{array}{l}
\Phi(\mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(\text{define}(n, x)), \rho, \sigma, \mathcal{K}) \\
\longrightarrow \begin{cases} \text{define}(v, \Phi(\mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(x), \rho, \sigma, \mathcal{K}[n = \langle x, v \rangle])) & \text{if } \beta \text{ is object} \\ \Phi(\mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(x), \rho, \sigma, \mathcal{K}[n = \langle x, ? \rangle]) & \text{otherwise} \end{cases} \\
\Phi(\mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(\text{refer}(n)), \rho, \sigma, \mathcal{K}) \\
\longrightarrow f_0(x, \text{refer}(v)) \text{ where } \mathcal{K} \vdash n = \langle x, v \rangle \\
\Phi(\mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(\text{new}(C_i(\bar{u}, \bar{w}))), \rho, \sigma, \mathcal{K}) \\
\longrightarrow \Phi(f_i(\text{new}(C_i(\bar{u}, \bar{w}))), \bar{u}, \bar{w}, \dots, \mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(w_k), \dots), \rho, \sigma, \mathcal{K}) \\
\Phi(\mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(\text{modify}(x, C_i(\bar{u}, \bar{w}))), \rho, \sigma, \mathcal{K}) \\
\longrightarrow \Phi(f_i(\text{modify}(x, C_i(\bar{u}, \bar{w}))), \bar{u}, \bar{w}, \dots, \mathcal{H}_{T'}(f_0, f_1, \dots, f_n)(w_k), \dots), \rho, \sigma, \mathcal{K})
\end{array}$$

Figure 4.4 Composition of object traversals

Here we extended Φ to include an extra parameter \mathcal{K} that holds bindings from variable names defined by the `define` constructor to pairs of objects and accumulated results. The first rule says that $f(\text{define}(n, x))$, where $f = \mathcal{H}_{T'}(f_0, f_1, \dots, f_n)$, is $\text{define}(v, f(x))$, where v is a new variable name and \mathcal{K} is extended to include $n = \langle x, v \rangle$. If later we find $f(\text{refer}(n))$ then this is $f_0(x, \text{refer}(v))$. If the type of accumulation β is not an object type then we are not permitted to insert `define` and `refer` into the resulting construction.

For example, `copy(define("x", new(node(1, refer("x"), refer("x")))))`, where `copy` is `tc_otree([?, n] → n, [o] → new(empty), [?, v, ?, ?, l, r] → new(node(v, r, l)))`, is equal to `define("y", new(node(1, refer("y"), refer("y"))))`.

For example, the composition `nodnum(reflect(x))=tc_otree(f0, f1, f2)(x)`, where

$$\text{nodnum} = \text{tc_otree}([?, ?] \rightarrow 0, [?] \rightarrow 0, [?, ?, ?, ?, l, r] \rightarrow l+r+1)$$

$$\text{reflect} = \text{tc_otree}([?, n] \rightarrow n, [o] \rightarrow o, [o, v, ?, ?, l, r] \rightarrow \text{modify}(o, \text{node}(v, r, l)))$$

is derived using the object promotion theorem:

- 1) $f_0(o, \text{nodnum}(n)) = \text{nodnum}(n) = 0$ (note that $\text{nodnum}(n)$ could not depend on n)
 $\Rightarrow f_0(?, ?) = 0$
- 2) $f_1(o) = \text{nodnum}(\text{empty}) = 0$

$$\begin{array}{l}
\Phi(\mathcal{F}_T(c, r_1, \dots, r_n)(\lambda \bar{x}.g(\bar{x})), \rho, \sigma) \longrightarrow \Phi(c(g(r_1, \dots, r_n)), \rho, \sigma) \\
\Phi(g(\mathcal{F}_T(c, r_1, \dots, r_n)(f)), \rho, \sigma) \\
\qquad \qquad \qquad \longrightarrow \mathcal{F}_T(\Phi(g \circ c, \rho, \sigma), \Phi(r_1, \rho, \sigma), \dots, \Phi(r_n, \rho, \sigma))(f) \\
\Phi(\mathcal{F}_T(c, r_1, \dots, r_n)(f), \rho, \sigma) \longrightarrow \mathcal{F}_T(\Phi(c, \rho, \sigma), \Phi(r_1, \rho, \sigma), \dots, \Phi(r_n, \rho, \sigma))(f)
\end{array}$$

Figure 4.5 Composition of second-order traversals

$$\begin{aligned}
3) f2(o, v, ?, ?, \text{nodnum}(l), \text{nodnum}(r)) &= \text{nodnum}(\text{modify}(o, \text{node}(v, r, l))) \\
&= \text{nodnum}(r) + \text{nodnum}(l) + 1 \Rightarrow f2(?, ?, ?, ?, u, w) = w + u + 1
\end{aligned}$$

Therefore, the composition is $\text{tc_otree}([?, ?] \rightarrow 0, [?] \rightarrow 0, [?, ?, ?, ?, u, w] \rightarrow w + u + 1)$.

4.8.5 Second-order Functions

We can also extend our model to capture higher order expressions, that is, expressions where there is at least one variable that is a function:

Definition 4.10: (Second-order Combinator) Let $T = T_1 \times \dots \times T_n \rightarrow T_0$ be a function type. The uniform traversal combinator for T is $\mathcal{F}_T(c, r_1, \dots, r_n) : T \rightarrow \beta$, where $c : T_0 \rightarrow \beta$ and $r_i : () \rightarrow T_i$, and it is defined as $\lambda f. c(f(r_1(), \dots, r_n()))$. Variable f is not permitted to be an accumulative result variable.

The promotion theorem for T is simply:

$$g \circ \mathcal{F}_T(c, r_1, \dots, r_n) = \mathcal{F}_T(g \circ c, r_1, \dots, r_n)$$

The composition algorithm is still correct because if h is a uniform traversal combinator then so is $\mathcal{F}_T(c, r_1, \dots, r_n)(h)$. More specifically, Algorithm 4.1 on page 68 is extended to include the rules in Figure 4.5.

In ADABTPL, a combinator \mathcal{F}_T for any type $T = T_1 \times \dots \times T_n \rightarrow T_0$ is written as `fcn`. For example, for $n = 3$ the second-order combinator \mathcal{F}_T is `fc3`:

```

function( alpha, beta, a1, a2, a3) fc3
  ( c: [alpha] → beta, r1: [] → a1, r2: [] → a2, r3: [] → a3 )
  : [[a1, a2, a3] → alpha] → beta;
  [f] → c(f(r1(), r2(), r3()));

```

For example, `map_list` over any function `f` is expressed as:

```
map_list(f) = tc_list( [] → nil, [b, ?, r] → fc1( [z] → cons(z, r), [] → b)(f))
```

The generic join defined in Section 3.2 is computed by:

```

function( alpha, beta, gamma)
  join ( x: set(alpha), y: set(beta),
        match: [alpha, beta] → boolean,
        concat: [alpha, beta] → gamma) : set(gamma);
  tc_set( [] → emptyset,
        [ex, ?, r] → tc_set( [] → r,
                              [ey, ?, s] → fc2( [o] → tc_boolean( [] → fc2( [u] → insert(u, s),
                                                                              [] → ex, [] → ey) (concat),
                                                                              [] → s) (o),
                              [] → ex, [] → ey) (match))(y))(x)

```

4.9 Use of Uniform Traversal Combinators

Uniform traversal combinators can be used for solving the problem introduced in Section 3.7: if all operations are expressed as uniform traversal combinators then the translation of an operation `f` is (from Equation 3.4):

$$F(x_1, \dots, x_n) = c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n))).$$

Since this is a composition of uniform traversal combinators, it is also a uniform traversal combinator. Furthermore, the composition algorithm will 'flatten out' the

resulting expression into traversals and constructors from the concrete types only. This can be proved by induction⁷: if the result of the composition is a projection the statement is true; if it is a construction $C(h_1(\bar{x}), \dots, h_s(\bar{x}))$ then none of the parameters of C return an abstract object because C is a constructor of a concrete type which by definition does not refer to any abstract type; if it is a traversal $\mathcal{H}_T(f_1, \dots, f_n)$ that returns a concrete type then all f_i return concrete types.

In addition, the composition algorithm can be used for proving theorems about uniform traversal combinators, because it reduces expressions involving traversals into traversals over single variables. Then testing the functional equality of two uniform traversal combinators is easier, because there is a unique way of expressing a function as a traversal over a variable (from the uniqueness property). In the next chapter we will explore the theorem proving process further.

⁷We assume that there are no types in common between the abstract and concrete types.

CHAPTER 5

THEOREM PROVING AND PROGRAM SYNTHESIS

The composition algorithm described in Section 4.5 is a reduction algorithm that translates any expression involving traversals, equalities, constructions, and variables into a uniform traversal combinator form. The latter, by including traversals over single variables only, provides a uniformity that aids theorem proving. Checking whether two programs have the same functionality is facilitated by expressing the program as a uniform traversal combinator. More specifically, the uniqueness property described in Section 4.2 says that there is only one way of expressing a function as a traversal over a certain variable. The equality combinator though, needed for extending the expressiveness of our algebra, introduces new possible alternative expressions for a function. Luckily, testing if two such alternatives have the same functionality is done by using the same algorithm for checking functional equalities to test if the two parts of the equality are equal. This is a unification process as it needs to generate bindings from variables to programs to be used in other parts of a program. For example, testing whether the program $(x=y) \Rightarrow (\text{succ}(x)=\text{succ}(y))$ is always equal to the program true can be achieved only after considering in the second equality the binding from x to y derived from the first equality.

This chapter considers the problem of theorem proving and program synthesis using uniform traversal combinators. Section 5.1 presents the algorithm for testing the equality of two uniform traversal combinators. It is used in Section 5.2 for proving first-order theorems. Section 5.3 extends the theorem proving process to

cover sets and vectors. Section 5.4 addresses theorems involving variables with restricted types, that is, types augmented by integrity constraints. Section 5.5 presents an algorithm for proving second-order theorems by exhaustively checking program patterns. The same algorithm is used for constructing programs from proofs. This program synthesis will be the core of the query optimization algorithm presented in Section 6.4. Section 5.6 proposes a method for proving meta-theorems, that is, theorems that hold for any canonical type or for any program of any type. Finally, Section 5.7 concludes this chapter.

5.1 Equality of Uniform Traversal Combinators

Proving functional equalities is a necessary process for proving equality theorems. The following algorithm tests whether any two uniform traversal combinators compute the same function. It is based on the uniqueness property that says that there is a unique way for expressing a function as a traversal combinator. We will use this algorithm in Section 5.2 for proving equality theorems.

Theorem 5.1: (Equality Theorem) Let $\lambda\bar{x}.f(\bar{x})$ and $\lambda\bar{x}.g(\bar{x})$ be two uniform traversal combinators. Then $\forall\bar{x} : f(\bar{x}) = g(\bar{x})$ iff $\mathcal{E}(\lambda\bar{x}.f(\bar{x}), \lambda\bar{x}.g(\bar{x}), [], \mathbf{true}) \neq \mathbf{fail}$ and $\forall\bar{x} : f(\bar{x}) \neq g(\bar{x})$ iff $\mathcal{E}(\lambda\bar{x}.f(\bar{x}), \lambda\bar{x}.g(\bar{x}), [], \mathbf{false}) \neq \mathbf{fail}$, where \mathcal{E} is computed by the following algorithm:

Algorithm 5.1: (Equality of Combinators) The algorithm in Figure 5.1 that tests the functional equality of two uniform traversal combinators is a unification algorithm. It returns a substitution list that contains bindings from variables to combinators. More specifically, the substitution list contains bindings of the form $x = e$ or $x \neq e$, indicating that variable x is always equal (or not equal) to the combinator e . Expression $\mathcal{E}(f, g, \rho, \beta)$ takes two combinators f and g , a substitution list ρ , and a boolean value β as input. If $f = g$ (or $f \neq g$

$$\begin{array}{l}
1) \mathcal{E}(f, g, \text{fail}, \beta) \longrightarrow \text{fail} \\
2) \mathcal{E}(\lambda \bar{x}.z, \lambda \bar{x}.z, \rho, \beta) \longrightarrow \text{if } \beta \text{ then } \rho \text{ else fail} \\
3) \mathcal{E}(g, \lambda \bar{x}.z, \rho, \beta) \longrightarrow \begin{cases} \text{if } \rho \vdash z = h \text{ then } \mathcal{E}(g, h, \rho, \beta) \\ \text{else if } \rho \vdash z \neq h \text{ then } \mathcal{E}(g, h, \rho, \neg\beta) \\ \text{else if } \beta \text{ then } \rho[z = g] \text{ else } \rho[z \neq g] \end{cases} \\
4) \mathcal{E}(\lambda \bar{x}.C(\dots, e_i(\bar{x}), \dots), \lambda \bar{x}.C(\dots, h_i(\bar{x}), \dots), \rho, \beta) \\ \quad \longrightarrow \mathcal{E}(e_1, h_1, \mathcal{E}(e_2, h_2, \mathcal{E}(\dots \rho, \beta), \beta), \beta) \\
5) \mathcal{E}(\lambda \bar{x}.C_1(\dots), \lambda \bar{x}.C_2(\dots), \rho, \beta) \longrightarrow \text{fail} \\
6) \mathcal{E}(g, \mathcal{E}_T(f, h, \phi), \rho, \beta) \\ \quad \longrightarrow \begin{cases} \text{let } \rho' \leftarrow \mathcal{E}(g, \Phi(\phi(\text{true}), [], []), \mathcal{E}(f, h, \rho, \text{true}), \beta) \text{ in} \\ \text{if } \rho' \neq \text{fail} \text{ then } \rho' \\ \text{else } \mathcal{E}(g, \Phi(\phi(\text{false}), [], []), \mathcal{E}(f, h, \rho, \text{false}), \beta) \end{cases} \\
7) \mathcal{E}(\lambda \bar{x}.\mathcal{H}_T(\dots, f_i, \dots)(y), \lambda \bar{x}.\mathcal{H}_T(\dots, g_i, \dots)(y), \rho, \beta) \\ \quad \longrightarrow \mathcal{E}(f_1, g_1, \mathcal{E}(f_2, g_2, \mathcal{E}(\dots \rho, \beta), \beta), \beta) \\
8) \mathcal{E}(\lambda \bar{x}.g(z), \lambda \bar{x}.\mathcal{H}_T(\dots, \phi_i, \dots)(z), \rho, \beta) \\ \quad \longrightarrow \begin{cases} \mathcal{E}(\Phi(g(C_1(\bar{x}_1, \bar{y}_1)), [], []), \Phi(\phi_1(\bar{x}_1, \bar{y}_1, \dots, g(y_k^1), \dots), [], []), \\ \mathcal{E}(\Phi(g(C_2(\bar{x}_2, \bar{y}_2)), [], []), \Phi(\phi_2(\bar{x}_2, \bar{y}_2, \dots, g(y_k^2), \dots), [], []), \\ \mathcal{E}(\dots, \rho, \beta), \beta) \end{cases}
\end{array}$$

Figure 5.1 Equality of uniform traversal combinators

when $\beta = \text{false}$) then $\mathcal{E}(f, g, \rho, \beta)$ returns ρ extended with the new bindings found during the unification. Otherwise it returns **fail**. The right sides of the fourth and the two last rules have the form $\mathcal{E}(f_1, g_1, \mathcal{E}(f_2, g_2, \mathcal{E}(\dots \rho, \beta), \beta), \beta)$. This tests whether all $f_i = g_i$ but it also accumulates all the bindings starting with ρ . The first rule says that if ρ is **fail** (because it was the result of some other equality that failed) then this equality must fail too. The second rule says that a projection is equal to itself. The third rule checks the equality $z = g(\bar{x})$ (or $z \neq g(\bar{x})$): if there is a binding $z = h(\bar{x})$ in ρ then $g = h$ (or $g \neq h$) must be true; if there is a binding $z \neq h(\bar{x})$ in ρ then $g \neq h$ (or $g = h$) must be true; otherwise ρ is extended to include the binding $z = g(\bar{x})$ (or $z \neq g(\bar{x})$). The fourth and fifth rules check constructions: two constructions

are equal only if they formed by the same constructor. The sixth rule says that $g = \phi(f = h)$ if $g = \phi(\text{true})$ provided that $f = h$, or $g = \phi(\text{false})$ provided that $f \neq h$. Note that if $f = h$ (or $f \neq h$) returns **fail** then this means that $\phi(f = g)$ is always equal to $\phi(\text{false})$ (or to $\phi(\text{true})$). The seventh rule says that two traversals over the same variable y are equal if their component functions are equal. The last rule considers every construction $C_i(\bar{x}_i, \bar{y}_i)$ of T to check the equality $g(C_i(\bar{x}_i, \bar{y}_i)) = \phi_i(\bar{x}_i, \bar{y}_i, \dots, g(y_k^i), \dots)$. Note that if z is used by any of ϕ_i then $z = C_i(\bar{x}_i, \bar{y}_i)$. For example, for `tc_list` the last rule becomes:

$$\begin{aligned} & \mathcal{E}(\lambda \bar{x}.g(z), \lambda \bar{x}.\text{tc_list}(\phi_1, \phi_2)(z), \rho, \beta) \\ & \longrightarrow \begin{cases} \mathcal{E}(\Phi(g(\text{nil}), [], []), \phi_1(), \\ \mathcal{E}(\Phi(g(\text{cons}(a, l)), [], []), \Phi(\phi_2(a, l, g(l)), [], []), \rho, \beta), \beta) \end{cases} \end{aligned}$$

For the purpose of simplifying the algorithm in Figure 5.1 we ignored variable renaming and assumed that we have the same variables \bar{x} in both sides of the equation $\mathcal{E}(\lambda \bar{x}.f, \lambda \bar{x}.g, \rho, \beta)$. In addition, we did not put the reflective image of some rules, such as a rule for $\mathcal{E}(\lambda \bar{x}.z, g, \rho, \beta)$ because it is similar to the rule for $\mathcal{E}(g, \lambda \bar{x}.z, \rho, \beta)$.

