PROGRAMMING DATA STRUCTURES IN LOGIC

Russell Turpin
The University of Texas at Austin, 1992

Abstract: Current programming languages that are grounded in a formal logic — such as pure Lisp (based on the lambda calculus) and Prolog (based on Horn clause logic) — do not support the use of complex, pointer-based data structures. The lack of this important feature in logically grounded languages contrasts sharply with its strong support in the imperative programming languages that have enjoyed wide application, of which C is a prime example. Unfortunately, the formal methods for reasoning about imperative languages have not proved broadly useful for reasoning about programs that manipulate complex, pointer-based data structures. Between these two camps resides an open question: How can we verify programs involving complex, pointer-based data structures?

This work gives an answer to this question. It describes a programming language in which a programmer can define logical predicates on data structures and pointers, and use these predicates to specify programs that manipulate complex, pointer-based data structures. These programs may dynamically allocate memory and destructively modify their arguments. This solution is grounded in two theoretical advances. (1) This work develops a *first-order logic for data structures* that formalizes the notions that are necessary for defining and reasoning about relationships between data structures, including notions such as the address of a data structure, pointer reference, reachability via pointer reference, and data structure overlap. (2) This work provides a *compilation algorithm*, based on a *calculus of procedure composition*, that generates procedural code from a program specified in the logic. Compilation is in the style of automatic programming, and relies on the programmer, using theorem proving tools, to verify assertions in the logic that are generated by the compilation algorithm.

www.manaraa.com

Table of Contents

www.manaraa.com

www.manaraa.com

www.manaraa.com

1. Introduction

The goal of this research is to establish a representation for programs and an accompanying compilation process satisfying three criteria:

(1) **Expressiveness for data structures.** Programmers can use the representation to define and manipulate general data structures, including those that involve complex pointer relationships.

(2) **Run-time efficiency.** The representation compiles to efficient code.

(3) **Support for formal reasoning.** The representation supports formal reasoning about the data structures and their manipulations.

The importance of this research lies in the combination of these criteria. Imperative programming languages, of which C will serve as the paradigmatic example, achieve the first two criteria, but they do not support formal reasoning. Logic languages and applicative languages support formal reasoning — in their theoretically pure forms — but they do not provide the expressive power and efficiency of C when dealing with general data structures, especially those with a complex pointer structure. This work presents a new paradigm for logic programming that attains the stated goal.

The traditional implementations of logic programming, based on unification and goal expansion, search the space of possible proofs by generating all possible substitution terms, each of which satisfies some of the clauses of the logic program. Efficient execution of a logic program must avoid this kind of *speculative computation*. Efficiently maintaining some kinds of data structures also requires *destructive update*, where the resulting data structure instance is produced by changing a portion of the original data structure instance, without *any* copying of the unchanged portion.

The compilation process described in this work determines a static order of computation for a program that realizes a logical predicate. It builds a procedure that realizes the predicate from procedures that realize the logical components of the predicate. At run-time, the program binds values to variables in

www.manaraa.com

an order that incrementally satisfies portions of the predicate, until all output variables are bound to values that satisfy the entire predicate. The run-time execution is *progressive*: once made, a binding is never undone. The run-time execution is also *effective*: if there exist output values that satisfy the predicate, then the execution finds these values.

The compilation process supports the incremental construction of libraries of verified programs. A program that is produced by the compilation process is known to satisfy its specification, and becomes available for use in compiling a larger program whose specification makes use of the first program's predicate.

The compilation process applies generally to languages based on the first-order predicate calculus. In counterpoint to the good qualities listed above, it suffers two weaknesses. First, the compilation process may require the programmer to modify, in a way that preserves logical equivalence, the original predicate definition. Predicates can be defined at a level that is "too abstract" for compilation, in which case they must be written in a computationally more concrete fashion so that the compilation process can produce code for them. Second, the compilation process depends on an oracle that tells whether assertions in the logic are true. In practice, this oracle is the programmer working with available theorem-proving tools. Wrongly verifying a false assertion may cause the compilation process to produce incorrect code. Failure to verify a true assertion will prevent the compilation process from producing code for the program in question. Both of these issues are examined in detail in this work.