Proof: We will prove the last two rules in Figure 5.1. All the other rules are obvious. From the uniqueness property we have:

$$\forall i \forall \bar{x}_i \forall \bar{y}_i : g(C_i(\bar{x}_i, \bar{y}_i)) = \phi_i(\bar{x}_i, \bar{y}_i, g(y_1^i), \dots, g(y_{i_r}^i)) \Leftrightarrow g = \mathcal{H}_T(\phi_1, \dots, \phi_n)$$

That is, g is equal to $\mathcal{H}_T(\dots, \phi_i, \dots)$ if and only if for all i : $g(C_i(\bar{x}_i, \bar{y}_i))$ is equal to $\phi_i(\bar{x}_i, \bar{y}_i, g(y_1^i), \dots, g(y_{i_r}^i))$ (this is the last rule). The seventh rule serves as the bottom case for the last rule to avoid falling into an infinite recursion and it is also derived from the uniqueness property. \square

For example, suppose that we want to prove the commutativity law for integer addition $x+y==y+x$, where `==` here is the structural equality `equal_int` for integers. This is expressed as:

$$\text{tc_int}([\] \rightarrow y, [?, i] \rightarrow \text{succ}(i))(x) == \text{tc_int}([\] \rightarrow x, [?, j] \rightarrow \text{succ}(j))(y)$$

Let $g(y) = \text{tc_int}([\] \rightarrow y, [\ ?, i] \rightarrow \text{succ}(i))(x)$ then we apply the uniqueness property for the equality $g(y) == \text{tc_int}([\] \rightarrow x, [\ ?, j] \rightarrow \text{succ}(j))(y)$ (last rule in Figure 5.1):

- 1) $y = \text{zero}$: $(g(\text{zero}) == x) = (\text{tc_int}([\] \rightarrow \text{zero}, [\ ?, i] \rightarrow \text{succ}(i))(x) == x) = \text{true}$
- 2) $y = \text{succ}(j)$: $(g(\text{succ}(j)) == \text{succ}(g(j))) =$
 $(\text{tc_int}([\] \rightarrow \text{succ}(j), [\ ?, i] \rightarrow \text{succ}(i))(x) == \text{succ}(g(j)))$

Let $f(x) = \text{succ}(g(j))$. We apply the uniqueness property again:

- 1) $x = \text{zero}$: $(f(\text{zero}) == \text{succ}(j)) = (\text{succ}(j) == \text{succ}(j)) = \text{true}$
- 2) $x = \text{succ}(i)$: $(f(\text{succ}(i)) == \text{succ}(f(i))) = (\text{succ}(f(i)) == \text{succ}(f(i))) = \text{true}$

Another example that demonstrates the use of the binding list is for testing the equality between $(x == y) \Rightarrow (\text{succ}(x) == \text{succ}(y))$ and true. This equality in combinator form is:

$$\text{eq_int}([\] \rightarrow x, [\] \rightarrow y, [z] \rightarrow \text{tc_bool}([\] \rightarrow \text{eq_int}([\] \rightarrow \text{succ}(x), [\] \rightarrow \text{succ}(y), [w] \rightarrow w),$$

$$[\] \rightarrow \text{true})(z) = \text{true}$$

From the sixth rule in Figure 5.1 we have two cases:

- 1) $x = y \Rightarrow \text{test eq_int}([\] \rightarrow \text{succ}(x), [\] \rightarrow \text{succ}(y), [w] \rightarrow w) = \text{true} / [x=y]$
 $\text{test succ}(x) = \text{succ}(y) / [x=y]$
 $\text{test } x = y / [x=y]$
- 2) $x \neq y \Rightarrow \text{test true} = \text{true} / [x \neq y]$

Both cases do not fail. Therefore, the equality is true. Similarly, the equality between $x == y$ and true is not true, which is a correct statement because x and y are universally quantified variables.

5.2 Theorem Proving

The algorithms for composing uniform traversal combinators and for testing their functional equalities can be applied to proving theorems about uniform traversal combinators. Suppose that we want to prove that an expression $e(\bar{x})$ is always true. First we need to find all uniform traversal combinators associated with each function call in e . Then we use the composition algorithm to synthesize the uniform traversal combinator $f(\bar{x})$ equivalent to $e(\bar{x})$. Finally we are left with the simpler task of proving whether $f(\bar{x})$ is always true (a tautology). The following corollary comes directly from Theorem 5.1:

Corollary 5.1: (Tautology Checker) Let $f(\bar{x})$ be a uniform traversal combinator of type boolean. Then $\forall \bar{x} : f(\bar{x})$ iff $\mathcal{E}(f(\bar{x}), \text{true}, [], \text{true}) \neq \text{fail}$.

Note that we can also disprove a theorem $f(\bar{x})$ by testing $\mathcal{E}(f(\bar{x}), \text{false}, [], \text{true}) \neq \text{fail}$.

For example, suppose that we want to prove the associativity law for integer multiplication, that is: $(x*y)*z == x*(y*z)$, where multiplication is computed by:

$$x*y = \text{tc_int}([\] \rightarrow \text{zero}, [?,i] \rightarrow \text{tc_int}([\] \rightarrow i, [?,j] \rightarrow \text{succ}(j))(y))(x)$$

We start by composing $(x*y)*z$. We apply the promotion theorem with

$$g(u) = u*z = \text{tc_int}([\] \rightarrow \text{zero}, [?,i] \rightarrow \text{tc_int}([\] \rightarrow i, [?,j] \rightarrow \text{succ}(j))(z))(u)$$

$$\text{Let } g(\text{tc_int}([\] \rightarrow \text{zero}, [?,i] \rightarrow \text{tc_int}([\] \rightarrow i, [?,j] \rightarrow \text{succ}(j))(y))(x)) = \text{tc_int}(f_1, f_2)(x).$$

Then from the promotion theorem we have:

$$1) f_1() = g(\text{zero}) = \text{zero}$$

$$2) f_2(?,g(i)) = g(\text{tc_int}([\] \rightarrow i, [?,j] \rightarrow \text{succ}(j))(y)) = \text{tc_int}(h_1, h_2)(y)$$

Let $m=g(i)$. We apply the promotion theorem again:

- 1) $h1() = g(i) = m$
- 2) $h2(? , g(j)) = g(\text{succ}(j)) = \text{tc_int}([\] \rightarrow g(j), [?, k] \rightarrow \text{succ}(k))(z)$
 $\Rightarrow h2(? , w) = \text{tc_int}([\] \rightarrow w, [?, k] \rightarrow \text{succ}(k))(z)$

Therefore, $(x*y)*z$ is

$$\text{tc_int}([\] \rightarrow \text{zero},$$

$$[?, m] \rightarrow \text{tc_int}([\] \rightarrow m, [?, w] \rightarrow \text{tc_int}([\] \rightarrow w, [?, k] \rightarrow \text{succ}(k))(z))(y))(x)$$

We will compose now $x*(y*z)$:

$$x*(y*z) = \text{tc_int}([\] \rightarrow \text{zero}, [?, i] \rightarrow \text{tc_int}([\] \rightarrow i, [?, j] \rightarrow \text{succ}(j))(y*z))(x)$$

Let $g(u) = u+i = \text{tc_int}([\] \rightarrow i, [?, j] \rightarrow \text{succ}(j))(u)$. Expression $x*(y*z)$ becomes:

$$g(y*z) = \text{tc_int}(f1, f2)(y)$$

$$= \text{tc_int}([\] \rightarrow \text{zero}, [?, u] \rightarrow \text{tc_int}([\] \rightarrow u, [?, w] \rightarrow \text{succ}(w))(z))(y)$$

and the promotion theorem gives:

- 1) $f1() = g(\text{zero}) = i$
- 2) $f2(? , g(u)) = g(\text{tc_int}([\] \rightarrow u, [?, w] \rightarrow \text{succ}(w))(z)) = \text{tc_int}(h1, h2)(z)$

Let $k=g(u)$. We apply the promotion theorem again:

- 1) $h1() = g(u) = k$
- 2) $h2(? , g(w)) = g(\text{succ}(w)) = \text{succ}(g(w)) \Rightarrow h2(? , m) = \text{succ}(m)$

Therefore, $x*(y*z)$ is

$$\text{tc_int}([\] \rightarrow \text{zero},$$

$$[?, i] \rightarrow \text{tc_int}([\] \rightarrow i, [?, k] \rightarrow \text{tc_int}([\] \rightarrow k, [?, m] \rightarrow \text{succ}(m))(z))(y))(x)$$

Finally, we can see that $(x*y)*z$ is equal to $x*(y*z)$ (by applying the seventh rule in Figure 5.1 twice).

5.3 Proving Set and Vector Theorems

In Section 4.8.2 we defined the properties of uniform traversal combinator for sets. We saw that if $\text{tc_set}(f_1, f_2)$ is order-independent then set combinators are composed much like the combinators for regular recursive types. Unfortunately testing set equalities is not so easy. The fourth rule in Figure 5.1 is not true any more because $\text{insert}(a, r) = \text{insert}(b, s)$ does not necessarily implies that $a = b$ and $r = s$. In the following extension of Algorithm 5.1 the last rule says that if f and g are expressions of type set then $f = g \Rightarrow f \subseteq g \wedge g \subseteq f$. It must be used whenever the other rules are not applicable. The second rule is derived from the uniqueness property for sets (Theorem 4.8 on page 84). Here we assume that a is the chosen element of a set and s the rest and therefore $a \notin s$:

$$\begin{array}{l}
 \mathcal{E}(\lambda \bar{x}. \text{tc_set}(f_1, f_2)(z), \lambda \bar{x}. \text{tc_set}(g_1, g_2)(z), \rho, \beta) \longrightarrow \mathcal{E}(f_1, g_1, \mathcal{E}(f_2, g_2, \rho, \beta), \beta) \\
 \mathcal{E}(\lambda \bar{x}. g(z), \lambda \bar{x}. \text{tc_set}(\phi_1, \phi_2)(z), \rho, \beta) \longrightarrow \begin{cases} \mathcal{E}(\Phi(g(\text{emptyset}), [], []), \phi_1(), \\ \mathcal{E}(\Phi(g(\text{insert}(a, s)), [], []), \\ \Phi(\phi_2(a, s, g(s)), [], []), \rho, \beta), \beta) \end{cases} \\
 \mathcal{E}(f, g, \rho, \beta) \longrightarrow \begin{cases} \mathcal{E}(\Phi(f \subseteq g, [], []), \text{true}, \mathcal{E}(\Phi(g \subseteq f, [], []), \text{true}, \rho, \beta), \beta) \\ \text{(where } f \text{ and } g \text{ are of type set)} \end{cases}
 \end{array}$$

where $x \subseteq y$ is computed by:

$$\text{tc_set}([\] \rightarrow \text{true}, [a, ?, s] \rightarrow a \text{ and } \text{tc_set}([\] \rightarrow \text{false}, [b, ?, r] \rightarrow (a=b) \text{ or } r)(y))(x)$$

For example, the following proves that $\text{member}(a, s) \Rightarrow \text{insert}(a, s) = s$, where:

$$\text{member}(a, s) = \text{tc_set}([\] \rightarrow \text{false}, [b, ?, r] \rightarrow \text{if } a=b \text{ then true else } r)(s)$$

$$(x \Rightarrow y) = \text{tc_boolean}([\] \rightarrow y, [\] \rightarrow \text{true})(x)$$

This theorem is expressed as (after applying the composition algorithm):

$$\text{tc_set}([\] \rightarrow \text{true}, [b, ?, r] \rightarrow \text{if } a=b \text{ then } \text{insert}(a, s) = s \text{ else } r)(s)$$

Therefore, we need to test the equality:

$$\begin{aligned} \text{tc_set}([\] \rightarrow \text{true}, [b, ?, r] \rightarrow \text{if } a=b \text{ then insert}(a, s)=s \text{ else } r)(s) &= \text{true} \\ \Rightarrow g(s) &= \text{true} \end{aligned}$$

The uniqueness property for $s = \text{emptyset}$ and $s = \text{insert}(b, l)$ yields:

- 1) $\text{true} = \text{true}$
- 2) $\text{true} = \text{if } a=b \text{ then insert}(a, \text{insert}(b, l)) = \text{insert}(b, l) \text{ else } g(l)$

The second equality is true because $\text{insert}(a, \text{insert}(b, l)) = \text{insert}(b, l)$ is true (when the binding $[a=b]$ is used).

Testing equalities of vector combinators is more difficult. Here we extend the algorithm in Figure 5.1 to include the following rules:

$$\begin{aligned} &\mathcal{E}(\lambda \bar{x}. \text{tc_vector}(f_1, f_2)(z), \lambda \bar{x}. \text{tc_vector}(g_1, g_2)(z), \rho, \beta) \\ &\quad \longrightarrow \mathcal{E}(f_1, g_1, \mathcal{E}(f_2, g_2, \rho, \beta), \beta) \\ \mathcal{E}(f, g, \rho, \beta) &\longrightarrow \left\{ \begin{array}{l} \mathcal{E}(\Phi(\text{tc_vector}([\] \rightarrow \text{size}(f) = \text{size}(g), \\ \quad [i, v, r] \rightarrow r \text{ and } \text{index}(i, g) = v)(f), [], []), \text{true}, \rho, \beta) \\ \text{(where } f \text{ and } g \text{ are of type vector)} \end{array} \right. \end{aligned}$$

where $\text{index}(k, a)$ is $\text{tc_vector}([\] \rightarrow ?, [i, v, r] \rightarrow \text{if } i=k \text{ then } v \text{ else } r)(a)$ and $\text{size}(a)$ is $\text{tc_vector}([\] \rightarrow 0, [?, ?, s] \rightarrow s+1)(a)$.

5.4 Proving Theorems about Restricted Types

A type definition with a where clause restriction has the following form (explained in Section 3.1.7):

$$T' = T \text{ where}(x) f(x);$$

Whenever we prove theorems about values of type T' we must consider the constraint $f(x)$ that any value $x:T'$ satisfies. Furthermore, if T is also restricted or has a reference to a restricted type then all these restrictions must be considered. The following

'and-all' combinator is used for accumulating (anding together) all these restrictions. For the purpose of explanation we define this combinator for canonical types with direct recursions only.

Definition 5.1: (And-all Combinator) Let $T(\bar{\alpha})$ be a canonical type. An and-all combinator $\mathcal{A}_T(h_1, \dots, h_r)$, where $h_i : [\alpha_i] \rightarrow \text{boolean}$, is $\mathcal{H}_T(f_1, \dots, f_n)$, where each $f_i = \lambda \bar{x}_i. \lambda \bar{y}_i. \lambda \bar{z}_i. (\bigwedge_j \mathcal{R}_{i,j}(x_j^i)) \wedge (\bigwedge_j z_j^i)$ and

$$\mathcal{R}_{i,j} = \begin{cases} h_s & \text{if } T_{i,j}(\bar{\alpha}) = \alpha_s \text{ for some } s \\ \mathcal{A}_{T_{i,j}}(h_1, \dots, h_r) & \text{if } T_{i,j}(\bar{\alpha}) \text{ contains at least one of } \bar{\alpha} \\ \lambda x. \text{true} & \text{otherwise} \end{cases}$$

where x_j is of type $T_{i,j}(\bar{\alpha}) \equiv t_{i,j}(\bar{\alpha}, T(\bar{\alpha}))$.

For example, $[f] \rightarrow \text{tc_list}([\] \rightarrow \text{true}, [a, ?, r] \rightarrow f(a) \text{ and } r)$ is the and-all combinator for lists.

Definition 5.2: (Accumulated Restrictions) Let T' be a canonical type enhanced with restrictions. Then the accumulated restriction $\mathcal{I}\{T'\}$ is defined as:

$$\mathcal{I}\{T'\} = \begin{cases} \mathcal{A}_T(\mathcal{I}\{t_1\}, \dots, \mathcal{I}\{t_n\}) & \text{if } T' = T(t_1, \dots, t_n) \\ \lambda x. (f(x) \text{ and } \mathcal{I}\{T\}(x)) & \text{if } T' = T \text{ where } (y) f(y) \\ \lambda x. \text{true} & \text{otherwise} \end{cases}$$

Theorem 5.2: Let $f(\bar{x})$ be a theorem with universally quantified variables $x_i : T_i$. Then $f(\bar{x})$ is true for all $x_i : T_i$ in \bar{x} iff $\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow f(\bar{x})$ is true.

Proof: Any value $x_i : T_i$ must satisfy $\mathcal{I}\{T_i\}$. \square

For example, suppose that type `tt` is defined as:

`tt = list(range[0,5]) where(x) length(x) ≤ 10;`

and we want to prove that for any value `x:tt` we have `tc_list([\] → 0, [a, ?, r] → a+r)(x) ≤ 50`.

This theorem is expressed as follows:

$$\begin{aligned}
& (\text{tc_list}([\] \rightarrow \text{true}, [a, ?, r] \rightarrow a \geq 0 \text{ and } a \leq 5 \text{ and } r)(x) \\
& \text{and } \text{tc_list}([\] \rightarrow 0, [?, ?, r] \rightarrow r+1)(x) \leq 10) \\
& \Rightarrow \text{tc_list}([\] \rightarrow 0, [a, ?, r] \rightarrow a+r)(x) \leq 50
\end{aligned}$$

Note that restrictions do not necessarily decrease the chances of a theorem to be true. For example, the theorem $\text{succ}(x) == \text{succ}(y)$ is false, but if we add the restriction $x == y$ then the theorem becomes $x == y \Rightarrow \text{succ}(x) == \text{succ}(y)$ which is true. This is a very important observation because redundancies of access paths are expressed that way, causing two different programs to be equivalent if they use either of these paths. For example:

```

slist(alpha) = struct msl ( info: list(alpha), size: int )
                where(x) x.size=length(x.info);

```

The theorem $\text{length}(x.\text{info}) + \text{length}(y.\text{info}) == x.\text{size} + y.\text{size}$, for any x and y of type $\text{slist}(\text{alpha})$, is expressed as:

$$\begin{aligned}
& (x.\text{size} = \text{length}(x.\text{info}) \text{ and } y.\text{size} = \text{length}(y.\text{info})) \\
& \Rightarrow \text{length}(x.\text{info}) + \text{length}(y.\text{info}) == x.\text{size} + y.\text{size}
\end{aligned}$$

which is true.

5.5 Proving Second-order Theorems and Synthesizing Programs

So far we have proved first-order theorems only. In Section 4.8.5 the definition of uniform traversal combinators was extended to include second-order functions: $\mathcal{F}_T(c, r_1, \dots, r_n)(f) = c(f(r_1(), \dots, r_n()))$. But if a theorem contains such an expression in which variable f appears free then this theorem becomes a second-order theorem. Unfortunately, there may be more than one uniform traversal combinator f that satisfies this theorem. Even though it is not difficult to check whether two programs compute the same function (Algorithm 5.1), it is not easy to generate all

possible programs equivalent to a specific program, or even to test if there is at least one uniform traversal combinator f that satisfies $\mathcal{F}_T(c, r_1, \dots, r_n)(f) = g$, for some given g, c, r_1, \dots, r_n . If Algorithm 5.1 is extended to include higher-order equalities then it becomes a higher-order unification algorithm.

This section presents a method for generating all possible programs f that satisfy $\mathcal{E}(\mathcal{F}_T(c, r_1, \dots, r_n)(f), g, \rho, \beta) \neq \text{fail}$. We want to generate all possible programs, instead of just one, because this algorithm will be used for program optimization later, where programs need to be compared to find the cheapest. There may be none, one, more than one, or infinite number of uniform traversal combinators f that satisfy the theorem $\mathcal{F}_T(c, r_1, \dots, r_n)(f) = g$. There are two cases with infinite number of f here: when the theorem is true for all f (that is, when f is universally quantified) and when it is true for an infinite family of functions, such as f in the theorem $\text{fc0}([z] \rightarrow z > 8)(f)$. In that case f is existentially quantified¹. For example, there are 101 pairs of f_1 - f_2 that satisfy the theorem $f_1() + f_2() = 100$.

The following extension of Algorithm 5.1 assumes that there are either a finite number of solutions to the second-order theorem or that this theorem is universally quantified (it is true for all functions f). The case where there is an infinite family of functions f will be considered later. First, ρ is changed to be a set of binding lists. That way if f is bound in ρ it is associated with a set of expressions instead of just one. We denote this as $\rho \vdash \{f = e_1, \dots, f = e_k\}$. In the following algorithm we will write $\rho \vdash f = e$ to indicate that e is one of e_i in $\rho \vdash \{f = e_1, \dots, f = e_k\}$. The equality algorithm in Figure 5.1 is extended to include the following:

¹Note that the theorem $\exists x : f(x)$ (i.e. x is an existentially quantified variable) is proved by setting $x = \mathcal{F}_T(\lambda x.x)(g)$. and proving the theorem as a second-order theorem.

Algorithm 5.2: (Higher-order Unification)

$$\begin{aligned} & \mathcal{E}(\mathcal{F}_T(c, r_1, \dots, r_n)(f), \mathcal{F}_T(e, s_1, \dots, s_n)(f), \rho, \beta) \\ & \quad \longrightarrow \mathcal{E}(c, e, \mathcal{E}(r_1, s_1, \mathcal{E}(r_2, s_2, \mathcal{E}(\dots, \rho, \beta), \beta), \beta), \beta) \\ & \mathcal{E}(\mathcal{F}_T(c, r_1, \dots, r_n)(f), g, \rho, \beta) \longrightarrow \text{the set of all valid } f \text{ given in Figure 5.2} \end{aligned}$$

The first rule is obvious and serves as a bottom case for the second rule. If it does not fail then the equality is true for any f . If the second rule does not fail then it returns a binding that binds f to a set of combinators. In Figure 5.2 we consider all possible solutions for f . The set of valid solutions for f is the union of the solutions derived from all the rules that do not fail. To make the algorithm simpler we assume that no free variables can occur in the body of f . The first rule in Figure 5.2 checks whether f could be a projection: it tries any projection $\lambda\bar{x}.x_k$ to see if it fails. The second rule tries every constructor C_k of T_0 (the output type of f). As we do not know the components e_i of these constructions we set them as free variables f_i of function type. The same algorithm is used recursively to extract a solution for them and, therefore, the resulting binding will include bindings from f_i to a set of valid e_i . The third rule checks whether f is a traversal over one of the non-accumulative result variables \bar{x} . Again the components of the traversal are unknown and therefore they are set as free variables. The last rule checks whether f is an equality combinator.

We can see that this algorithm tries exhaustively all possible program patterns, but in practice a lot of impossible programs fail quickly and the tree of possible solutions is pruned very fast. If we were allowed to have traversals on traversals in our algebra then this algorithm would be very inefficient as we would have a very large number of program patterns to test. The restriction of uniform traversal combinators that traversals are over variables reduces the number of patterns. If there is an infinite set of solutions that satisfies $\mathcal{F}_T(c, r_1, \dots, r_n)(f) = g$, such as

$$\begin{array}{l}
\forall k \in [1, n] \text{ all } f = \lambda \bar{x}. x_k : \mathcal{E}(\Phi(c(r_k), [], []), g, \rho, \beta) \neq \text{fail} \\
\forall C_k \text{ all } f = \lambda \bar{x}. C_k(\dots, e_i(\bar{x}), \dots) : \\
\quad \mathcal{E}(\Phi(c(C_k(\dots, \mathcal{F}_{T_i}(\lambda x.x, r_1, \dots, r_n)(f_i), \dots)), [], []), g, \rho, \beta) \vdash f_i = e_i \\
\forall k \in [1, n] \text{ all } f = \lambda \bar{x}. \mathcal{H}_{T_k}(\dots, e_i, \dots)(x_k) : \\
\quad \mathcal{E}(\Phi(c(\mathcal{H}_{T_k}(\dots, \lambda \bar{z}_i. \mathcal{F}_{T_i}(\lambda x.x, \bar{z}_i)(f_i), \dots))(r_k), [], []), g, \rho, \beta) \vdash f_i = e_i \\
\text{all } f = \mathcal{E}_T(e_1, e_2, e_3) : \left\{ \begin{array}{l} \mathcal{E}(\Phi(c(\mathcal{E}_T(\mathcal{F}_T(\lambda x.x, r_1, \dots, r_n)(f_1), \\ \mathcal{F}_T(\lambda x.x, r_1, \dots, r_n)(f_2), \\ \mathcal{F}_{T'}(\lambda x.x, \lambda x.x)(f_3))), [], []), g, \rho, \beta) \vdash f_i = e_i \end{array} \right.
\end{array}$$

Figure 5.2 Solution of $\mathcal{E}(\mathcal{F}_T(c, r_1, \dots, r_n)(f), g, \rho, \beta)$

in the equality $\text{fc0}([z] \rightarrow z > 8)(f) = \text{true}$ whose set of solutions is $f = 9, 10, 11, \dots$, then this algorithm will fall into an infinite loop. To avoid this, we could set a depth limit to restrict the search in Algorithm 5.2 or we could set a limit to the number of solutions that this algorithm extracts. A combination of these two methods would be more preferable: the tree of possible solutions is searched to a certain depth; if the extracted solutions are at least equal to the prespecified number of solutions then we stop; otherwise we double the depth limit and repeat the search. In Section 6.4 we will present a method for guiding the search performed by this algorithm using cost functions. This method eliminates all programs that seem to be very inefficient from the beginning.

As an example of the first rule of Algorithm 5.2, we prove that the list map distributes over append:

$$\text{map_list}(f)(\text{append}(x, y)) = \text{append}(\text{map_list}(f)(x), \text{map_list}(f)(y))$$

where $\text{map_list}(f) = \text{tc_list}([] \rightarrow \text{nil}, [b, ?, r] \rightarrow \text{fc1}([z] \rightarrow \text{cons}(z, r), [] \rightarrow b)(f))$ (function fc_n for a number n is defined in Section 4.8.5). We apply the promotion theorem for $g(\text{append}(x, y)) = \text{tc_list}(f_1, f_2)(x)$, where $g = \text{map_list}(f)$:

$$f_1() = g(y) = \text{map_list}(f)(y)$$

$$f_2(b, ?, g(r)) = g(\text{cons}(b, r)) = \text{fc1}([z] \rightarrow \text{cons}(z, g(r)), [] \rightarrow b)(f)$$

$$\Rightarrow f2(b,?,u) = fc1([z] \rightarrow cons(z,u), [] \rightarrow b)(f)$$

Therefore, $map_list(f)(append(x,y))$ is

$$tc_list([] \rightarrow map_list(f)(y), [b,?,u] \rightarrow fc1([z] \rightarrow cons(z,u), [] \rightarrow b)(f))(x)$$

Similarly, $append(map_list(f)(x), map_list(f)(y))$ is equal to:

$$tc_list([] \rightarrow map_list(f)(y), [a,?,s] \rightarrow cons(a,s))(map_list(f)(x))$$

which is the same with $map_list(f)(append(x,y))$.

The previous algorithm can be used directly for synthesizing programs:

Algorithm 5.3: (Program Synthesis) Let $g(f_1, \dots, f_n)$ be a second-order theorem, where f_1, \dots, f_n are free variables of function type. Then the set of all possible uniform traversal combinators e_1, \dots, e_n that satisfy $g(e_1, \dots, e_n)$ is derived by $\mathcal{E}(g(f_1, \dots, f_n), true, []) \vdash f_i = e_i$.

Note that, if there is no such binding $f_i = e_i$ in $\mathcal{E}(g(f_1, \dots, f_n), true, [])$ then f_i is universally quantified.

For example, suppose that we want to prove the theorem:

$$\begin{aligned} \text{length}(fc2([z] \rightarrow z, [] \rightarrow x, [] \rightarrow y)(g)) = \\ tc_list([] \rightarrow tc_list([] \rightarrow zero, [?,?,j] \rightarrow succ(j))(y), [?,?,i] \rightarrow succ(i))(x) \end{aligned}$$

where length is $tc_list([] \rightarrow zero, [?,?,i] \rightarrow succ(i))$. That is, we want to find every combinator g that satisfies the above equation. Let g be $tc_list(g1,g2)(x)$ (this is the third case in Figure 5.2). From the promotion theorem we have:

$$1) \text{length}(g1) = tc_list([] \rightarrow zero, [?,?,j] \rightarrow succ(j))(y)$$

let $g1 = tc_list(h1,h2)(y)$ then

$$1.1) \text{length}(h1) = zero$$

$[] \rightarrow zero$ is the only component of length that returns zero $\Rightarrow h1 = nil$

$$1.2) \text{length}(h2(c,s,j)) = \text{succ}(\text{length}(j))$$

$[?,?,i] \rightarrow \text{succ}(i)$ is the only component of length that returns succ .

Let $h2(c,s,j) = \text{cons}(x1,x2)$ then $\text{length}(\text{cons}(x1,x2)) = \text{succ}(\text{length}(j))$

$$\Rightarrow \text{succ}(\text{length}(x2)) = \text{succ}(\text{length}(j)) \Rightarrow h2(c,s,j) = \text{cons}(x1,j)$$

$$2) \text{length}(g2(b,r,i)) = \text{succ}(\text{length}(i)) \Rightarrow g2(b,r,i) = \text{cons}(x3,i)$$

Therefore, $g(x,y)$ is:

$\text{tc_list}([\] \rightarrow \text{tc_list}(\text{nil}, [?,?,j] \rightarrow \text{cons}(x1,j))(y), [?,?,i] \rightarrow \text{cons}(x3,i))(x)$

where $x1$ and $x3$ are universally quantified variables that yield a class of solutions for $g(x,y)$.

5.6 Proving Meta-theorems

Uniform traversal combinators are highly stereotyped functions. Therefore, we can prove general properties about traversals by using the same approach we adopted for proving properties about specific types: that is, by considering traversals as data structures and the algorithms and theorems about traversals as traversals over these structures. That way, we can prove theorems for a whole class of functions, such as the transitive law of structural equality for any type of structure, or that any generic map is a uniform traversal combinator, or that any generic map preserves the shape of a structure, or even the promotion theorem.

One possible type definition for representing uniform traversal combinators is:

```
utc = union common ( lambda_vars: binding(type),
                    accumulative_vars: list(string) )
  ( projection: struct project ( var: string ),
    construction: struct construct ( constructor: string,
                                    arguments: list(utc) ),
    traversal: struct traverse ( var: string,
```

components: list(utc)),

equalp: struct equality (left: utc, right: utc, cont: utc));

binding = list(pair(string,utc));

type = struct make_type (name: string, constructors: list(utc));

The 'common' part of the union indicates that all union alternatives include the components `lambda_vars` and `accumulative_vars` as part of their structure. These are the lambda variables (non-accumulative and accumulative result variables) of the combinator. The 'type' is the type of the type definitions. It has a name and a list of constructors C_1, \dots, C_n put in `utc` form: that is, as $\lambda \bar{x}_i \lambda \bar{y}_i. C_i(\bar{x}_i, \bar{y}_i)$.

The traversal combinator `tc_utc` for this structure can be derived just as for any other data structure. Most functions and theorems about uniform traversal combinators can also be expressed in terms of `tc_utc` and can be put into a uniform traversal combinator form. For example, we claim that the following two functions can be put in combinator form²:

function `compose` (`g`: utc, `h`: list(utc), `rho`: binding, `sigma`: binding) : utc;

function `eq_utc` (`g`: utc, `h`: utc, `rho`: binding, `beta`: boolean) : binding;

Function `compose`($g, (h_1, \dots, h_r), \rho, \sigma$) returns the composition $\Phi(g(h_1, \dots, h_r), \rho, \sigma)$, while function `eq_utc`(g, h, ρ, β) returns the binding $\mathcal{E}(g, h, \rho, \beta)$.

For example, suppose that we want to prove that $\forall x : \mathcal{H}_T(C_1, \dots, C_n)(x) = x$ for every canonical type T , where C_i is the i th constructor of T . This theorem is expressed as follows (here type `tp` appears free):

```
eq_utc(equality(cons(pair(x,tp),nil),nil,
                traverse(cons(pair(x,tp),nil),nil,x,tc_type([?.cl] →cl)(tp)),
                project(cons(pair(x,tp),nil),nil,x),
                project(cons(pair(y,boolean_type),nil),nil,y)),
        construct(nil,nil,"true",nil),nil)
```

²Further research is needed to justify this claim.

We need to prove that the above expression is a tautology. This can be done by using the same methodology for proving regular theorems.

5.7 Using the Theorem Prover for Program Translation

We have presented a complete method of proving theorems about uniform traversal combinators. Our method can prove and disprove theorems that involve both universally and existentially quantified variables by providing a solution for the later. This method was extended to include second-order expressions which was used for program synthesis by deriving programs from proofs.

This theorem prover is used in various stages of the transformation process to assure that the mapping is correct, that is, that the functionality in the abstract layer is preserved in the concrete layer, especially after the optimization stage which is usually incomplete. It is also used for testing whether a coding and a representation function is a valid pair of functions that satisfies Equation 3.2.

Another use of the theorem prover is verifying that transactions preserve the database integrity constraints [71]. Let db be the current database state and $F(\bar{x})$ a database transaction that updates the database state. For each such transaction we can derive a function $f(\bar{x}, db)$ that accepts the database db along with the extra parameters \bar{x} and returns a new database state. If we assume that db satisfies the database integrity constraints then the new database must do that too:

$$\mathcal{I}\{db\} \Rightarrow \mathcal{I}\{f(\bar{x}, db)\}$$

In case of inconsistency, that is, when the previous theorem fails, we could assume that there is a precondition $p(\bar{x})$ that variables from \bar{x} satisfy and prove the high order theorem:

$$\mathcal{I}\{db\} \wedge \mathcal{F}_T(\lambda z.z, \bar{x})(p) \Rightarrow \mathcal{I}\{f(\bar{x}, db)\}$$

If this theorem does not fail it will generate all possible solutions for p that could be displayed to the user as suggestions for additional preconditions attached to the transaction F for avoiding misusing the database consistency [75, 55].

The program synthesis algorithm can be used to derive all valid solutions of the high order theorem (Equation 3.3 on page 42):

$$r_0(F(x_1, \dots, x_n)) = f(r_1(x_1), \dots, r_n(x_n))$$

where F is an existentially quantified variable of type function. Algorithm 5.2 can synthesize all possible uniform traversal combinators F that satisfy this equation. This is also true for the general case of implementing f using mapping dependencies (Equation 3.1 on page 39):

$$m_1(x_1, y_1) \wedge \dots \wedge m_n(x_n, y_n) \Rightarrow m_0(f(x_1, \dots, x_n), F(y_1, \dots, y_n))$$

If there are constraints (restrictions) attached to the values $x_i : T_i$ then Equation 3.3 becomes:

$$\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow r_0(F(x_1, \dots, x_n)) = f(r_1(x_1), \dots, r_n(x_n))$$

which, like the previous theorems, can be used for generating all valid solutions for F . This form is very important because if we have redundant information kept as part of a value then the synthesis algorithm will generate all alternative ways of accessing this information. The objective of query optimization is to find the alternative with the minimal cost. In Section 6.4 we will present an algorithm for guiding the exhaustive enumeration of solutions in Algorithm 5.2 using a cost function to prune out solutions that do not look very promising. Another additional way of managing the intractability of the optimization is to let the implementor define both the coding and representation functions. Then the second-order theorem becomes:

$$F(x_1, \dots, x_n) = c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n)))$$

which yields fewer solutions for F , because F now has a unique functionality, even though there may be more than one alternative uniform traversal combinators that

compute F . This approach is preferable to the others as it yields solutions for F that are expressed in terms of concrete primitives only. The synthesis algorithm can be used during the design phase to suggest possible coding functions C_{t_1, t_2} assigned to a specific representation function R_{t_2, t_1} , generated during the proof of the second-order theorem:

$$R_{t_2, t_1}(r_1, \dots, r_n)(\mathcal{F}_T([z] \rightarrow z, [] \rightarrow x)(C_{t_1, t_2}(c_1, \dots, c_n))) = x$$

The program synthesis algorithm can also be used for translating view updates. Let db be a database and f a database view, that is, $f(db)$ is the database information accessible from the view. Let also u be a transaction that updates this view. We want to find a transaction U that when applied to the database db it causes the same effect as u . This is expressed as:

$$f(U(db)) = u(f(db))$$

This theorem is very similar to Equation 3.3 on page 42 and can be solved using the same technique.

CHAPTER 6

PROGRAM TRANSLATION AND OPTIMIZATION

The optimization problem appears when there are alternative paths to the same data in a database, because there is redundancy of information, or when there are alternative methods of execution, due to equivalences between programs. The redundancy of information is derived by expressing object dependencies as integrity constraints attached to the objects' types during system specification and implementation. Our objective is to access data as cheaply as possible. The query translation process can be seen as a search over alternative paths with the goal of reducing the total execution cost of a query. Consider for example a table in a relational system with an index on one of its columns. Suppose that we want to find a row in this table containing a specific value in its indexed attribute. We can either access the rows sequentially to locate the tuple or make use of the index. The first case needs time proportional to the size of the table, while the second case needs time proportional to the logarithm of its size. For large enough tables the second alternative is better than the first. We could use the theorem prover to guide the search for an optimal solution by proving that the cost of an access path is smaller than the cost of another, whenever the cost parameters, such as the relation cardinality, becomes large. For the example above, it is not difficult to prove that the second alternative is cheaper than the first. But if we have queries that depend on more than one parameter then a proof is not always possible, because increasing the value of one parameter and decreasing another may result

in a different optimal translation. One such example is the natural join operation whose optimal translation depends on the data stored in the join tables. This is a well known fact in database systems: the cost of query processing depends not only on the database schema itself, but also on the actual data stored in the database. Therefore, it is not enough to know how the parameter types are mapped to their concrete implementations but also we need to know some statistics about the actual data passed as parameters.

In this chapter we examine the problem of translating and optimizing programs expressed as unary traversal combinators. The type transformation model presented in Chapter 3 gives a method for translating a query in the abstract domain into a query in the concrete domain. More specifically, if we provide type mappings for the input values and the output value of a function then translating a function is translating the input values, applying the function, and then translating back the result. This default translation can be optimized by using the program synthesis algorithm presented in Section 5.5 to generate all possible uniform traversal combinators equivalent to some specific uniform traversal combinator. Deriving the best among these alternatives is performed by specifying a cost function that evaluates the quality and efficiency of a uniform traversal combinator.

Section 6.1 presents an extension to the type system for incorporating mapping specifications. Section 6.2 presents a complete example of mapping a database type that includes a part-subpart hierarchy. Section 6.3 describes a cost model well-suited to our algebra, flexible enough to capture many diverse cost models. Section 6.4 presents the query optimization algorithm and ways of improving its efficiency. Section 6.5 applies the cost analysis and optimization methods to a language with program abstractions. Section 6.6 describes a specification language for describing the concrete layer. Finally, Section 6.7 concludes this chapter.

6.1 The Transformation Specification Language

In this section we describe a language for expressing our type transformations. It is designed in a way that supports flexibility and modularity.

We introduce a new type constructor **map** with no operations and no instances. It has the form:

```
map t1 into t2 via r and c
```

It defines a type transformation from type t1 to type t2 with representation function r and coding function c. A **map** type has no instances and therefore we cannot define operations or variables of this type. The only place it is used is for constructing other more complex mappings.

A simple example of a mapping is the identity mapping id:

```
id(alpha) = map alpha into alpha via [x] → x and [x] → x;
```

Type name id is defined to represent the mapping from any type *alpha* to itself whose representation and coding functions are the identity functions.