This combination of qualities in the compilation process — on the positive side, the generation of progressive and effective executables for programs that are logically specified, and the hierarchical composition of verified programs, and on the negative side, the expense and trouble of an interactive compilation process — may be useful in application domains other than data structure programming. This research direction is not explored in the current work.

www.manaraa.com

1.1 The Problem and the Approach

Addressed memory is the primary store for virtually all modern computer systems.[1] This basic fact permeates the art and practice of software. Software engineers think of data structures in terms of addressed memory: this piece over here points to that piece over there. Programmers still cut their eye teeth on Knuth's multivolume work [32], large portions of which are concerned with the problems of addressed memory. The popular imperative languages such as C, Pascal, and Modula are characterized by the freedom they give the programmer to allocate, release, and point to memory. (In this regard, the modern language C++ is distinguished by the complexity of mechanisms available and the subtlety of the rules that must be followed in writing correct programs.)

Addressed memory is cheap, and its direct use in programming languages is powerful. But this power carries a price. In industry, software projects that use these languages are plagued by subtle bugs that are traced — with difficulty, and often after great expense — to mangled pointers or erroneous memory management. Businesses that develop software evaluate expensive software development tools partly on their ability to help with these kinds of bugs. Even the customer becomes aware of the software problems associated with the cavalier use of addressed memory, when a delivered product exhibits a memory "leak" or displays an occasional tendency to trounce on the wrong data.

Academia has proposed a variety of formal approaches to deal with the software problem. Programming languages are given formal semantics, so that one can prove things about them, and mathematical languages such as the λ–calculus and Horn clause logic are implemented as programming systems. These approaches provide views of data structures that are considerably more abstract than the addressed memory visible to the C programmer. Conversely, the C

[1] While novel memory architectures are explored in research environments and serve some specialized niches — such as the use of associative memory for regular pattern matching and the synaptic memory in neural nets for more complex pattern recognition — random access memory is far more economical for most purposes, and is likely to remain so for the foreseeable future.

www.manaraa.com

programmer's understanding of data structures has resisted formal treatment.

This work takes the pointer-based data structures that are the staple of the C programmer as a litmus test of formal methods. It assumes that the expressiveness of C in this regard is worthwhile, and that for practical reasons, it is not something that will be easily discarded. Using this litmus test, it is fair to classify how formal methods deal with data structures into three groups.

Avoidance. Many semantically "clean" languages rely only on a view of data structures that is considerably simpler than that available to the C programmer. For example, pure Lisp [62] relies on lists and Prolog [14] relies on functors. Algebraic data types [23] rely on initial algebras. The problem with this approach is that the reliance on simpler data structures limits the applications for which the language is used. Often, a language such as these is extended with constructs that support more general data structure use, but that sully its formal cleanliness.

Prevention. The semantics of languages such as C can be axiomatized using continuations and a notion of global memory state [22, 53]. This approach has proved impractical, despite some efforts to build support for verification into programming environments [3]. Languages such as C have resisted the broad use of formal techniques to verify complex programs written in them.

Camouflage. Infinite, recursive structures provide a view of data structures that are roughly equivalent to the pointer-based data structures of C. Similarly, graph grammars [54, 55] have been used to implement data structures more general than lists and arrays. While these formal approaches attempt to deal head on with complex data structures, their views of data structures are foreign to programmers familiar with the classical notions, and they provide less expressiveness than C, especially regarding memory management and destructive update.

Rather than rejecting the pointers and data structures of C as something too ugly for study, trying to create an axiomatic or denotational semantics for an imperative language (very hard!), or hiding data structures behind mathematical objects such as infinite graphs, this work embeds the data structures and pointers of C in a first-order logic, giving them a formal foundation while preserving most of the C programmer's understanding of them. The problem this work tackles, and the approach it takes, are summarized below.

www.manaraa.com

Problem Statement:

> Is there a way to program pointer-based data structures that preserves the expressiveness and efficient execution of C, while providing support for formal reasoning?

Approach:

(1) Develop a *first-order logic* in which one can express invariants for, operations on, and proofs about data structures.