The following type definition defines type boolmap to be the type transformation from booleans to [0,1]:

```
boolmap = map boolean into range[0,1]
          via tc_int( [] → true, [?,?] → false)
          and tc_boolean( [] → 0, [] → 1);
```

Bounded parameterization (defined in Section 3.1.8) can be used for creating polymorphic type transformation. For example, a polymorphic transformation of the type pair, defined as:

```
pair(alpha,beta) = struct pairup ( first: alpha, second: beta );
```

is the following:

```

pairmap( m1(a1,b1)[r1:[b1]→a1,c1:[a1]→b1] = map a1 into b1 via r1 and c1,
        m2(a2,b2)[r2:[b2]→a2,c2:[a2]→b2] = map a2 into b2 via r2 and c2 ) =
  map pair(a1,a2) into pair(b1,b2)
  via tc_pair([x,y]→pairup(r1(x),r2(y)))
  and tc_pair([x,y]→pairup(c1(x),c2(y)));

```

That is, pairmap has two bounded type parameters: m1 and m2 that are mappings. For example, pairmap(id(int),id(int)) is a valid instantiation of pairmap that binds a1, b1, a2 and b2 to int and r1, c1, r2 and c2 to [x]→x.

Another example is mapping sets into ordered lists (Section 3.3). Here there is an extra parameter before to be used in the restrictions:

```

keyed( m(a,b)[rb:[b]→a,ca:[a]→b] = map a into b via rb and ca )
  [ before: [b,b]→boolean ] =
  map set(a)
  into list(b) where(x) x=tc_list([],nil,[a,?,r]→ordered_cons(a,r,before))(x)
  via tc_list([],emptyset,[a,?,r]→insert(rb(a),r))
  and tc_set([],nil,[a,?,r]→ordered_cons(ca(a),r,before));

```

where ordered_cons is an order independent function defined as:

```

function(alpha) ordered_cons
  ( a: alpha, r: list(alpha), before: [alpha,alpha]→boolean ) : list(alpha);
tc_list([],cons(a,nil),
  [b,l,s]→if before(a,b)
    then if before(b,a) then cons(b,l) else cons(a,cons(b,l))
    else cons(b,s))(r);

```

For example, keyed(id(person))[x,y]→x.ssn≤y.ssn] maps set(person) into list(person) ordered by the social security numbers.

The `tc.list` in `ordered_cons` traverses `r`, an accumulative result variable. Another version of the coding function of the mapping keyed which is a uniform traversal combinator is:

```
[x] → tc_set( [] → nil,
              [?, l, ?] → tc_set( [] → ?, [a, ?, r] → if tc_set( [] → zero, [?, ?, i] → succ(i))(l) =
                                tc_set( [] → zero,
                                          [b, ?, s] → if before(ca(a), ca(b))
                                                         then succ(s)
                                                         else s)(x)
                                then ca(a)
                                else r)(x))(x)
```

The following creates a mapping from a set to a pair of lists sorted by the functions `f1` and `f2`:

```
double_keyed(m(a, b)[rb: [b] → a, ca: [a] → b] = map a into b via rb and ca)
  [ f1: [b, b] → boolean, f2: [b, b] → boolean ] =
  apairmap(keyed(m(a, b)[rb, ca])[f1], keyed(m(a, b)[rb, ca])[f2]);
```

where `apairmap` is a mapping from any type `a` to a pair `pair(b1, b2)`:

```
apairmap( m1(a, b1)[r1: [b1] → a, c1: [a] → b1] = map a into b1 via r1 and c1,
          m2(a, b2)[r2: [b2] → a, c2: [a] → b2] = map a into b2 via r2 and c2 ) =
  map a into pair(b1, b2) where(x) tc_pair( [x, y] → r1(x)=r2(y)
  via tc_pair( [x, ?] → r1(x)
  and [x] → pairup(c1(x), c2(x));
```

For example, the following creates a mapping from a set of persons to a pair of lists. The first list is sorted by the person's name and the second by ssn:

```
double_keyed(id(person))[ [x, y] → x.name ≤ y.name, [x, y] → x.ssn ≤ y.ssn ]
```

The following transforms a set into a hash table whose entries are sets consisting of elements with the same hash value:

```

hashed( m(a,b)[rb:[b]→a,ca:[a]→b] = map a into b via rb and ca )
  [ size: int, hash: [b]→range[0,size-1] ] =
  map set(a)
  into vector(set(b))
  where(x) tc_vector( []→true,
                    [i,v,r]→tc_set( []→true,
                                     [a,?,s]→i=hash(a) and s)(v) and r)(x)
  via tc_vector( []→emptyset, [?,v,r]→tc_set( []→r, [a,?,s]→insert(rb(a),s))(v))
  and tc_set( []→newvector(size,emptyset), [a,?,r]→vector_insert(ca(a),r,hash));

```

where `vector_insert(a,r,hash)` is the order independent function:

```
tc_vector( []→r, [i,v,s]→if i=hash(a) then update(i,insert(a,v),s) else s)(r)
```

Suppose now that we want to map the result of this mapping `vector(set(b))` into `vector(set_impl(b))`, where `set_impl` is some implementation of sets. The following type transformation takes a mapping `m1` from type `a` to type `vector(c)` and a mapping `m2` from `c` to `d` and returns a new mapping from `a` to `vector(d)`:

```

comp_vec( m1(a,b=vector(c))[r1:[b]→a,c1:[a]→b] = map a into b via r1 and c1,
          m2(c,d)[r2:[d]→c,c2:[c]→d] = map c into d via r2 and c2 ) =
  map a into vector(d)
  via [x]→r1(tc_vector( []→x, [i,v,r]→update(i,r2(v),r))(x))
  and [x]→tc_vector( []→x, [i,v,r]→update(i,c2(v),r))(c1(x));

```

For example, the following is a transformation of a set into a hash table of length `size` and hash function `hash` whose entries are lists ordered by some function `before`:

```

hash_table( m(a,b)[rb:[b]→a,ca:[a]→b] = map a into b via rb and ca )
    [ size: int, hash: [b]→range[0,size], before: [b,b]→boolean ] =
    comp_vec(hash(m(a,b)[rb,ca])[size,hash],keyed(m(a,b)[rb,ca])[before]);

```

For example, `hash_table(id(person))[100, [x]→x.ssn mod 100, [x,y]→x.name≤y.name]`, where `a mod n = a-(a div n)*n` and `a div n` is:

```
tc_int( []→0, [i,r]→if i*n<a then i else r)(a)
```

6.2 Database Implementation

We assume that there is only one persistent object in a program, namely the database, which is of a non-polymorphic canonical type. Typically, the database is a big tuple whose components are the values we want to persist. For example, in a relational system the database object is a tuple that consists of relations. Database queries are functions from the database type to any canonical type. A database transaction can be simulated by a function that accepts the current database object as input, along with some extra parameters, and return a new database object which is different from the original object if the transaction contains destructive updates. We will explore these language issues in Appendix A.

The database mapping is defined by the special ADABTPL language primitive database that has the form: `database name : type-expression`, where `name` is the name of the database mapping and `type-expression` is a type expression that constructs a mapping. There must be only one database definition in a program. For example, this type expression can be a `map` construction:

```
database name : map db into DB via r and c;
```

where `db` is the type of the abstract database object and `DB` the type of the concrete database object.

Consider, for example, the database type `db` that includes a part-subpart hierarchy and a set of part orders:

```

part = union ( basep: struct base ( name: string,
                                   cost: int ),
               compositep: struct composite ( name: string,
                                               cost: int,
                                               subparts: set(part) ) );

order = struct make_order ( customer: person,
                             ordered: set(part),
                             quantity: int );

db = struct make_db ( parts: set(part),
                     customers: set(person),
                     orders: set(order) );

```

For example, the following is a query in combinator form that computes the number of all ordered parts (counting all subparts of composite parts):

```
tc_db( [?, ?, os] → tc_set( [] → 0, [o, s] → s + tc_order( [?, ps, q] → sparts(ps, q))(o))(os))(db)
```

where `sparts(ps, q)` is:

```

tc_set( [] → 0,
        [p, u] → u + tc_part( [?, ?] → q,
                               [?, ?, si] → q + tc_set( [] → 0,
                                                         [i, r] → i + r)
                               (si))
        (p))(ps)

```

One possible implementation for the recursive type `part` is `alist`, a tree whose nodes have arbitrary number of children:

```

alist(alpha, beta) =
  union ( leafp: struct leaf ( info: beta );
         brunchp: struct brunch ( info: alpha,
                                   children: list(alist(alpha, beta)) ) );

```


More specifically, type `part` is mapped into:

```
ipart = alist(pair(string,int),pair(string,int));
```

The detailed mapping is the following:

```
partmap(m[r: [list(ipart)] → set(ipart), c: [set(ipart)] → list(ipart)] =
    map set(ipart) into list(ipart) via r and c) =
    map part into ipart
    via tc_alist( [v] → tc_pair( [n,i] → base(n,i))(v),
        [v,?,s] → tc_pair( [n,i] → composite(n,i,r(s)))(v))
    and tc_part( [n,i] → leaf(pairup(n,i)),
        [n,i,?,s] → brunch(pairup(n,i),c(s)));
```

If we implement the set of parts as a list of parts ordered by name then:

```
pmap = partmap(keyed(id(ipart))([x,y] → name(x) ≤ name(y)));
```

where `name(x)` is `tc_alist([v] → tc_pair([n,?] → n)(v), [v,?,?] → tc_pair([n,?] → n)(v))(x)`.

The mapping of order is parameterized using the mapping of `set(part)`:

```
ordermap(m[r: [a] → set(part), c: [set(part)] → a] =
    map set(part) into a via r and c) =
    map order into pair(pair(person,int),a)
    via tc_pair( [p,s] → tc_pair( [c,q] → make_order(c,r(s),q))(p))
    and tc_order( [c,s,q] → pairup(pairup(c,q),c(s)));
```

Similarly, we parameterize the mapping of `db` using the mappings of `set(part)`, `set(person)`, and `set(order)`:

```
dbmap(m1[r1: [a] → set(part), c1: [set(part)] → a] =
    map set(part) into a via r1 and c1,
    m2[r2: [b] → set(person), c2: [set(person)] → b] =
```

```

    map set(person) into b via r2 and c2,
    m3[r3: [g] → set(order), c3: [set(order)] → g] =
    map set(order) into g via r3 and c3) =
map db into pair(pair(a, b), g)
via tc_pair([v, o] → tc_pair([p, c] → make_db(r1(p), r2(c), r3(o)))(v))
and tc_db([p, c, o] → pairup(pairup(c1(p), c2(c)), c3(o)));

```

One possible implementation of the database db is the following:

database dbmapping :

```

dbmap(keyed(pmap)[[x, y] → name(x) ≤ name(y)],
      hash_table(id(person))[100, [x] → x.ssn mod 100, [x, y] → x.ssn ≤ y.ssn],
      keyed(ordermap(keyed(pmap)[[x, y] → name(x) ≤ name(y)]))
      [[x, y] → x.customer.ssn ≤ y.customer.ssn]);

```

6.3 Cost Model

In Section 5.7 we presented a method for translating an abstract operation into an expression that contains concrete primitives only. In some cases, especially for data intensive applications, this default translation is not efficient enough. There may be programs equivalent to this default translation, that is, with the same functionality, but with lower execution cost. Finding the best of these programs is the task of a query optimizer. Effective operation optimization requires comparing alternative methods of execution. We have already seen how redundant information in the form of integrity constraints attached to types and the extra parameters in the general coding function offer a number of translations to choose from. We will present a method for searching this space of alternatives in Section 6.4.

The execution costs of two programs can be compared by using a cost function, which is a mapping from programs to costs (typically numeric values). Statistics

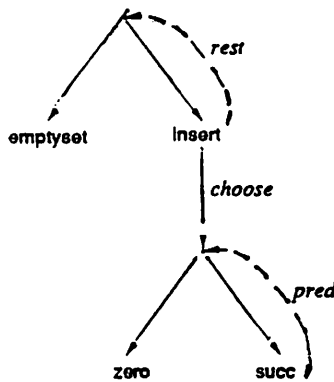


Figure 6.1 The constructor tree of `set(int)`

about the persistent data can be derived at compile-time to aid the estimation of valid costs. Exact costs can only be computed after the actual operation is performed and all consumed resources that constitute the cost function are measured.

In this section we present a flexible cost model well-suited to programs expressed as uniform traversal combinators. It is based on a non-standard abstract interpretation of programs, aided by statistical information. The following cost analysis applies to both the abstract and concrete layers but we intend to use it for the concrete layer only.

We use a special tree, called the **constructor tree**, as a comprehensive image of the actual data stored in the database. Suppose that the database type `db` is a set of integers. The graph in Figure 6.1 captures all possible expressions of type `db` that involve constructors only. For example, `insert(succ(succ(zero)),insert(zero,emptyset))` is a valid traversal of the graph in Figure 6.1. If we ignore the recursive type references, that is, if we ignore the `rest` component of the set and the `pred` component of the `succ` constructor (the dotted lines in Figure 6.1) then the resulting tree is the constructor tree. That is, `db` has two constructors `emptyset` and `insert`, and the `choose` component of the `insert` has two constructors `zero` and `succ`. More formally:

Definition 6.1: (Constructor Tree) Let $T(\bar{\alpha})$ be a canonical type with n constructors $C_i : T_{i,1}(\bar{\alpha}) \times \dots \times T_{i,k_i}(\bar{\alpha})$, where each C_i is associated with the selectors $a_{i,1}, \dots, a_{i,k_i}$. The constructor tree $CT(T(\bar{\alpha}))$ of a type $T(\bar{\alpha})$ is a tree whose root has children nodes C_i and each node C_i has children that are all the trees $CT(T_{i,j}(\bar{\alpha}))$ such that $T_{i,j}(\bar{\alpha}) \neq T(\bar{\alpha})$. The tree edges from C_i to its children are labeled by the selector names $a_{i,j}$.

We call the constructor tree associated with a database type the **database constructor tree**.

For example, the database constructor tree $CT(db)$ in Figure 6.2 represents the part-subpart database schema (defined in Section 6.2). The outgoing arrows from each node labeled by a constructor name are labeled by selector names (not shown in this figure). Dotted lines represent selectors of recursive types and therefore they are ignored in this figure. Tree nodes associated with the integer and string types are not shown in this figure (we put boxes for integers and circles for strings instead).

Definition 6.2: (Constructor Path) A constructor path of the constructor tree $CT(T(\bar{\alpha}))$ is either the empty path $[]$ or one of $[C_i]$ or $P \circ [a_{i,j}, C_i]$, where C_i a constructor of $T(\alpha)$, $a_{i,j}$ a selector of C_i of type $T_{i,j}(\bar{\alpha})$, and P is a constructor path of $CT(T_{i,j}(\bar{\alpha}))$ (operator 'o' concatenates constructor paths).

We call a constructor path associated with a database type a **database constructor path**. For example, the following is a constructor path in the part-subpart database:

[succ,ssn,make_person,customer,make_order,choose,insert,orders,make_db]

Note that there are finite number of constructor paths because there are no recursive references in the constructor tree.

Definition 6.3: (Schema Cost) The schema cost of a database constructor tree $CT(T(\alpha))$ is a mapping $\Psi(P)$ from any database constructor path P of $CT(T(\alpha))$ to a cost.

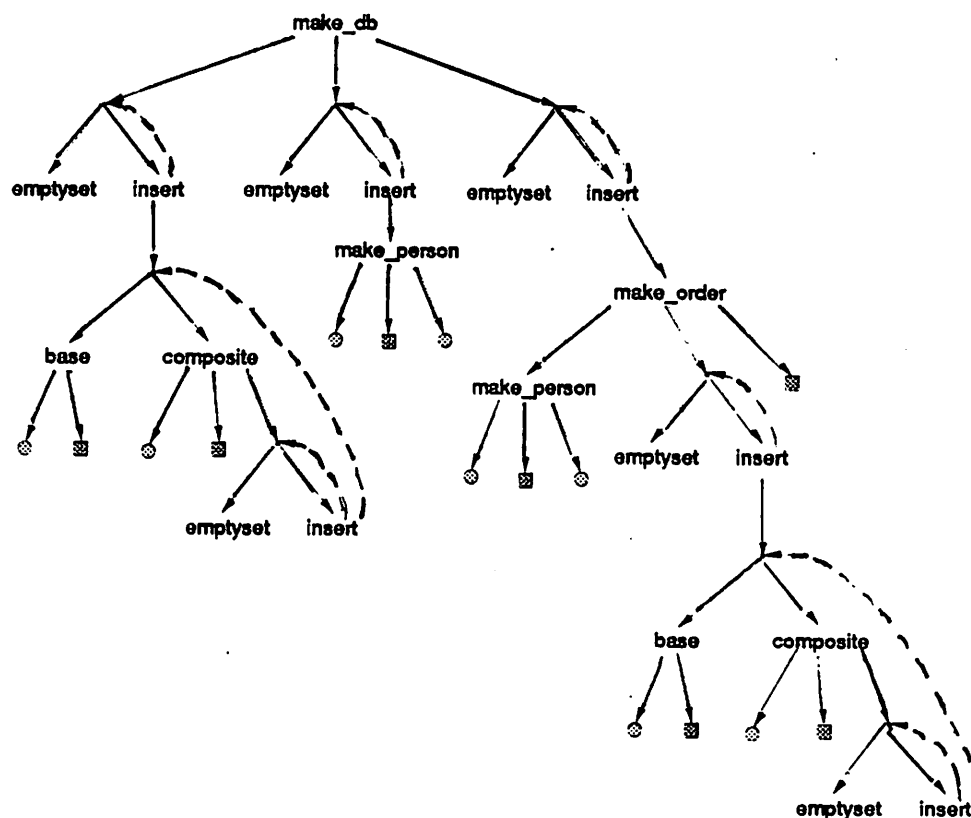


Figure 6.2 The constructor tree of the part-subpart database

A cost can be of any type, provided that there is a total function for comparing costs. Costs are usually numeric values whose comparison function is $<$. The best program among a set of programs is the one with the minimum cost.

As an example of a numerical cost model we could assign **average constructor numbers** to any node in the constructor tree labeled by some constructor C_i . This number is equal to one for the root (we have only one database) and for any other constructor node C_i , a child of some node C_j , it is the average number of objects in the database constructed by C_i that are children of any object constructed by C_j .

For example, suppose that the part-subpart database has 100 parts. Each part tree has an average of 4 base parts and 2 composite parts and each composite part has an average of 3 subparts. Suppose also that the database has 20 customers and

60 orders. Each order has one customer and an average of 3 ordered parts. Each such part has the same average numbers with the other parts.

The average constructor numbers assigned to a database scheme can be easily computed during compile time by executing queries that counts all objects in the database that belong to a specific type and then normalize these numbers by taking averages. The programs that compute these measures can be generated automatically by having the compiler check the database type details. The database implementor can intervene in this process by providing explicit programs for the computation of schema costs. More specifically, we introduce a new type constructor cost:

$$T(\bar{a}) \text{ cost}(\bar{x}) c(\bar{x})$$

This defines the cost of type $T(\bar{a})$ to be $c(\bar{x})$, where x_i is the cost of a_i . That is, c is a function from $C^n \rightarrow (T(\bar{a}) \rightarrow C)$, where $n = |\bar{a}|$ and C is the type of the cost.