(2) Develop a *compilation process* that turns operations expressed in the logic into efficient C programs.

There is a special benefit to this approach that deserves comment. Code reuse is one of the major problems in the engineering of large software systems involving complex data structures. With procedural programming, code reuse relies on the programmer understanding that a particular procedure has the desired behavior, and identifying that procedure by name. The development of object-oriented languages makes reuse a little easier, by allowing the programmer to specify some logical properties between different kinds of data structures, to wit, that one kind of data structure is a specialization of a second, and shares in its invariants and methods, unless these are explicitly redefined. With the programming paradigm developed in this work, data structure invariants are fully stated in a first-order logic. These invariants are used to verify that programs are applied only to data structures on which they correctly operate. In short, a procedural programming environment "knows" only the name of a piece of code. An object-oriented programming environment additionally "knows" about inheritance and genericity relationships between data types. But a logic programming environment "knows" fully of what a program does, and what preconditions are required for it. This leads to the possibility of code reuse through theorem proving, something that could become especially important for large software systems that rely on complex data structures [11]. This theme is only lightly explored in this work, though its promise is a motivating influence.

The approach stated above requires the development of a logic and the development of a compilation process. These are now summarized.

www.manaraa.com

The Logic

The logic presented in Chapter 2 is built on a hierarchy of sorts that capture fairly well the data types of C, including a notion of pointer reference. A measure of the fidelity of this mapping is that most sort expressions are carried by the compiler into C declarations without any syntactic change. For example, a logical variable whose sort is `struct Pt {float X; float Y;}` will be identically typed in C. Pointers in the logic generally behave as they do in C, though some of the intricacies of C's pointer arithmetic are absent.

The logic was developed as an extension of the many-sorted first-order predicate calculus, rather than as a variety of Horn clause logic. This departure from the custom of the logic programming community was motivated by a personal preference for the programming style afforded by the predicate calculus, and made possible by the compilation process. Regarding the first issue, the author believes that localizing the definition of a new predicate in a single logical formula makes programs more understandable. This style of logical definition is closer to the way logic is used in the many other areas to which it is applied as a medium for rigorous communication. Regarding the second issue, much of the impetus for Horn clause logic stems from the use of algorithms that require it, such as SLD resolution [14]. For the sake of run-time efficiency, the logic programming language described in this work does not use SLD resolution, nor even perform run-time unification. Departing from this kind of implementation removes much of the motivation for using Horn clause logic. (Bowen wrote an early paper [10] on the use of full first-order logic for programming.)

Horn clause logic has also been popular for theoretical reasons. Because Horn clause logic restricts the use of negation, it is easy to assign semantic models to new predicates defined in it [2, 37]. Recent developments in semantics [20, 57] have broadened the class of logic programs that can be assigned models, and lessened the restrictions on the use of negation. Chapter 4 presents the semantic theory of the logic used in this work. A recursively defined predicate is assigned a model that is a least-fixed point extension of the logic, providing the recursive definition meets a syntactic restriction similar to stratification. Unlike

www.manaraa.com

most work in the logic programming community, negation is given its classical meaning.

The Compilation Process

As adumbrated above, this work describes an implementation of programs specified in logic that is very different from the traditional implementation logic programming languages, such as Prolog. A large portion of the current work explains a compilation process that turns a program specification into an executable unit of code in a conventional language, of which C is taken as the prime example. The compilation process is sketched in Figure 1.

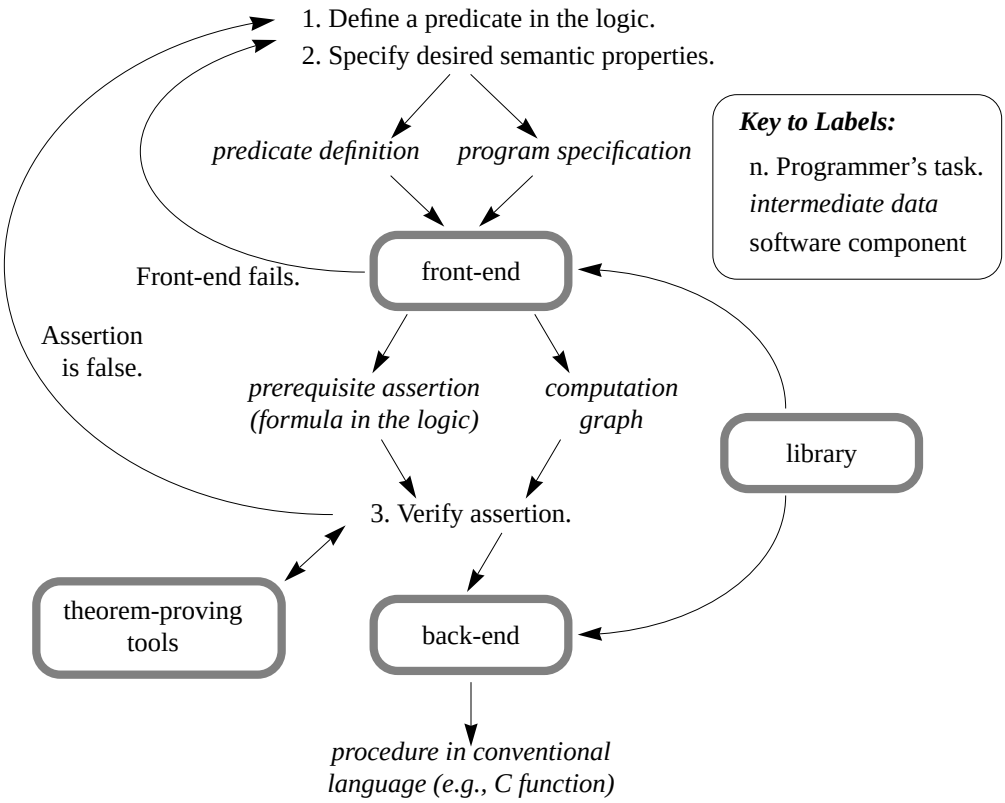


Figure 1: The compilation process.

The process begins with the programmer defining a predicate that states the post-condition for a desired program. The programmer then forms a program specification that references the predicate as the program's postcondition. The

www.manaraa.com

program specification also specifies the program's precondition, which of the predicate's arguments are intended to receive input values and which are intended to produce output values,[2] and a list of the desired *execution properties* for the program. Execution properties are requirements such as whether the program should produce all sets of output values that satisfy the predicate or only one such set, and whether the program must terminate if the predicate cannot be satisfied.[3]

A front-end compiler parses the predicate definition, determines if it satisfies the syntactic restrictions, and builds from the program specification an intermediate representation of a procedure that realizes the predicate. This procedure is a composition of (1) procedures that are part of a presupplied stock, and (2) procedures that were created from the earlier, successful compilation of other programs. The front-end relies on a *calculus of procedure composition*. This calculus has a set of composition rules each of which composes a procedure that realizes a logical construct — e.g., $\theta \Rightarrow \varphi$ — from procedures that realize the constituent formulae — e.g., θ and φ. For each such composition, the calculus derives (1) execution properties for the composed procedure, and (2) a precondition that qualifies the input set for which the composed procedure realizes its postcondition. The precondition for the final procedure must be entailed by the precondition in the program specification. This entailment is the *prerequisite assertion* for the compiled program. In other words, the front-end compiler generates an intermediate representation of a procedure and a meta-theorem that says "this procedure satisfies the program specification if the *prerequisite assertion* α holds," where α is a closed formula in the logic.

At this point, a traditional compiler would produce code in the target language from the intermediate representation. In our scheme, there is little point in producing code unless its execution will satisfy the program specification, and this is known to be the case only if the prerequisite assertion generated by the front-end is a theorem in the logic. Before code is produced, the programmer must verify

---

[2] In the logic programming community, the direction of data flow through a variable is called the variable's *mode*.

[3] Chapter 3 defines the execution properties and describes program specification.

www.manaraa.com

the prerequisite assertion. If this assertion is not true, the programmer must revise the program specification. Ideally, the programmer will work with a suite of theorem proving tools that help verify assertions or detect how they fail. The kind of theorem proving tools that are needed, and how they should be integrated into the programming environment, are important issues that lie beyond the scope of this dissertation.