The generated queries that compute the schema cost can be compiled by the same compiler that optimizes regular run-time queries. That way if we have a view of the database that contains these statistics then retrieving these numbers consists simply of fetching these statistics from the views. Alternatively, these statistics can be stored in a special database that is updated periodically.

Any non-accumulative result variable that appears in a uniform traversal combinator can be associated with one node in the constructor tree. Accumulative result variables do not correspond to any part of the database as they are the results of traversing parts of the database. Luckily the only place that we need to assign constructor paths for computing valid costs is when traversing variables. But these variables cannot be accumulative result variables and therefore they can always be assigned constructor paths. For example, the following query that computes the sum of all integers in the database `set(int)`:

```
tc_set( [] → 0, [a,l,s] → tc_int( [] → s, [i,r] → succ(r))(a))(x)
```

- 1) $\text{cost}(z, \mathcal{G}) = \phi_1(\Psi(\mathcal{G}(z)))$
- 2) $\text{cost}(C(h_1(\bar{x}), \dots, h_n(\bar{x})), \mathcal{G}) = \phi_2(\text{list}(\text{cost}(h_1(\bar{x}), \mathcal{G}), \dots, \text{cost}(h_n(\bar{x}), \mathcal{G})))$
- 3) $\text{cost}(\mathcal{H}_T(f_1, \dots, f_n)(z), \mathcal{G})$
 $= \begin{cases} \phi_3(\text{list}(\dots, \text{pairup}(\Psi([C_i] \circ \mathcal{G}(z)), \text{cost}(f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i), \mathcal{G}_i)), \dots)) \\ \text{where } \mathcal{G}_i = \mathcal{G}[\dots, x_j^i \rightarrow [a_{i,j}, C_i] \circ \mathcal{G}(z), \dots, y_j^i \rightarrow \mathcal{G}(z), \dots] \end{cases}$
- 4) $\text{cost}(\mathcal{E}_T(f, h, g), \mathcal{G}) = \phi_4(\text{cost}(f, \mathcal{G}), \text{cost}(h, \mathcal{G}), \text{cost}(g, \mathcal{G}))$

Figure 6.3 Definition of the cost function $\text{cost} = \mathcal{H}_C(\phi_1, \phi_2, \phi_3, \phi_4)$

has the following mapping from local variables to constructor paths:

$x \rightarrow []$, $a \rightarrow [\text{choose}, \text{insert}]$, $l \rightarrow []$, $i \rightarrow [\text{choose}, \text{insert}]$

Definition 6.4: (Cost of a Uniform Traversal Combinator) The cost of a uniform traversal combinator $h(x)$ that manipulates the database object x is $\text{cost}(h(x), \mathcal{G})$, where \mathcal{G} is a mapping from variable names to constructor paths and function cost is the traversal combinator $\text{cost} = \mathcal{H}_C(\phi_1, \phi_2, \phi_3, \phi_4)$, defined in Figure 6.3.

C_i in Figure 6.3 is the i th constructor of T and $a_{i,1}, \dots, a_{i,k_i}$ are the associated selectors, $\mathcal{G}[x \rightarrow e]$ extends \mathcal{G} with the mapping from variable x to the constructor path e , and $\mathcal{G}(z)$ returns the constructor path associated with variable z . The first rule says that the cost of accessing a variable z associated with a database constructor path $\mathcal{G}(z)$ depends only on the cost $\Psi(\mathcal{G}(z))$. The cost of a construction depends on the list of costs of the constructor components (second rule). The third rule says that the cost of a traversal depends on the list of the costs of the traversal components f_i paired with the costs of the paths $[C_i] \circ \mathcal{G}(z)$ (because f_i corresponds to the constructor C_i only). When computing the cost of f_i the mapping \mathcal{G} is expanded to include mappings from the local variables of f_i to database constructor paths (note that there are no paths assigned to the accumulative result variables z_j^i).

The last rule computes the cost of an equality combinator. Note that this model can also compute the cost of constant expressions (that is, constructions that do not refer to any part of the database).

A cost function is valid if it is monotonic, that is, if the cost of a combinator is greater than the cost of its component functions.

Definition 6.5: (Monotonic Cost Function)

A cost function $\text{cost} = \mathcal{H}_C(\phi_1, \phi_2, \phi_3, \phi_4)$ is monotonic if:

$$\begin{aligned} \forall r : \quad & \phi_2(r) > \max\{c_i/c_i \in r\} \\ \forall r : \quad & \phi_3(r) > \max\{c_i/\text{pairup}(s_i, c_i) \in r\} \\ \forall c_1 \forall c_2 \forall c_3 : \quad & \phi_4(c_1, c_2, c_3) > \max\{c_1, c_2, c_3\} \end{aligned}$$

If a cost function is monotonic then the composition algorithm (Algorithm 4.1) is guaranteed to yield better programs (if programs are compared using this cost function). Furthermore, algorithms for optimizing programs, such as the best-first search algorithm that will be described in Section 6.4, can quickly prune out unfruitful paths without worrying that these may turn out to be optimal later.

For example, one monotonic cost function that returns numeric costs is the following:

$$\begin{aligned} \text{cost} = \text{tc_cost}([\?] \rightarrow 0, [r] \rightarrow \text{tc_list}([\] \rightarrow 1, [a,?,s] \rightarrow s+a)(r), \\ [r] \rightarrow \text{tc_list}([\] \rightarrow 0, [a,?,s] \rightarrow s+\text{tc_pair}([c,q] \rightarrow c*q)(a))(r), \\ [f,g,h] \rightarrow f+g+h) \end{aligned}$$

In that case, Definition 6.4 is equivalent to the following:

$$\begin{aligned} \text{cost}(z, \mathcal{G}) &= 0 \\ \text{cost}(C(h_1(\bar{x}), \dots, h_s(\bar{x})), \mathcal{G}) &= 1 + \sum_{i=1}^s \text{cost}(h_i(\bar{x}), \mathcal{G}) \\ \text{cost}(\mathcal{H}_T(f_1, \dots, f_n)(z), \mathcal{G}) &= \begin{cases} \sum_{i=1}^n (\Psi([C_i] \circ \mathcal{G}(z)) \times \text{cost}(f_i(\bar{x}_i, \bar{y}_i, \bar{z}_i), \mathcal{G}_i)) \\ \text{where } \mathcal{G}_i = \mathcal{G}[\dots, x_j^i \rightarrow [a_{i,j}, C_i] \circ \mathcal{G}(z), \dots, y_j^i \rightarrow \mathcal{G}(z), \dots] \end{cases} \\ \text{cost}(\mathcal{E}_T(f, h, g), \mathcal{G}) &= \text{cost}(f, \mathcal{G}) + \text{cost}(h, \mathcal{G}) + \text{cost}(g, \mathcal{G}) \end{aligned}$$

Intuitively, this estimates the number of constructed values visited by a program. For example, for the set of integers database *db* that has 100 set elements the cost of the query $tc_set([\] \rightarrow nil, [a,s] \rightarrow cons(a,s))(db)$ is $1*1+100*1=101$.

For the part-subpart database, whose average constructor numbers described on page 130, the cost of the query that computes all ordered parts is:

$$1*(1*1+60*(1+1*(1*1+3*(1+4*1+2*(1+3*1))))))=2461.$$

Let *x* be a set whose cardinality is greater than the cardinality of set *y*. Function call $union(x,y)$ can either be expressed as $tc_set([\] \rightarrow y, [a,s] \rightarrow insert(a,s))(x)$ or $tc_set([\] \rightarrow x, [a,s] \rightarrow insert(a,s))(y)$. The cost of the first expression is the cardinality of *x* plus one while the cost of the second is the cardinality of *y* plus one. Therefore, the second expression is more efficient than the first. Section 6.4 presents a complete algorithm for generating equivalent expressions and comparing their costs.

The cost model described previously is not very flexible for database applications. It will be preferable to allow the database implementor to assign individual cost estimation functions to some carefully selected operations to affect the cost of other operations that use them. This is very helpful when the database implementor models the concrete layer using abstractions in our constructor algebra that have the same functionality as the concrete primitives but their comparable quality and efficiency, as presented by the cost model described earlier, is not reflected in the same way in the concrete layer.

For example, suppose that we model a B-tree as an ordered list:

$btree(alpha, beta)[f: [alpha, alpha] \rightarrow boolean] = list(pair(alpha, beta))$

where(*x*) $x = tc_list([\] \rightarrow nil,$

$[p, ?, r] \rightarrow ordered_cons(p, r, [y, z] \rightarrow f(tc_pair([a, ?] \rightarrow a)(y),$

$tc_pair([a, ?] \rightarrow a)(z))))(x);$

where *alpha* is the key type and *beta* the information. Let *find* be the operation that finds an element in a *btree*. If we model *find* in terms of the *tc_list* then its cost is

proportional to the size of the btree (using the cost model with average constructor numbers). This is not valid because for real B-trees the cost is logarithmic. This could generate translations that are not optimal. We can extend ADABTPL to include the language primitive `cost` that declares the cost of a function explicitly:

```
function(alpha,beta) find ( x: btree(alpha,beta), val: alpha, def: beta ) : beta;
tc_list( [] → def, [p,?,r] → tc_pair( [a,b] → if a=val then b else r )(p))(x);
cost log2(size(x));
```

This declares the cost of `find` to be equal to the logarithm in base two of the size of the btree `x`. This cost function is computed once for each btree variable before the program translation, as is done for the general cost functions.

6.4 The Optimization Algorithm

Section 5.5 presented a method of generating all uniform traversal combinators equivalent to a specific one by trying various program patterns and using the equality algorithm to abort the incompatible ones. This section unifies this algorithm with the cost estimation program presented in Section 6.3 to extract only one from all these alternative programs: the one with the minimum cost. As in Section 6.1, it is assumed here that all type transformations are defined in terms of both representation and coding function as this always produces solutions expressed in terms of concrete primitives only.

Let f be an abstract operation of type $T_1 \times \dots \times T_n \rightarrow T$ where each T_i may be a restricted type. Then its implementation F is derived from Equation 3.4 and Theorem 5.2:

$$\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow F(x_1, \dots, x_n) = c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n))) \quad (6.1)$$

If f , c_0 and all r_i are uniform traversal combinators then all possible solutions that satisfy the Equation 6.1 are derived from the following algorithm:

Algorithm 6.1: (Optimal Function Implementation) Let $f(x_1, \dots, x_n)$ be an abstract function, \mathcal{G} a mapping from the variables x_i to database path signatures, and F a concrete implementation of f that satisfies Equation 6.1 and has minimum cost $\text{cost}(F, \mathcal{G})$. Then F is called the optimal implementation of f with respect to \mathcal{G} if it is computed by:

$$F = \text{best}(\{g_1, \dots, g_r\}, \mathcal{G}) : \\ \mathcal{E}(\Phi(\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow F(x_1, \dots, x_n) = c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n))), \\ [], []), \text{true}, \rho, \beta) \vdash \{F = g_1, \dots, F = g_r\}$$

where $\text{best}(\{\lambda \bar{x}.g_1(\bar{x}), \dots, \lambda \bar{x}.g_r(\bar{x})\}, \mathcal{G})$ is some $g_k, k \in [1, r]$ such that

$$\text{cost}(\lambda \bar{x}.g_k(\bar{x}), \mathcal{G}) = \min\{\text{cost}(\lambda \bar{x}.g_1(\bar{x}), \mathcal{G}), \dots, \text{cost}(\lambda \bar{x}.g_r(\bar{x}), \mathcal{G})\}$$

That is, we synthesize all possible uniform traversal combinators g_i that satisfy Equation 6.1 and then we select the best.

For example, let `alist` be a list extended with the redundant information of its length:

```
alist(alpha) = struct ma ( info: list(alpha), len: int )
                where(x) tc_alist([i,l] → length(i)=l)(x);
```

Let also $f(x,y)$ be an operation on x and y of type `alist` defined as:

```
f(x,y) = length(x.info)+length(y.info)
```

Then Algorithm 6.1 gives (here r_i and c_0 are identity mappings):

```
tc_alist([i,l] → length(i)=l)(x) and tc_alist([i,l] → length(i)=l)(y)
⇒ f(x,y) = length(tc_alist([i,?] → i)(x))+length(tc_alist([i,?] → i)(y))
```

which is expressed as:

```

tc_alist([i,l] → if length(i)=l
          then tc_alist([i,l] → if length(i)=l
                          then fc2([z] → z=length(tc_alist([i,?] → i)(x))
                                      +length(tc_alist([i,?] → i)(y)),
                                      [] → x, [] → y)(f)
                          else true)(y)
          else true)(x)

```

This yields the following solutions for $f(x,y)$:

$\text{length}(\text{tc_alist}([i,?] \rightarrow i)(x)) + \text{length}(\text{tc_alist}([i,?] \rightarrow i)(y))$

$\text{tc_alist}([?,l] \rightarrow l)(x) + \text{length}(\text{tc_alist}([i,?] \rightarrow i)(y))$

$\text{length}(\text{tc_alist}([i,?] \rightarrow i)(x)) + \text{tc_alist}([?,l] \rightarrow l)(y)$

$\text{tc_alist}([?,l] \rightarrow l)(x) + \text{tc_alist}([?,l] \rightarrow l)(y)$

The last solution is the cheapest as it does not require traversing the lists $x.\text{info}$ and $y.\text{info}$.

Algorithm 6.1 is inefficient because there may be a large number of possible solutions that satisfy Equation 6.1. A better method is to incorporate the cost estimation algorithm in Figure 6.3 with Algorithm 5.2. This is achieved by having the mapping \mathcal{G} from variable names to database constructor paths passed as an extra parameter to the equality tester \mathcal{E} .

Algorithm 6.2: (Optimization Algorithm) The optimal implementation F of a function f that satisfies Equation 6.1 is:

$$\mathcal{E}(\Phi(\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow F(x_1, \dots, x_n) = c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n))), [], []), \text{true}, \rho, \beta, \mathcal{G}) \vdash F = e$$

where \mathcal{G} maps each x_i into a database constructor path and the Algorithm 6.1 has been extended to include the rules shown in Figure 6.4. Note that this algorithm always yields a solution because there is at least one F that satisfies Equation 6.1, namely $c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n)))$.

$$\begin{aligned}
f = \text{best}(\{ & \lambda \bar{x}. x_k / \mathcal{E}(\Phi(c(r_k), [], []), g, \rho, \beta) \neq \text{fail}\} \\
& \cup \{ \lambda \bar{x}. C_k(\dots, e_i(\bar{x}), \dots) / C_k \text{ is a constructor of } T \text{ and} \\
& \quad \mathcal{E}(\Phi(c(C_k(\dots, \mathcal{F}_{T_i}(\lambda x.x, r_1, \dots, r_n)(f_i), \dots)), [], []), g, \rho, \beta, \mathcal{G}) \vdash f_i = e_i \} \\
& \cup \{ \lambda \bar{x}. \mathcal{H}_{T_k}(\dots, e_i, \dots)(x_k) / x_k : T_k \text{ is non-accumulative from } \bar{x} \text{ and} \\
& \quad \mathcal{E}(\Phi(c(\mathcal{H}_{T_k}(\dots, \lambda \bar{z}_i. \mathcal{F}_{T_i}(\lambda x.x, \bar{z}_i)(f_i), \dots)(r_k)), [], []), g, \rho, \beta, \mathcal{G}_i) \vdash f_i = e_i, \\
& \quad \text{where } \mathcal{G}_i = \mathcal{G}[\dots, x_j^i \rightarrow [a_{i,j}, C_i] \circ \mathcal{G}(z), \dots, y_j^i \rightarrow \mathcal{G}(z), \dots] \} \\
& \cup \{ \lambda \bar{x}. \mathcal{E}_T(e_1, e_2, e_3) / \\
& \quad \mathcal{E}(\Phi(c(\mathcal{E}_T(\mathcal{F}_T(\lambda x.x, r_1, \dots, r_n)(f_1), \\
& \quad \quad \mathcal{F}_T(\lambda x.x, r_1, \dots, r_n)(f_2), \\
& \quad \quad \mathcal{F}_{T'}(\lambda x.x, \lambda x.x)(f_3))), [], []), g, \rho, \beta, \mathcal{G}) \vdash f_i = e_i \}, \mathcal{G})
\end{aligned}$$

Figure 6.4 Optimization of $\mathcal{F}_T(c, r_1, \dots, r_n)(f) = g$

The algorithm in Figure 6.4 is a depth-first search that always selects the program with the minimum cost. Therefore, in this case we do not have to return a list of bindings because there is always only one solution for f .

Let us further analyze the way Algorithm 6.2 works by concentrating on traversals only. We want to find the best $f(\bar{x})$ equal to a known traversal combinator g . We assume that f is a traversal of the form $\mathcal{H}_{T_k}(\dots, f_i, \dots)(x_k)$, where x_k of type T_k is one of \bar{x} . Finding the traversal components f_i is proving this equality as a high order theorem with f_i appearing free as a second-order function. Normally, if we did not prune out the bad solutions (like we did in Algorithm 5.2) then we could have $k_i > 0$ solutions for each f_i and therefore we have $\prod_i k_i$ solutions for $f = \mathcal{H}_{T_k}(\dots, f_i, \dots)(x_k)$ (or zero if any of the equalities fail). Now that we return only one solution each time, we have only one solution for each f_i and thus we have only one solution for $f = \mathcal{H}_{T_k}(\dots, f_i, \dots)(x_k)$. Therefore, the search space, which is a tree whose nodes are traversals and children are the traversal components, is fully searched.

Here, as in Algorithm 5.2, we may have an infinite set of solutions. In this case we want to find a solution for F that has better cost than the default program $c_0(e_1, \dots, e_m)(f(r_1(x_1), \dots, r_n(x_n)))$. Therefore, we can stop the depth-first search algorithm whenever we reach a depth c times proportional to the size of this default program (for example, c could be 2). This is based on the assumption that the optimal solution is unlikely to be too much longer than the default program. This part of the search space is guaranteed to include at least one solution (which may not be optimal): namely the default solution. There are many ways of improving Algorithm 6.2. One way is to assign a cost to each uniform traversal combinator expression so that there is no need of computing the cost of a program multiple times.

A more efficient algorithm is using a best-first search strategy in which the most promising node is expanded at each decision point:

Algorithm 6.3: (Best-first Search Optimization Algorithm) The algorithm in Figure 6.5 returns the optimal solution for $\bigwedge_i I\{T_i\} \Rightarrow F(x_1, \dots, x_n) = g$ using a best-first search method. Here we use the notation $\langle \lambda \bar{x}.f(\bar{x}) \rangle$ as a new type of node in the expression tree of a uniform traversal combinator. This is a placeholder for a combinator that will be synthesized by this algorithm. Note that f and all \bar{x} in $\langle \lambda \bar{x}.f(\bar{x}) \rangle$ are variable names. The cost of the $\langle \lambda \bar{x}.f(\bar{x}) \rangle$ node is zero, as there is no information available about f . We say that a combinator containing $\langle \lambda \bar{x}.f(\bar{x}) \rangle$ is 'expanded' if the $\langle \lambda \bar{x}.f(\bar{x}) \rangle$ node is replaced by a combinator $\lambda \bar{x}.h(\bar{x})$. Set S holds the partially expanded combinators that contain at least one node of the form $\langle \lambda \bar{x}.f(\bar{x}) \rangle$. We start the algorithm with S containing the theorem. At each step, the algorithm selects a partially expanded combinator $e(\bar{z})$ from S with the best cost, expands one of its nodes $\langle \dots \rangle$ into each of the compatible patterns of programs, and repeats the same process. If a fully expanded expression is derived, it is compared with

```

S ← { $\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow \langle \lambda x_1 \dots \lambda x_n. F(x_1, \dots, x_n) \rangle$ };
opt_cost ← maximum_cost;
while S ≠  $\emptyset$  do
{ find  $e(\bar{z}) \in S$  with minimum cost( $e(\bar{z}), \mathcal{G}$ );
  S ← S -  $e(\bar{z})$ ;
  find one node  $\langle \lambda \bar{x}. f(\bar{x}) \rangle$  in the expression tree  $e(\bar{z})$ ;
  S ← S + { $e_1(\bar{z}), \dots, e_k(\bar{z})$ };   where  $e_i(\bar{z})$  is equal to  $e(\bar{z})$  but with node
                                      $\langle \lambda \bar{x}. f(\bar{x}) \rangle$  replaced by one of:

    1)  $\lambda \bar{x}. x_k$ 
    2)  $\lambda \bar{x}. C_k(\dots, \langle \lambda \bar{x}. f_i(\bar{x}) \rangle, \dots)$ 
    3)  $\lambda \bar{x}. \mathcal{H}_{T_k}(\dots, \langle \lambda \bar{z}_i. f_i(\bar{z}_i) \rangle, \dots)(x_k)$ 
    4)  $\lambda \bar{x}. \mathcal{E}_T(\langle \lambda \bar{x}. f_1(\bar{x}) \rangle, \langle \lambda \bar{x}. f_2(\bar{x}) \rangle, \langle \lambda y. f_3(y) \rangle)$ 
  for all  $e_i(\bar{z})$  with no  $\langle \dots \rangle$  nodes do
  { S ← S -  $e_i(\bar{z})$ ;
    if  $\mathcal{E}(e_i(\bar{z}), g, \rho, \beta, \mathcal{G}) \neq \text{fail}$  and cost( $e_i(\bar{z}), \mathcal{G}$ ) < opt_cost
    then { opt ←  $e_i(\bar{z})$ ; opt_cost ← cost( $e_i(\bar{z}), \mathcal{G}$ ) } }
};
return opt;

```

Figure 6.5 A best-first search solution of $\bigwedge_i \mathcal{I}\{T_i\} \Rightarrow F(x_1, \dots, x_n) = g$

g (using Algorithm 5.1 only, as it does not contain higher-order variables), and its cost is compared with the cost of the best solution found so far. If its cost is better, it becomes the new best solution. Note that if we have an infinite number of solutions for F then this algorithm will fall into an infinite loop. To avoid this, we need an additional condition in the while-loop to stop the search if the size of S or the number of non-failed $e_i(\bar{z})$ becomes too large.

Algorithm 6.3 is more efficient than Algorithm 6.2 in most cases but they both suffer from the same problem: they are too rigid to incorporate heuristics for improving the optimization time. For example, there are well-known heuristics in relational algebra for selecting the best order of joins from a number of permutations [43, 67], even though not all permutations are examined. An open problem remains: how can the database implementor insert customized heuristics in the optimization algorithm

to eliminate paths that do not look very promising, according to some criteria that cannot be incorporated in the cost function? A necessary tool for such a facility is the use of compile-time linguistic reflection [72] which allows the manipulation and insertion of new program fragments during compile-time, changing the behavior of the compiler itself.

6.5 Translating Encapsulated Functions Incrementally

So far we have assumed that a function is translated after it is put into a uniform traversal combinator form, that is, after all function calls have been expanded and the composition algorithm has been used to eliminate traversals on traversals. But this is inefficient as it might perform the same optimization of a function multiple times. In addition, this destroys encapsulation, where the implementation of a function is hidden from the user of the function. The translation process can be considerably improved by having functions encapsulated, because the implementation of functions can be changed without affecting the other functions.

Definition 4.4 of uniform traversal combinators is extended to include function calls of the form:

$$\lambda\bar{x}.\text{name}(e_1(\bar{x}), \dots, e_n(\bar{x}))$$

where `name` is a previously defined function computed by a uniform traversal combinator and each e_i is a uniform traversal combinator. If the expression that computes function `name` traverses one of the function parameters then we cannot pass an expression to this parameter that contains a free accumulative result variable.

The following theorem says that we can always find the implementation of any uniform traversal combinator (that may have function calls) from Equality 6.1 without opening any function definition:

Theorem 6.1: Let $f(\bar{x})$ be a uniform traversal combinator that contains a call to a function $h(e_1(\bar{x}), \dots, e_m(\bar{x}))$, that is, $f(\bar{x}) = g(h(e_1(\bar{x}), \dots, e_m(\bar{x})), \bar{x})$

for some uniform traversal combinator g . If F is an implementation of f then there is an implementation H of h such that $F(\bar{y}) = G(H(E_1(\bar{y}), \dots, E_m(\bar{y})), \bar{y})$, where none of the G and E_i depends on h or H .

Proof: Let $F(\bar{y}) = c_0(f(r_1(y_1), \dots, r_n(y_n)))$, for some coding c_0 and some representation r_i . This is equal to $c_0(g(h(\dots, e_i(r_1(y_1), \dots, r_n(y_n)), \dots), r_1(y_1), \dots, r_n(y_n)))$. But $h(\dots, e_i(r_1(y_1), \dots, r_n(y_n)), \dots) = h(\dots, r_{e_i}(E_i(\bar{y})), \dots)$ for some implementation E_i of e_i , where r_{e_i} is a representation function for the output type of e_i . The last expression is equal to $r_h(H(\dots, E_i(\bar{y}), \dots))$ for some implementation H of h , where r_h is a representation function for the output of h . Therefore, $F(\bar{y}) = c_0(g(r_h(H(\dots, E_i(\bar{y}), \dots))), r_1(y_1), \dots, r_n(y_n)) = G(H(\dots, E_i(\bar{y}), \dots), \bar{y})$, where G is an implementation of g . \square

A function is called **opaque**, expressed in ADABTPL as

opaque function name signature;

body;

if the body is not permitted to be seen by the compiler during the translation of other functions (otherwise it is called **transparent**).

Algorithm 5.1 (on page 99) that tests the equality of two uniform traversal combinators is expanded to include the following rules:

$\mathcal{E}(\lambda \bar{x}. \text{name}(e_1(\bar{x}), \dots, e_m(\bar{x})), \lambda \bar{x}. \text{name}(h_1(\bar{x}), \dots, h_m(\bar{x})), \rho, \beta)$ $\longrightarrow \mathcal{E}(e_1, h_1, \mathcal{E}(e_2, h_2, \mathcal{E}(\dots \rho, \beta), \beta), \beta)$	
$\mathcal{E}(\lambda \bar{x}. \text{name}(e_1(\bar{x}), \dots, e_m(\bar{x})), g, \rho, \beta)$ $\longrightarrow \mathcal{E}(\lambda \bar{x}. \Phi(\text{expand}(\text{name}))(e_1(\bar{x}), \dots, e_m(\bar{x})), [], [], g, \rho, \beta)$	<p style="text-align: right;">where name is transparent</p>
$\mathcal{E}(\lambda \bar{x}. \text{name}(e_1(\bar{x}), \dots, e_m(\bar{x})), g, \rho, \beta)$	<p style="text-align: right;">where name is opaque</p>

where $\text{expand}(\text{name})$ returns the uniform traversal combinator that computes name . This algorithm indicates that if the function name is opaque then a call to this

function can only be equal to a call to the same function. If name is transparent then its definition is unfolded and passed to the equality tester.

Algorithm 5.2 (on page 110) needs no changes for transparent functions. For opaque functions though one possible solution for

$$\mathcal{E}(\mathcal{F}_T(c, r_1, \dots, r_n)(f), \lambda \bar{x}. \text{name}(e_1(\bar{x}), \dots, e_m(\bar{x})), \rho, \beta)$$

is $f = \lambda \bar{x}. \text{name}(\dots, h_i(\bar{x}), \dots)$ where:

$$\begin{aligned} & \mathcal{E}(\Phi(c(\text{name}(\dots, \mathcal{F}_{T_i}(\lambda x.x, r_1, \dots, r_n)(f_i), \dots)), [], []), \\ & \lambda \bar{x}. \text{name}(e_1(\bar{x}), \dots, e_m(\bar{x})), \rho, \beta) \vdash f_i = h_i \end{aligned}$$

Similarly, Algorithm 6.2 that optimizes programs can be changed accordingly to capture opaque function calls. The cost of an opaque function call, as well as the optimized implementation of this call, depends on the database path signatures of the input parameters of this call. Next we will present a method of deriving this information incrementally from the program that computes the opaque function.

Let name be an abstract opaque function of type $[T_1, \dots, T_n] \rightarrow T$. For each such function name we associate a mapping from constructor paths to implementations of name:

Definition 6.6: (Constructor Path Signature) Let name be an abstract opaque function of type $[T_1, \dots, T_n] \rightarrow T$. Then a constructor path signature $\mathcal{PS}_{\text{name}}$ is a tuple $[p_1, \dots, p_n]$, where each p_i is a database constructor path of type T_i .

For example, one constructor path signature in the part-subpart database for the function part_cost defined as:

```
function part_cost ( p: part ) : int;
tc_part( [?,c] →c, [?,c,ps] →tc_set( [] →c, [a,?,r] →r+a)(ps))(p);
```

is [[choose,insert,parts,make_db]]

Different path signatures of name may produce different optimized programs for NAME, the implementation of name. All these possible optimized programs need to be considered when we translate a program that calls name.

Definition 6.7: (Encapsulated Implementation) The encapsulated implementation of a function f is a mapping from any constructor path signature \mathcal{G} of f to the best implementation of f with respect to \mathcal{G} .

There is a finite number of constructor path signatures for a function f because there are a finite number of database constructor paths of type T_i . More specifically, if we have k_i database constructor paths of type T_i then we have $\prod_i k_i$ path signatures. Therefore, we need to extract $\prod_i k_i$ translations of f . Luckily, this is not really necessary. We could consider only the constructor path signatures that are actually referred to by the program. We start with transactions or database queries that work directly on the database object. Their path signatures are fixed (they are empty). Then, if a function f has a path signature \mathcal{G} and there is a function call $g(\bar{e})$ in the body of f then a new path sequence for g is derived by considering the path sequences of the expressions \bar{e} by traversing the database constructor tree. These possible translations of f could be stored in a database so that f could be used from different programs without the need of deriving the same optimizations. When a new path signature is encountered then this database is expanded to include this new implementation. Note that this database needs to be updated periodically as the statistics about the database become invalid after a period of time.

6.6 The Concrete Layer

The program optimizer described in Section 6.4 forms the basis of a flexible user-controlled translator. Using this model, all abstract programs can be translated

into lower-level primitives. These primitives are part of the concrete layer which can also be changed and extended as the users needs change and database technology evolves. The concrete layer implementor is assigned the job of modifying these primitives. This task, keeping the system in accordance with the users' needs, is as important as the programming process itself. A lot of effort has been devoted recently to special database applications, such as VLSI and CAD design, that need special storage structures efficient enough to form an interactive system. There is no indication that the storage structures can be limited to a fixed set that can be applied to everyone's needs, and we believe that new requirements will be found that cannot be handled efficiently by the current methods, as new applications are steadily discovered for computers.

There are two different tasks that have to be performed by the concrete layer implementor:

- to define the storage objects and write the code for the primitive operations;
- to describe the underlying theory of this layer.

To achieve the first task, ADABTPL provides a fixed layer, called the lowest layer, that has the primitives needed to implement the concrete layer. An important issue in defining a concrete layer is the uniform treatment of all types and operations. All abstract objects, as well as the operations upon them, must be supported by at least one type of implementation by this system. This requirement is not trivial for languages like ADABTPL where functions are first class objects. For these languages, the meta-data structures (for example, the program schemes and the type representations) must be able to be mapped in the available storage structures like any other object. It is also very important to have redundancy in storage selection, because our system is not a rigid persistent language translator. Here the implementor must have the ability to select his/her own choice of implementation among a reasonably large variety of possibilities.

In order to prove that a translation is correct we need to have the theory specifying the abstract layer, the theory specifying the concrete layer, and the theory produced by the transformation specification. We have already seen that the theory that describes a database specification in ADABTPL is derived by the schema definition and the operation specification. Therefore, the theory of the concrete layer must be given in the same way: by defining the types of the storage objects along with the primitive operation definitions. But the storage objects and the operations upon them may be very complex. Consider for example a B-tree object and its delete operation. This operation is several hundred lines of code long, as it needs to include many different cases to reduce the size of a B-tree. But even if we use these structures to generate the concrete layer theory, this theory must refer to some other theory, such as the theory about the lowest layer that has more primitive operations that do explicit system calls for disk I/O. We think that it is more reasonable to stop refining the theory before the concrete layer and assume that all primitive operations in the concrete layer are correct. Therefore, we need to specify a theory for the concrete layer that does not reflect exactly the specification of the actual types and operations of this layer but relates the concrete operations with the concrete types the same way, that is, the two theories are the same. This can be done by giving explicitly all the lemmas needed to describe it or give a set of type definitions that has the same theory. We adopted the latter approach, as it gives a well-formed theory to work with. The consequence of this convention is that now we have to assume that the concrete layer is correct. By assuming that and using correctness-preserving transformations we guarantee that the implementation maintains the validity of the specification. It is very difficult to prove that the specification of a concrete operation has the same behavior (functionality) as the actual operation. The concrete layer implementor must be very careful to avoid inconsistencies, introduced when the concrete layer interface is overspecified.

The specification of a concrete operation is given in the same way as the abstract operation. But in the concrete case a program that actually computes this operation during the program execution time also needs to be provided. A cost function matching the implemented operation must be provided if the default cost function does not reflect the implementation. We have already seen how the default cost of a program can be changed using the cost language primitive attached to a function definition. Here we extend the function definition to include the implementation primitive:

```
function function-name function-specification;
body;
implementation implementation;
cost cost-function;
```

The implementation is an ADABTPL program of the same type and the same modeled behavior (presumably) as function-name. When the compiler produces code that contains a call to function-name then the actual call will be to implementation. For example, the following concrete operation is performed over the type `btree` (defined on page 134):

```
function(alpha,beta) find ( x: btree(alpha,beta), val: alpha, def: beta ) : beta;
tc_list( [] → def, [p,?,r] → tc_pair( [a,b] → if a=val then b else r)(p))(x);
implementation Btree.find;
cost log2(size(x));
```

This declares that function `find` has the specification presented at the first two lines above but it is actually implemented as the ADABTPL function `Btree.find` which need not be expressed in combinator form and could be very complex. The fourth line of this declaration says that the cost function is the logarithm of the size of `btree(x)`.

Another example is the canonical type `int` implemented as the system-defined ADABTPL type `integer` that supports the usual integer operations, such as integer addition:

```
int = union ( zerop: singleton zero;
             succp: struct succ ( pred: int ) );
      implementation integer;
      cost 1;

function plus ( x: int, y: int ) : int;
tc_int( [] →y, [?,i] →succ(i))(x);
implementation [x,y] →x+y;
cost 1;
```

6.7 Conclusion

We have presented a program translator that translates abstract programs into optimal concrete programs expressed in terms of concrete primitives only. This optimizer does not destroy modularity and encapsulation as modules can be translated and optimized separately, without revealing the module implementations to the user of the module. This requires that modules have more than one optimized implementation that depend on the database paths assigned to the inputs of the module. The whole translation system is very flexible with many points where the database implementor can intervene and customize parts of the process.

C H A P T E R 7

CONCLUSION

7.1 Summary

During the course of this thesis we have seen how the type transformation model provides a framework for database system implementation with a high degree of data-independence, allowing significant separation of specification and implementation without the computational limits of current database languages. The internal schema writer provides the relationship between specification and implementation, which is used by the compiler to translate and optimize application programs. This process is facilitated by having all programs expressed in a special algebra, called the *uniform traversal combinator algebra*. The following summarizes the main features and the uses of the algebra:

- it is rich enough to capture most bulk data types and to express most polynomial time functions over these types;
- programs are expressed in a highly stereotyped recursive form;
- the theorem prover is just a reduction algorithm that uses the inductive properties of programs;
- abstract programs are translated by the type transformation model into concrete programs that are expressed in concrete primitives only;
- our algebra facilitates program optimization as there is a tractable number of equivalent programs to consider;

- integrity constraints attached to types offer alternative methods of execution;
- the program optimizer is a search over these alternatives guided by cost functions;
- cost functions for uniform traversal combinators are easier to write than for more general forms.

The goal of this research was to present a framework for effective database implementation. More specifically, the following presents some of the problems and goals set at the beginning of this thesis and the solutions given during the course of the thesis:

- A high degree of data independence by separating the specification from implementation. This was achieved by the type transformation model: the concrete layer is completely independent of the abstract layer; it is the responsibility of the internal schema writer to provide the relationship between these two layers.
- Partial control by the system designer over the translation process. Mainly achieved by the type transformation model and by the flexible cost model.
- The transformation specification is expressed in a modern programming language style, supporting features such as orthogonality, uniformity, modularity, reusability, and flexibility. The `map` type constructor, which was used for specifying type transformations, was defined to support these features.
- The design of a new algebra that supports complex structures and programs, but restricted enough to facilitate program translation and optimization. The uniform traversal combinator algebra proved to be well-suited for this task, as it captures most data types and most bulk operations required by complex database applications. Most standard optimization techniques, such as pushing

a selection inside a join, are captured by only one reduction method: the composition algorithm for uniform traversal combinators.

- An optimizer that examines alternative methods of execution to find the cheapest:
 - Generation of alternative paths of execution for accessing the same piece of information. This was done by using semantic information in the form of integrity constraints attached to types, specifying the dependencies between the instances of these types.
 - Generation of alternative equivalent programs that implement the same function. The search space must be large enough to contain the optimal program but restricted enough to make the search feasible in most cases. This was done by using the uniform traversal combinator algebra. Programs in this algebra are expressed in (or reduced to) minimal forms. Other forms of programs, such as traversals over traversals, are unlikely to be optimal, so they are reduced by the composition algorithm to minimal forms in our algebra. The only case that these forms are optimal is when they are irreducible (described in Section 4.7).
- An algorithm for comparing the execution costs of two programs. Expressing such algorithms was simplified by the fact that programs in our algebra have a very uniform structure.
- Validating the translation by using a theorem prover. Even though this was not a necessary step because all transformations used preserve program correctness, the theorem prover can be used in various stages for validating the application specification, such as proving that transactions preserve the integrity constraints, and for assuring that a representation-coding function pair is valid.

- Support of schema evolution and data restructuring. This was done by using the type transformation model.