Once the prerequisite assertion is verified, the back-end compiler produces code from the intermediate representation. The recursive definition of the new predicate, the program specification, and the newly produced program are stored in a library, so that future programs can be built on the ones previously compiled. (The data flow into the library is not shown in Figure 1.)

There are two places where the programmer can be stymied in trying to compile a program. First, the front-end may not find a composition of procedures that realizes the program's predicate. This occurs when the predicate is defined too abstractly in the logic. For example, the logic includes a primitive predicate, *data structure isomorphism*, that can be used to specify the deep copy of arbitrarily complex data structures. No compiler can automatically produce code that performs general data structure copies. When the compiler fails, the programmer must redefine the predicate in terms that are computationally simpler, though logically equivalent. Given the above example, instead of merely asserting isomorphism between two data structures, the programmer would write a longer, logically equivalent, probably recursive, predicate that defines relationships between the *component parts* of the two data structure instances. In essence, this tells the front-end: this is how to copy this kind of data structure. Such rewriting is always possible, because the compiler can generate code for the basic operations on the primitive data structure sorts.

Second, the programmer can also be stymied after the front-end produces the prerequisite assertion. If the assertions is false, it indicates that the program involves operations whose correctness depends on run-time conditions. Two examples are ubiquitous. First, to subscript an array, the subscript expression must be within array bounds. Second, to dereference a pointer, the pointer must

www.manaraa.com

reference a component of a passed data structure. If the prerequisite assertions are not theorems in the logic, the programmer must either rewrite the predicate so that the assertions are tested at run-time, or the programmer must make these assertions part of the precondition for the program. In the latter case, the prerequisite assertion becomes an assumption about the data structures against which the program may execute. The front-end takes into account the preconditions of component procedures that compose into the current compilation, so that these are guaranteed either by the nature of the composition or by the precondition of the composed procedure.

It is worth noting that the semantic theory and compilation process presented in this work are largely independent of the primitive functions and predicates that are defined in the logic, and also of the compiler's target language. A primitive predicate is built into the system through axioms that logically define the predicate and library procedures with known execution properties that realize the predicate. (The axioms make the primitive predicate known to the theorem-proving tools, while the library procedures give the compiler's front-end a way to use the primitive predicate in composing larger programs.) The compiler's back-end isolates the target language in the usual fashion. This work focuses on data structure programming, and targets C as an example, but as suggested earlier, the general framework may be useful to other application domains.

1.2 Related Work

There are three kinds of related work. First, researchers in automatic programming have investigated the general question of turning high-level specifications into executable code. Second, there has been a variety of work on the representation and verification of data structures. Third, there is some relevant work on the optimization of logic programs in their more traditional guise. Each of these three areas is discussed below. The current research is positioned as a novel kind of automatic programming.

www.manaraa.com

Automatic Programming

There is a body of work on *automatic programming*, that is, the generation of programs from high-level specifications through automatic and semi-automatic transformation and refinement of the specification. Because the research presented here concerns the generation of programs from high-level specifications, it falls into this genre, though as will be described shortly, there are some aspects that set it apart. In [44], Mostow presents a scheme for classifying different automatic programming systems and surveys several such systems. Mostow describes this classification scheme as follows:

- *Scope:* What kinds of software are addressed?
- *Power:* How much is automated?
- *Level:* What part of the route from informal requirements to machine language is addressed?
- *Purpose:* What parts of the software lifecycle are addressed?
- *Knowledge:* What kinds of knowledge are explicitly used by the system?

The specification languages used by these systems support the manipulation of data at an abstract level. At some point during the transformation from high-level specification into executable program, these abstract data types and operations must be realized as concrete data structures and procedures that operate on them. Some of these systems support only simple data types for which code can be automatically generated without much difficulty. The systems that support more complex data types are more interesting to our purpose. Of these latter, the method each uses to refine its data types can be placed into one of three categories.

(1) *Programmer selection from data structure library:* Interactive annotation is used to select concrete data structures to represent the high-level data types. This is the route taken in KIDS [60].