7.2 Contributions

This thesis introduced many innovative ideas. The most important of them are summarized below:

- The integration of the type transformation model with query optimization methods for data intensive applications, making this model effective in a database context.
- The use of semantic information to guide the query optimizer. Even though there is a lot of work on semantic query optimization there is very little work on using integrity constraints as the main source of alternatives for query optimization.
- The development of a new algebra that facilitates program optimization and theorem proving. The theorem prover is a complete and efficient program, expressed in a very small number of rules.
- The use of an effective theorem prover to validate parts of the specification and the translation process.
- The use of the mapping model to perform data restructuring.

7.3 Future Research

Even though the work described in this thesis is a framework for effective query translation and optimization, it is far from being a complete program optimization system. In the future we intend to:

- Make adjustments to our algebra:
 - Extend the definition of traversal combinators to capture all unary traversal combinators (that is, all traversals expressed in our algebra) as well as structural equalities of any type. That way there will be no need for an explicit equality combinator. Furthermore, if there is a reduction law for these traversals as powerful as the composition algorithm, then every operation will be considerably simplified: the definition of uniform traversal combinators will have only three items, because equality will be a traversal; there will be no need for an equality algorithm for testing if two expressions compute the same function, because this is done by reducing the equality traversals; theorem proving will consist of testing whether the expression produced by the reduction algorithm is a tautology, a trivial test. We have already experimented with n-ary traversal combinators that capture all the unary ones, all structural equalities, and other forms of recursion. Even though these traversals satisfy a theorem similar to the promotion theorem, the composition algorithm based on this theorem is incomplete. This is a very exciting area that we would like to spend more time in the future.
 - Introduce new types of traversals that can more naturally express functions already expressible in our algebra, such as reverse, or can capture functions that may not be expressible in our algebra, such as transitive closure.
- Apply the UTC algebra and simplification algorithms to domains other than the database implementation, such as for functional programming and category theory. We believe that the UTC composition algorithm can be easily extended to simplify categorical terms on categories other than the set category, and, therefore, to become a universally applied optimization algorithm.

- Extend the UTC algebra to capture more mutable structures, such as mutable vectors and objects with sharing and cycles. The database object itself must be a completely mutable object so that updating the database does not imply copying the old database to a new database that reflects the updates. Proving functional equalities between programs that manipulate mutable structures is very difficult. For example, expression `new(state)` creates a new object which may or may not be equal to expression `new(state)` in another program. One way to avoid this problem is to make the `utc` meta-type (that is, the type that represents all programs in our algebra, described in Section 5.6) an object type with identity that may include sharing. That way testing whether two expressions `new(state)` are the same reduces to testing whether they have the same identity.
- Include imperative constructs in the concrete layer. Compiling a sequence of destructive updates to a functional expression that transforms the program state is usually inefficient. The only way to overcome this problem is to include imperative primitives in our algebra that express these updates naturally and efficiently. One example of such construct is the statement sequence, such as the `begin-end` statement in Pascal.
- Have the theorem prover/optimizer learn short cuts during the rewrite process, such as already proved theorems and blocks of rewrite rules that appear often, in order to avoid redundant computations during optimization and theorem proving.
- Experiment more with the optimization algorithm and the cost functions. Efficient optimization can be considerably facilitated by using heuristic information to guide the search through the choices of implementations. An example of

such heuristic information is considering traversals last whenever we generate program patterns, because they are usually more expensive.

- Write a complete implementation library of type mappings intended to be used for real database applications. This must be rich enough to contain storage structures and programs usually found in commercial database management systems, such as B-trees and hash tables. It should also consider the limitations of resources imposed by hardware and operating system, such as limited memory space for data buffering.
- Apply this model to special complex applications, such as rule-based systems and text-retrieval systems.
- Make the translator suggest to the designer a set of storage structures to choose from, based on compile-time and/or run-time analysis of the way these structures are manipulated by the program. This is a very difficult task, as is demonstrated by systems such as SETL, mainly because the number of structures applicable to a certain program may be very large.

APPENDIX A

THE USER-LEVEL LANGUAGE

Our uniform traversal combinator algebra described in Chapter 4 has a very precise definition but it is far from being an easy to use programming language. In this appendix we present a language for expressing database queries and updates in a more user-friendly way. We call this language **uniform traversal combinator language (UTCL)**. Below we give a brief description of the UTCL primitives whose notation and translation are presented using denotational semantics [48].

In Figure A.1 we use the notation $\mathcal{E}\langle e \rangle$ to denote the translation of the UTCL expression e into a uniform traversal combinator form. In this figure names starting with e are UTCL expressions while names starting with v or w are variable or function names. Here the types of values are canonical types, function types, sets, vectors, or objects with identity. The basic language primitive for traversing a value in UTCL is **scan**:

$$\text{scan } e \text{ into } v \{ \dots C_i(\bar{v}_i) \rightarrow e_i; \dots \}$$

where e can be any UTCL expression. The compiler will use the composition algorithm for transforming this into a traversal over a single variable. Variable v stands for the result of the traversal at each scan step so that the accumulated results can be retrieved by accessing this variable. Each variable from \bar{v}_i is bound to the associated component of the object being traversed. The result accumulated by traversing the value of v_j^i of type T is accessed as $v : v_j^i$, where v is the name of the scan. This is translated into a new variable name $v_v_j^i$ that represents an

$$\begin{aligned}
\mathcal{E}\langle v \rangle &\longrightarrow \text{if } v \text{ is a function name then } \text{unfold}(v) \text{ else } v \\
\mathcal{E}\langle v_1 : v_2 \rangle &\longrightarrow v_1.v_2 \\
\mathcal{E}\langle (e) \rangle &\longrightarrow (\mathcal{E}\langle e \rangle) \\
\mathcal{E}\langle C_i(e_1, \dots, e_m) \rangle &\longrightarrow C_i(\mathcal{E}\langle e_1 \rangle, \dots, \mathcal{E}\langle e_m \rangle) \\
\mathcal{E}\langle [v_1, \dots, v_n] \rightarrow e \rangle &\longrightarrow [v_1, \dots, v_n] \rightarrow \mathcal{E}\langle e \rangle \\
\mathcal{E}\langle e(e_1, \dots, e_n) \rangle &\longrightarrow (\mathcal{E}\langle e \rangle)(\mathcal{E}\langle e_1 \rangle, \dots, \mathcal{E}\langle e_n \rangle) \\
\mathcal{E}\langle \text{let } v_1 := e_1, \dots, v_n := e_n \text{ in } e \rangle &\longrightarrow ([v_1, \dots, v_n] \rightarrow \mathcal{E}\langle e \rangle)(\mathcal{E}\langle e_1 \rangle, \dots, \mathcal{E}\langle e_n \rangle) \\
\mathcal{E}\langle \text{case } e \{ \dots C_i(\bar{v}_i) \rightarrow e_i; \dots \} \rangle &\longrightarrow \text{tc_T}(\dots, [\bar{v}_i, ?, \dots, ?] \rightarrow \mathcal{E}\langle e_i \rangle, \dots)(\mathcal{E}\langle e \rangle) \\
\mathcal{E}\langle \text{scan } e \text{ into } v \{ \dots C_i(\bar{v}_i) \rightarrow e_i; \dots \} \rangle &\longrightarrow \begin{cases} \text{tc_T}(\dots, [\bar{v}_i, \bar{w}_i] \rightarrow \mathcal{E}\langle e_i \rangle, \dots)(\mathcal{E}\langle e \rangle) \\ \text{where } w_j^i \text{ is } v.v_j^i \text{ for some } v_j^i \text{ of type T} \end{cases} \\
\mathcal{E}\langle \text{if } e \text{ then } e_1 \text{ else } e_2 \rangle &\longrightarrow \text{if } \mathcal{E}\langle e \rangle \text{ then } \mathcal{E}\langle e_1 \rangle \text{ else } \mathcal{E}\langle e_2 \rangle \\
\mathcal{E}\langle e_1 = e_2 \rangle &\longrightarrow \mathcal{E}\langle e_1 \rangle = \mathcal{E}\langle e_2 \rangle \\
\mathcal{E}\langle e.v_i \rangle &\longrightarrow \text{tc_T}([\dots, v_i, \dots] \rightarrow v_i)(\mathcal{E}\langle e \rangle) \quad \text{where } v_i \text{ is the } i\text{th selector of T} \\
\mathcal{E}\langle e_1[e_2] \rangle &\longrightarrow \text{tc_vector}([\] \rightarrow ?, [i, v, r] \rightarrow \text{if } i = \mathcal{E}\langle e_2 \rangle \text{ then } v \text{ else } r)(\mathcal{E}\langle e_1 \rangle) \\
\mathcal{E}\langle \text{all } v \text{ in } e_1 \text{ where } e_2 \rangle &\longrightarrow \begin{cases} \text{tc_set}([\] \rightarrow \text{emptyset}, \\ [v, ?, r] \rightarrow \text{if } \mathcal{E}\langle e_2 \rangle \text{ then } \text{insert}(v, r) \text{ else } r)(\mathcal{E}\langle e_1 \rangle) \end{cases} \\
\mathcal{E}\langle \text{for all } v \text{ in } e_1 : e_2 \rangle &\longrightarrow \text{tc_set}([\] \rightarrow \text{true}, [v, ?, r] \rightarrow \mathcal{E}\langle e_2 \rangle \text{ and } r)(\mathcal{E}\langle e_1 \rangle) \\
\mathcal{E}\langle \text{for some } v \text{ in } e_1 : e_2 \rangle &\longrightarrow \text{tc_set}([\] \rightarrow \text{false}, [v, ?, r] \rightarrow \mathcal{E}\langle e_2 \rangle \text{ or } r)(\mathcal{E}\langle e_1 \rangle)
\end{aligned}$$

Figure A.1 Translation of UTCL expressions

accumulative result variable. Another way of saying this is at each node $C_i(\bar{v}_i)$ we do $v \leftarrow f(\bar{v}_i, v : v_{k_1}^i, \dots, v : v_{k_m}^i)$, that is, v depends on the accumulated results from the nodes $v_{k_j}^i$. If there is only one accumulated result when traversing a node, that is, when there is no ambiguity, then this result can alternatively be accessed as v .

For example, the UTCL expression:

scan x into app

```
{ cons(a,r) → cons(a,app);
  nil → y }
```

is translated into $tc_list([\] \rightarrow y, [a,r,app_r] \rightarrow cons(a,app_r))(x)$. Note that we may use either $cons(a,app)$ or $cons(a,app:r)$ because $app:r$ is the only accumulative result here.

This is not true for the following program:

scan x into sum

```
{ empty → 0;
  node(i,l,r) → i+sum:l+sum:r }
```

which is translated into $tc_tree([\] \rightarrow 0, [i,l,r,sum_l,sum_r] \rightarrow i+sum_l+sum_r)(x)$. Another example is a program that returns the set of names of every subpart of part x , where type `part` was defined on page 125:

case x

```
{ base(?,?) → emptyset;
  composite(?,?,sps) → scan sps into snames
    { emptyset → emptyset;
      insert(p,?) → scan p into pnames
        { base(nm,?) → insert(nm,snames);
          composite(nm,?,?) → insert(nm,pnames) } } }
```

Here the `case` primitive is like ADABTPL's `case` expression. It can be translated into a traversal combinator that ignores all accumulative results. This program checks

first if x is basic or composite. If it is composite it traverses the set of subparts to extract all their names (this is the `snames scan`). For each part p in this set it checks if it is basic or not. If it is basic then the answer to the query is `insert(nm,snames)`, that is, the name of this subpart inserted to the accumulated result of the `snames scan`, which is the names of the rest of the `sps` subparts. If p is composite then it insert the name of p into the value accumulated when scanning the subparts of p , that is, into `pnames`.

The first rule in Figure A.1 translates a variable name. If v is the name of a function defined globally by a function unit of the form:

```
function  $v$  (  $v_1 : t_1, \dots, v_n : t_n$  ) :  $t$ ;  
 $e$ ;
```

where e is a UTCL expression, then it returns `unfold(v) = [v_1, \dots, v_n] \rightarrow e` . This is typically used in the fifth rule where $e = v$ is the name of a function. The `let` statement in the seventh rule evaluates all the bindings at the same time. Nested `let` statements can be used for sequential bindings, that is, when there is a binding whose value depends on the value of another binding. Projection `$e.v_i$` in the twelfth rule is a tuple traversal over the type T of e that selects the i th tuple component of e , where v_i is the i th selector of T . The thirteen rule says that `$e_1[e_2]$` is a vector traversal. Vector and set traversal can also be expressed using the regular `traverse` primitive. We introduce for sets three additional operations: `all v in e_1 where e_2` that returns all elements v in the set e_1 that satisfy e_2 , or `for all v in $e_1 : e_2$` and `for some v in $e_1 : e_2$` that return true when all/at-least-one of the elements v in the set e_1 satisfy e_2 .

Destructive updates, such as removing one element from a set, can be expressed more easily and naturally in an imperative language. For example, we could write `$s := \text{remove}(s,e)$` to indicate that the element e is removed from the set s . UTCL

$$\begin{aligned}
\mathcal{T}_\sigma\langle v_i := e \rangle &\longrightarrow [x] \rightarrow \text{tc}_\sigma([\bar{v}_i, v_i, \bar{v}_r] \rightarrow C_\sigma(\bar{v}_i, \mathcal{E}\langle e \rangle, \bar{v}_r))(x) \\
\mathcal{T}_\sigma\langle v_i.e_1 := e_2 \rangle &\longrightarrow [x] \rightarrow \text{tc}_\sigma([\bar{v}_i, v_i, \bar{v}_r] \rightarrow C_\sigma(\bar{v}_i, \mathcal{T}_{\tau_i}\langle e_1 := e_2 \rangle(v_i), \bar{v}_r))(x) \\
\mathcal{T}_\sigma\langle \{ \} \rangle &\longrightarrow [x] \rightarrow x \\
\mathcal{T}_\sigma\langle \{ s_1; s_2; \dots; s_n \} \rangle &\longrightarrow \mathcal{T}_\sigma\langle \{ s_2; \dots; s_n \} \rangle \circ \mathcal{T}_\sigma\langle s_1 \rangle \\
\mathcal{T}_\sigma\langle \text{if } e \text{ then } s_1 \text{ else } s_2 \rangle &\longrightarrow [x] \rightarrow \text{if } \mathcal{E}_\sigma\langle e \rangle(x) \text{ then } \mathcal{T}_\sigma\langle s_1 \rangle(x) \text{ else } \mathcal{T}_\sigma\langle s_2 \rangle(x) \\
\mathcal{T}_\sigma\langle \text{if } e \text{ then } s \rangle &\longrightarrow [x] \rightarrow \text{if } \mathcal{E}_\sigma\langle e \rangle(x) \text{ then } \mathcal{T}_\sigma\langle s \rangle(x) \text{ else } x \\
\mathcal{T}_\sigma\langle \text{let } v_1 := e_1, \dots, v_n := e_n \text{ in } s \rangle &\longrightarrow \begin{cases} [x] \rightarrow \text{tc}_{\sigma'}([\bar{w}, \bar{v}] \rightarrow C_\sigma(\bar{w})) \\ \quad (\mathcal{T}_{\sigma'}\langle s \rangle(\text{tc}_\sigma([\bar{w}] \rightarrow C_{\sigma'}(\bar{w}, \mathcal{E}_\sigma\langle e_1 \rangle(x), \dots, \mathcal{E}_\sigma\langle e_n \rangle(x))))(x)) \\ \text{where } \sigma' = \sigma \diamond [v_1 : t_1, \dots, v_n : t_n] \text{ and } e_i \text{ is of type } t_i \end{cases} \\
\mathcal{T}_\sigma\langle \text{for each } v \text{ in } e \text{ do } s \rangle &\longrightarrow \begin{cases} [x] \rightarrow \text{tc}_{\sigma'}([\bar{w}, v] \rightarrow C_\sigma(\bar{w})) \\ \quad (\text{tc}_{\text{set}}([\bar{w}] \rightarrow \text{tc}_\sigma([\bar{w}] \rightarrow C_{\sigma'}(\bar{w}, ?))(x), \\ \quad \quad [v, ?, r] \rightarrow \mathcal{T}_{\sigma'}\langle s \rangle(r))(\mathcal{E}_\sigma\langle e \rangle(x)) \\ \text{where } \sigma' = \sigma \diamond [v : t] \text{ and } e \text{ is of type } \text{set}(t) \end{cases} \\
\mathcal{T}_\sigma\langle \text{for } v := e_1 \text{ to } e_2 \text{ do } s \rangle &\longrightarrow [x] \rightarrow \text{tc}_{\text{int}}([\bar{w}] \rightarrow x, [i, r] \rightarrow \mathcal{T}_\sigma\langle \text{let } v := i + e_1 \text{ in } s \rangle(r))(\mathcal{E}_\sigma\langle e_2 - e_1 \rangle(x))
\end{aligned}$$

Figure A.2 Translation of UTCL imperative statements

supports two ways of using imperative-style constructs. The first is **imperative function**:

imperative function $v (v_1 : t_1, \dots, v_n : t_n) : t;$

$s;$

where s is an UTCL statement. No side effects are permitted here and no globally defined values, such as the database state, can be accessed in s other than passing them as parameters. Parameter values can be modified but these updates do not persist beyond the scope of this function. The name v also stands for the name of the result. Therefore, v must be set at least once in the body of s . This definition is an alternative way of expressing a pure functional expression with no side effects.

The other language construct is **transaction**:

transaction v ($v_1 : t_1, \dots, v_n : t_n$);

s ;

This assumes that the database type db is a tuple type consisting of all persistent components. In that case transaction v is equivalent to a function that accepts the current database state as input along with the extra values v_i and returns a new database state. This simulates all possible database updates expressible in our algebra. Note that we are not concerned here with issues such as consistency, recovery, or concurrency control. Transactions here are simply state transformers.

Figure A.2 shows some possible forms of UTCL statements and their translation into our algebra. Each statement is translated into a state transformer that accepts the old state x and returns a new state, possibly different. The initial state of the transaction defined above consists of all database state components plus all parameters v_i . The output state is projected into the database type because the new values of v_i are not of any interest. The initial state of an imperative function consists of the function parameters and a placeholder for the result that has the same name with the function. The output state is projected into the result value. We use the notation $\mathcal{T}_\sigma(s)$ to indicate the state transformer for the state σ that simulates the statement s . Because σ is a tuple constructed by C_σ , the traversal $tc_\sigma([\bar{w}] \rightarrow \dots)(x)$ decomposes the state x into its components \bar{w} . In addition, before translating an expression e using $\mathcal{E}(e)$ we decompose the state to extract its components; this is $\mathcal{E}_\sigma(e)$:

$$\mathcal{E}_\sigma(e) = [x] \rightarrow tc_\sigma([\bar{w}] \rightarrow \mathcal{E}(e))(x)$$

The first two rules in Figure A.2 indicate that if we change the i th state component v_i (or a part of it) then we decompose the state, we leave the other components intact, and we replace the i th component. The third and fourth rules handle the case of statement sequences. The state transformer of a statement sequence is the

composition of the state transformers of the constituent statements. The seventh rule extends the state to include local bindings in a form of a let statement. Here the inner $tc_σ$ extends the old state $σ$ into a new state $σ'$ that includes all e_i . Statement s is translated using this new state and the result is projected into the old state $σ$ (this is the first $tc_σ'$). The eighth rule is for the case of iterations over sets: for each v in e do s . This says that statement s is to be executed for every element v in the set e . Like the let statement, it extends the state $σ$ to include a placeholder for v . The inner tc_set creates a new state of type $σ'$ at each step. The first such state value (when e is empty) is equal to the old state x extended with an unspecified value $?$ for v . At each step of the iteration the state from the previous step r is used as a new state for evaluating statement s . That way all updates from the previous steps are accumulated to the resulting state. We are not concerned about efficiency here as the produced translation will be optimized by the program optimizer described in Chapter 6. The for statement in the last rule executes s by iterating v from integer e_1 to integer e_2 .

Note that if we have an imperative primitive **scan**, similar to the UTCL expression **scan**, then the semantics should look like this:

$$\mathcal{T}_σ\langle \mathbf{scan} \ e \ \mathbf{into} \ v \ \{ \dots \ C_i(\bar{v}_i) \rightarrow s_i; \dots \} \rangle \\ \longrightarrow \left\{ \begin{array}{l} tc_T(\dots, [\bar{v}_i, \bar{w}_i] \rightarrow [x] \rightarrow \mathcal{T}_σ\langle s_i \rangle(w_k(\dots w_2(w_1(x))))), \dots) (\mathcal{E}\langle e \rangle) \\ \mathbf{where} \ w_j^i \ \mathbf{is} \ v.v_j^i \ \mathbf{for} \ \mathbf{some} \ v_j^i \ \mathbf{of} \ \mathbf{type} \ T \end{array} \right.$$

that is, the tc_T traversal should return a state transformer (instead of a state) so that all accumulative results w_j are state transformers and the input state to the state transformer $\mathcal{T}_σ\langle s_i \rangle$ is the composition of all these state transformers. If w_j were states then we could not accumulate all these side effects into one state. We have not included traversals that return functions in our algebra because it is very difficult to optimize these forms. Therefore, we decided not to support scans as an imperative language primitive. All traversals should be done using the UTCL **scan** expression or using the special traversals for sets and integers.

For example, the following program updates the salary of an employee:

```

for each e in employees do
  for each d in departments do
    if e.dno=d.dno and e.name="leo" and d.name="CS"
      then e.salary:=e.salary+10000;

```

If the database state db is a tuple of two components es , the set of employees, and ds , the set of departments, then this update is translated into:

```

[x] → tc_db1([es,ds,?] → make_db(es,ds))
      (tc_set([ ] → tc_db([es,ds] → make_db1(es,ds,?))(x),
            [e,?,r] → tc_db2([es,ds,e,?] → make_db1(es,ds,e))
            (tc_set([ ] → tc_db1([es,ds,e] → make_db2(es,ds,e,?))(r),
            [d,?,s] → upd(s,e,d))
            (tc_db1([?,ds,?] → ds)(r))))
      (tc_db([es,?] → es)(x)))

```

where $db1$ is db extended with e , $db2$ is $db1$ extended with d , and $upd(s,e,d)$ is:

```

if e.dno=d.dno and e.name="leo" and d.name="CS"
  then tc_db2([es,ds,e,d] → make_db2(es,ds,
            tc_empl([nm,sal] → make_empl(nm,sal+10000))(e),d))(s)

```

Another example is creating a temporary set using a let statement:

```

let temp := emptyset in
for each e in employees do
  if e.name="leo" then temp:=insert(e,temp);

```

This is translated into:

```

[x] → tc_db1([es,ds,?] → make_db(es,ds))
  ( let y := tc_db([es,ds] → make_db1(es,ds,emptyset))(x) in
    tc_db2([es,ds,temp,?] → make_db1(es,ds,temp))
      (tc_set([] → tc_db1([es,ds,temp] → make_db1(es,ds,temp,?))(y),
        [e,?,r] → if e.name="leo" then tc_db2([es,ds,temp,e] →
          make_db2(es,ds,insert(e,temp),e))(r))
        (tc_db1([es,?,?] → es)(y))) )

```

where the let $y := \dots$ was used for making this program easier to read.

APPENDIX B

THE TRAVERSAL COMBINATORS IN CATEGORICAL TERMS

Definition 4.1 of traversal combinators and the promotion theorem can be easily explained using basic category theory [63]. A category consists of a collection of objects and a collection of arrows from objects (domains) to objects (co-domains). There is also an associative composition operator \circ that defines a composite arrow $g \circ f$ from two arrows f and g and an identity arrow id which when composed with any arrow f gives f . Here the only category that we use is the **Set** category that has sets as objects and total functions as arrows. Properties of categorical constructions are often stated and proved using diagrams. For example, if $f \circ g = h \circ k$ then we say that the following diagram commutes:

$$\begin{array}{ccc}
 A & \xrightarrow{g} & B \\
 k \downarrow & & \downarrow f \\
 C & \xrightarrow{h} & D
 \end{array}$$

The upper part of the diagram in Figure B.1 gives the definition of the list traversal combinator. It says that the type $\text{list}(\alpha)$ has two constructors: $\text{nil} : 1 \rightarrow \text{list}(\alpha)$ and $\text{cons} : \alpha \times \text{list}(\alpha) \rightarrow \text{list}(\alpha)$ (where 1 is the terminal object for **Set**, the cartesian product of zero sets). The list traversal combinator $h = \text{tc_list}(f_1, f_2)$ is a unique arrow (represented as a dotted arrow) from $\text{list}(\alpha)$ to β . Its component functions are $f_1 : 1 \rightarrow \beta$ and $f_2 : \alpha \times \text{list}(\alpha) \times \beta \rightarrow \beta$. The domain of f_2 is the result of applying h to the domain of cons ($\langle f, g \rangle : C \rightarrow A \times B$ is the product function

of $f : C \rightarrow A$ and $g : C \rightarrow B$ such that $\langle f, g \rangle (x) = \text{pairup}(f(x), g(x))$. The two upper diagrams in Figure B.1 commute:

$$h = \text{tc_list}(f_1, f_2) \Rightarrow \begin{cases} h \circ \text{nil} = f_1 \\ h \circ \text{cons} = f_2 \circ (\text{id} \times \langle \text{id}, h \rangle) \end{cases}$$

which is equivalent to the definition of the list traversal combinator.

The lower part of the diagram in Figure B.1 gives the promotion theorem. From the lower two diagrams that commute we have:

$$g \circ h = \text{tc_list}(\phi_1, \phi_2) \Rightarrow \begin{cases} g \circ f_1 = \phi_1 \\ g \circ f_2 = \phi_2 \circ (\text{id}^2 \times g) \end{cases}$$

which is equivalent to the list promotion theorem.

The upper part of the diagram in Figure B.2 gives the definition of the traversal combinator for any type T . Here we use the symbol $(-)$ to denote any type other than T . The upper diagram gives:

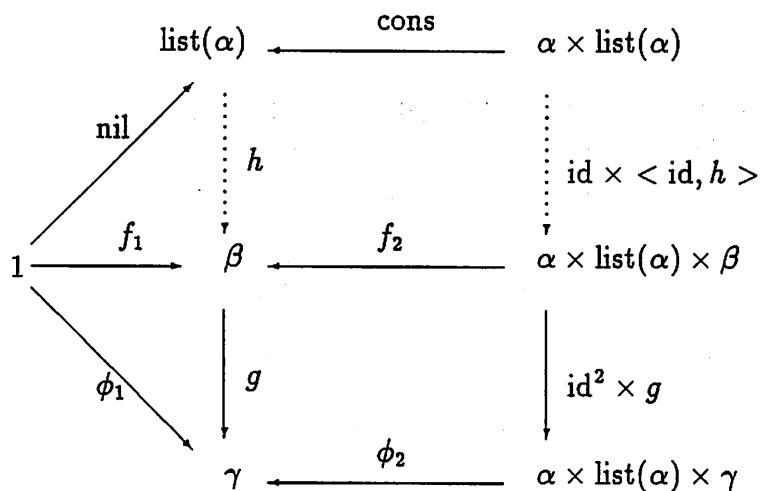
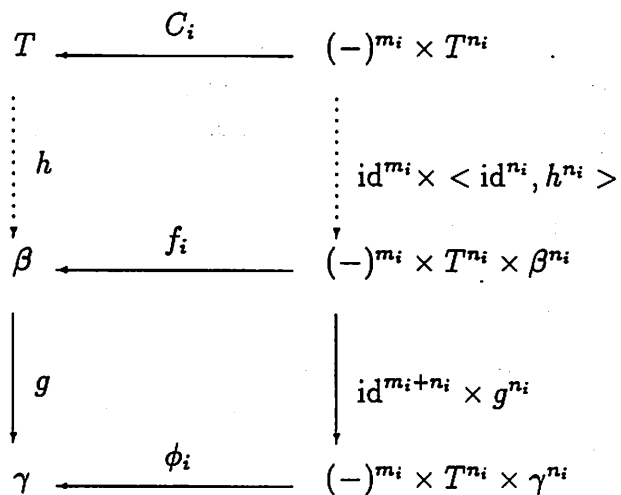
$$h = \mathcal{H}_T(f_1, \dots, f_n) \Rightarrow h \circ C_i = f_i \circ (\text{id}^{m_i} \times \langle \text{id}^{n_i}, h^{n_i} \rangle)$$

which is equivalent to the definition of the T traversal combinator, and the lower diagram gives:

$$g \circ h = \mathcal{H}_T(\phi_1, \dots, \phi_n) \Rightarrow g \circ f_i = \phi_i \circ (\text{id}^{m_i+n_i} \times g^{n_i})$$

which is equivalent to the promotion theorem for any type T .

The categorical programming language Charity [18] is based on categorical constructs similar to those described above. Their fold^L operator is very similar to our traversal combinator but more generalized to capture lambda currying explicitly and uniformly.

Figure B.1 The list traversal combinator $h = \text{tc.list}(f_1, f_2)$ Figure B.2 The traversal combinator $h = \mathcal{H}_T(f_1, \dots, f_n)$

B I B L I O G R A P H Y

- [1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. Technical report, GIP ALTAIR, Le Chesnay, France, September 1989. TR 30-89.
- [2] M. P. Atkinson and O. P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105-190, June 1987.
- [3] F. Bancilhon. Object-Oriented Database Systems. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas*, pages 152-162, September 1988.
- [4] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711-1729, November 1988.
- [5] D. S. Batory. Modeling the Storage Architectures of Commercial Database Systems. *ACM Transactions on Database Systems*, 10(4):463-528, December 1985.
- [6] D. S. Batory. Progress Toward Automating the Development of Database System Software. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 261-296. Springer-Verlag, 1985.
- [7] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. Wise. Genesis: A Reconfigurable Database Management System. Technical report, Department of Computer Science, University of Texas at Austin, March 1986. TR-86-07.
- [8] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *International Conference on Database Theory, Paris, France*, pages 72-88. Springer-Verlag, December 1990.
- [9] S. Bellantoni and S. Cook. A new Recursion-Theoretic Characterization of Polynomial Time. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.
- [10] R. S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487-504, October 1984.
- [11] D. Bjorner and C. B. Jones. *Formal Specification & Software Development*. Prentice-Hall International, 1982.

- [12] T. Bloom and S. B. Zdonik. Issues in the Design of Object-Oriented Database Programming Languages. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida*, pages 441-451, October 1987.
- [13] C. Bohm and A. Berarducci. Automatic Synthesis of Typed λ -Programs on Term Algebras. *Theoretical Computer Science*, 39:135-154, 1985.
- [14] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
- [15] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pages 9-19. Morgan Kaufmann Publishers, Inc., August 1991.
- [16] R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44-67, January 1977.
- [17] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471-522, December 1985.
- [18] R. Cockett and T. Fukushima. About Charity. Unpublished paper, University of Calgary, Alberta, Canada, May 1992.
- [19] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29-72, March 1988.
- [20] J. Darlington and R. Burstall. A System which Automatically Improves Programs. *Acta Informatica*, 6(1):41-60, 1976.
- [21] J. Darlington. A Synthesis of Several Sorting Algorithms. *Acta Informatica*, 11(1):1-30, 1978.
- [22] J. Darlington. An Experimental Program Transformation and Synthesis System. *Artificial Intelligence*, 16(1):1-46, March 1981.
- [23] J. Darlington. The Synthesis of Implementations for Abstract Data Types. In *Computer Program Synthesis Methodologies*, pages 309-334. D. Reidel Publishing Company, 1983.
- [24] R. Dewar, A. Grand, S. Liu, J. Schwartz, and E. Schonberg. Programming by Refinement, as Exemplified by the SETL Representation Sublanguage. *ACM Transactions on Programming Languages and Systems*, 1(1):27-49, July 1979.
- [25] M. S. Feather. A System for Assisting Program Transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1-20, January 1982.

- [26] L. Fegaras, T. Sheard, and D. Stemple. The ADABTPL Type System. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 243–254. Morgan Kaufmann Publishers, Inc., June 1989.
- [27] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In *Proceedings of the 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York*. Springer-Verlag, June 1992.
- [28] L. Fegaras and D. Stemple. Using Type Transformation in Database System Implementation. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pages 337–353. Morgan Kaufmann Publishers, Inc., August 1991.
- [29] J. C. Freytag and N. Goodman. Translating Aggregate Queries into Iterative Programs. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan*, pages 138–146, August 1986.
- [30] J. C. Freytag and N. Goodman. Translation of Relational Queries into Iterative Programs. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D.C.*, pages 206–214, May 1986.
- [31] J. C. Freytag and N. Goodman. On the Translation of Relational Queries into Iterative Programs. *ACM Transactions on Database Systems*, 14(1):1–27, March 1989.
- [32] J. C. Freytag. *Translating Relational Queries into Iterative Programs*. Springer-Verlag, 1987. Lecture Notes in Computer Science 261.
- [33] J. C. Freytag. A Rule-Based View of Query Optimization. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 173–180, December 1987.
- [34] K. Futatsugi, J. A. Coguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana*, pages 52–66, January 1985.
- [35] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 160–171, December 1987.
- [36] G. Graefe and K. Ward. Efficient Evaluation of Right-, Left-, and Multi-Linear Rules. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 358–366, June 1989.
- [37] G. Graefe. The Stability of Query Evaluation Plans and Dynamic Query Evaluation Plans. Technical report, Department of Computer Science and Engineering, Oregon Graduate Center, 1988. CS/E 88-003.

- [38] D. Gries and J. Prins. A New Notion of Encapsulation. *ACM Symposium on Language Issues in Programming Environments*, pages 131–139, July 1985.
- [39] D. Gries and D. Volpano. The Transform—A New Language Construct. *Structured Programming*, 11:1–10, 1990.
- [40] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.
- [41] G. Huet and B. Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica*, 11(1):31–35, 1978.
- [42] N. Immerman, S. Patnaik, and D. Stemple. The Expressiveness of a Family of Finite Set Languages. *Proceedings of the 10th ACM Symposium on Principles of Database Systems, Denver, Colorado*, pages 37–52, May 1991.
- [43] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.
- [44] C. B. Jones. Systematic Program Development. In *Software Specification Techniques*, pages 89–109. Addison-Wesley, 1986.
- [45] E. Kant. On the Efficient Synthesis of Efficient Programs. *Artificial Intelligence*, 20(3):253–305, May 1983.
- [46] D. Kapur and M. Srivas. A Rewrite Rule Based Approach for Synthesizing Abstract Data Types. In *Mathematical Foundations of Software Development*, pages 188–207. Springer-Verlag, March 1985.
- [47] W. Kim. Global Optimization of Relational Queries: A First Step. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 206–216. Springer-Verlag, 1985.
- [48] A. Lloyd. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1978.
- [49] D. B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, January 1977.
- [50] J. R. Low. Automatic Data Structure Selection: An Example and Overview. *Communications of the ACM*, 21(5):376–385, May 1978.
- [51] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, June 1989.
- [52] Z. Manna and R. Waldinger. Toward Automatic Program Synthesis. *Communications of the ACM*, 14(3):151–165, March 1971.
- [53] Z. Manna and R. Waldinger. Knowledge and Reasoning in Program Synthesis. *Artificial Intelligence*, 6(2):175–208, 1975.

- [54] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.
- [55] S. Mazumdar. *Enhancing Database Integrity and Security through Feedback to Designers*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, September 1991.
- [56] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pages 124–144, August 1991.
- [57] A. Mili, J. Desharnais, and J. Gagne. Formal Models of Stepwise Refinement of Programs. *ACM Computing Surveys*, 18(3):231–276, September 1987.
- [58] A. Ohori. Representing Object Identity in a Pure Functional Language. In *International Conference on Database Theory, Paris, France*, pages 41–55. Springer-Verlag, December 1990.
- [59] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli – A Polymorphic Language with Static Type Inference. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 46–57, May 1989.
- [60] R. Paige. Transformational Programming—Applications to Algorithms and Systems. *Proceedings of the 10th ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 73–87, January 1983.
- [61] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.
- [62] F. Pfenning and C. Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics, New Orleans, Louisiana*, pages 209–228, March 1989.
- [63] B. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.
- [64] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill Company, 1967.
- [65] D. Sannella. A Survey of Formal Software Development Methods. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1988. ECS-LFCS-88-56.

- [66] E. Schonberg, J. Schwartz, and M. Sharir. An Automatic Technique for Selection of Data Representations in SETL Programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126-143, April 1981.
- [67] G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pages 23-34, May 1979.
- [68] T. Sellis. Global Query Optimization. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D.C.*, pages 191-205, May 1986.
- [69] T. Sheard. Generalized Recursive Structure Combinators. Technical report, Department of Computer and Information Science, University of Massachusetts, 1989. TR89-26.
- [70] T. Sheard. Automatic Generation and Use of Abstract Structure Operators. *ACM Transactions on Programming Languages and Systems*, 19(4):531-557, October 1991.
- [71] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 12(3):322-368, September 1989.
- [72] T. Sheard. A User's Guide to TRPL: A Compile-time Reflective Programming Language. Technical report, Department of Computer and Information Science, University of Massachusetts, 1990. TR90-109.
- [73] W. Snyder and J. Gallier. Higher-order Unification Revisited: Complete Sets of Transformations. *Journal of Symbolic Computation*, 8(2):101-140, 1989.
- [74] M. Spivey. A Categorical Approach to the Theory of Lists. In *Mathematics of Program Construction*, pages 399-408. Springer-Verlag, June 1989.
- [75] D. Stemple, S. Mazumdar, and T. Sheard. On the Modes and Meaning of Feedback to Transaction Designers. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 374-386, May 1987.
- [76] P. Trinder. Comprehensions: A Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pages 55-68. Morgan Kaufmann Publishers, Inc., August 1991.
- [77] P. Valduriez and S. Danforth. Query Optimization for Database Programming Languages. Unpublished paper.

- [78] P. Valduriez, S. Danforth, B. Hart, T. Briggs, and M. Cochinwala. Compiling FAD, A Database Programming Language. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 375–393. Morgan Kaufmann Publishers, Inc., June 1989.
- [79] B. Vance. Towards an Object-Oriented Query Algebra. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, January 1992. CS/E 91-008.
- [80] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, March 1988. Lecture Notes in Computer Science 300.
- [81] P. Wadler. Comprehending Monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.
- [82] N. Wirth. The Development of Programs by Stepwise Refinement. *Communications of ACM*, 14(4):221–227, April 1971.