(2) *Automatic selection from data structure library:* The automatic programming system supports a particular application domain, and it uses knowledge of this domain to automatically select data structures from a stock set. This is

www.manaraa.com

exemplified in ELF [59], which supports the creation of VLSI design programs, and ϕ**NIX** [6], which supports the creation of applications related to oil well logging.

(3) *Hand-programming of data structures:* The programmer is given a way to extend the kinds of data structures that are targeted. For example, in KBEmacs [67], the programmer can add new *cliches* that deal with new kinds of data structures.

None of these methods supports the creation of verified code for general data structures. (1) and (2) rely on data structure libraries that are written and tested prior to their use in the automatic programming system. These systems do not address how these libraries are created nor how they are tested. (3) permits the programmer to build data structure procedures, but does not support their verification.

This suggests that the construction of verified data structure libraries is an appropriate domain for automatic programming. From the perspective of automatic programming, that precisely describes the purpose of the current work. Applying Mostow's categorization scheme, our research is summarized as an experiment in automatic programming with the following characteristics.

- *Scope:* Data structure libraries.
- *Power:* Automatic code generation, but reliance on theorem-prover or programmer interaction to verify necessary assertions.
- *Level:* From specifications in first-order logic to code in C.
- *Purpose:* Creation and verification of component libraries.
- *Knowledge:* Characteristics of addressed memory, as expressed in the logic's axioms and embedded in the calculus of procedure composition.

Representation of Data Structures

The tension between formal neatness and execution efficiency in the representation of data structures has arisen in several fields. In the field of logic programming, Kifer [30, 31] and Beeri [7] extend Horn clause logic to include more complex kinds of data than is available in Prolog, the first with objects and

www.manaraa.com

the second with recursive record structures. These extensions do not provide the expressiveness of C, and neither researcher is concerned with execution efficiency to the degree with which it is treated in the current work.

The algebraic specification of abstract data types, which directly addresses this problem, enjoys a rich body of literature. The paper by Guttag [23] is fundamental. But again, algebraic specification does not provide the desired expressive power, especially with regard to pointer reference. Furthermore, efficient implementation remains an open issue.

Hoare [26] wrote on using recursive structures instead of pointer relationships, arguing that pointers had bad semantic properties similar to the `goto` statement. General recursive data structures are formalized using graph grammars, and researchers such as Pratt [54, 55], Engels [17], and Nagl [45] have applied these to software development, with Pratt especially focusing on the representation of data structures. There are two problems. First, it is hard to program graph grammars. Second, in most of the representations, it is not clear that the various ways of programming graph grammars support reasoning about their programs.

Optimization of Logic Programs

The third kind of related work concerns those researchers who have tried to improve the execution efficiency of logic and applicative programming languages, especially those concerned with destructive update. Nikhil [51] and Miwelski [42] have both investigated destructive updates in the applicative framework. Nam and Henschen [46] turn a certain class of Prolog programs into procedural code.

The research on constraint programming [5, 36, 64] largely investigates the efficient execution of certain classes of logic programs, though it is not always couched in these terms. (Curiously, straddling both sides of the fence, Montanari [43] uses graph grammars to solve constraint satisfaction problems.) The research into constraint programming is related to our research through its concern with

www.manaraa.com

determining a static data flow that will solve a logical query. Research into optimizing Prolog execution also touches on this. The work of Warren [66] and the work of Van Roy and Despain [65] are examples.

1.3 Plan of the Dissertation

This work may be explored in several ways. A computer scientist whose interest has been sufficiently piqued to delve further might first want to take the quickest route from logic to C code. This is the core track shown in Figure 2. This reader will peruse Chapter 2 to become acquainted with the logic, but might only casually examine the theoretical details, such as the axioms that are presented. Chapter 3 describes how programs are specified and gives several example programs. The specification language, which includes the logic as its major part, is called Galois.[4] Chapter 5 describes the calculus of procedure composition and provides the compilation algorithm that generates a procedure composition from a program specification. The first example in Chapter 8 shows the working of the compilation algorithm in a simple case and then meticulously steps through the production of C code for this case.

[4] It is named after the French mathematician Evariste Galois, whose youthful exuberance gave us an important part of number theory, but also led to his unfortunately early demise.

www.manaraa.com

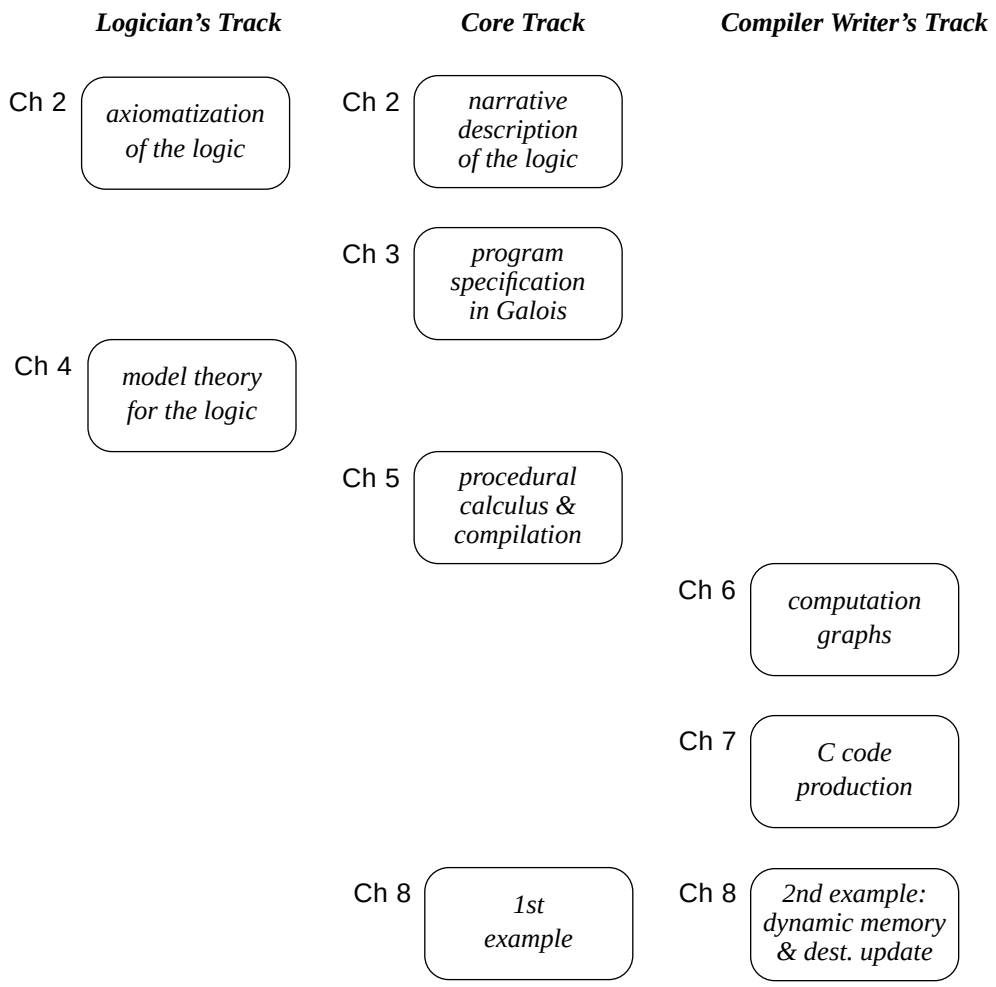


Figure 2: Plan of the dissertation.

A reader who has a logical bent or who is interested in putting the logic into a theorem prover will return to Chapter 2, to examine more carefully the details of the logic's axioms and the presentation of its intended model. Chapter 4 describes how the logic's intended model is extended for recursively defined predicates that satisfy certain syntactic restrictions.

The reader who is interested in the details of compilation will return to Chapter 6 and Chapter 7. Chapter 6 describes a graphical intermediate representation for programs. It presents an algorithm that analyzes the deallocation of memory required by a specified program. It also shows how the graphical representation can be used to display a program's prerequisite assertion in an

www.manaraa.com

intuitive fashion, helping the programmer to understand its origin and to understand how to modify the program specification when this is necessary. (The computation graph and the procedure composition from Chapter 5 provide equivalent information, but the former is more useful for certain kinds of program analysis and display.) Chapter 7 describes the production of C code. For the most part, it works from the procedure composition of Chapter 5, but there are some references to Chapter 6. These are easily skipped on a first reading. The second example in Chapter 8 presents a program that works on a dynamically allocated data structure. This example is used to illustrate the memory deallocation algorithm from Chapter 6.

The research described in this dissertation is largely conceptual and theoretical, though the author hopes that the concepts are interesting, that the theory is sound, and that the current work will see a practical fruition that provides real benefit. As usual, the last chapter provides a conclusion, and discusses future research. Of special note is work now in progress by three colleagues who are supplying the major effort in implementing a prototype programming environment based on Galois. Mohan Kumar has produced the core of an initial compiler. Patrick Ray is working on the user interface and is investigating what programming methodologies are appropriate to Galois. Bhalchandra Ghatate has put the logic's sorts and its axioms into a form acceptable to the Boyer-Moore theorem prover. Using this theorem prover, he has proved the prerequisite assertions that are generated by the compiler on some examples. This continuing work will undoubtedly find where the real problems lie with the theory and algorithms described here, leading to the inevitable revision of theory based on practice.

In each chapter, figures, definitions, lemmas, and theorems are numbered from the same sequence, e.g., a chapter with Figure 1, Lemma 2, and Figure 3 will not have a Figure 2. Lemma 2 lies between Figure 1 and Figure 3. A reference to a figure, definition, lemma, or theorems always refers to the chapter in which the reference occurs unless explicitly stated otherwise. The Glossary will help the reader with how terms are used in this work.

www.manaraa.com

2. The Logic

A logic is no more than a language that is made formal through a syntax that can be automatically recognized, rules of inference that can be automatically applied, and a recursive set of assumptions, called axioms, about the domain of discourse. In creating a logic, the hard part is finding a formalization that fits the intended purpose. What *things* must be expressible as terms in the logic? What *concepts* must be expressible as predicates in the logic? What *propositions* must be expressible as sentences in the logic? What *knowledge* must be captured in the axioms?

The logic described in this Chapter is meant to formalize discourse about data structures, the kind of discourse that passes between C programmers discussing a program, or that one reads in programming texts such as Knuth's [32]. The following English sentences are examples of statements that can be formally expressed in the logic.

> "A linked list is a set of nodes such that (1) every node has a pointer labeled **`next`**, (2) there is a distinguished head node and tail node, (3) every node in the set can be reached by chasing the **`next`** pointer from the head node, and (4) the last node has a null **`next`** pointer."
>
> "These two B-trees are the same."
>
> "The insert operation generates a B-tree that has the following relationship to the input B-tree: ..."
>
> "This data structure instance has unreachable parts."
>
> "This data structure instance has dangling pointers."
>
> "The result of the insert operation satisfies the invariants for a B-tree."

The axioms of the logic capture basic knowledge about data structures. Two examples are expressed in English below.

> "If two data structure instances in the same memory state have the same address, then they are either identical, or one is an initial part of the other."
>
> "There is no data structure instance whose address is **`NULL`**."

www.manaraa.com

In short, the logic is meant to provide a formal language in which one can define classes of data structures, define input-output relationships for operations on data structures, and prove things about these classes and relationships. This chapter provides a formal description of the logic. First, it places the logic within the range of traditional logics. It then discusses the syntactic categories — the sorts — of the logic. These are the things that data structures are made of: characters, integers, floats, addresses, arrays, structures, and sets. The last half of the chapter provides and discusses the axioms that define the various base concepts that are needed, and that capture the necessary knowledge about data structures. (On first reading, the reader might want to only peruse the actual axioms, to return to them on a second reading, or when chasing a reference from later chapters.)

To the extent possible, much of the syntax follows that of C. If `x` is a data structure, `&x` is its address; if `x` is an array, then `x[1]` is the second element; and if `x` is a structure with a field `color`, then `x.color` is a reference to that field in `x`. Because these are all first-rate terms in a traditional, first-order logic, they follow rules that are somewhat different from those of C. But the resemblance is much more than syntactic gloss! As described in Chapter 3, the logic is used to write programs that compile to C, and this compilation (described in later chapters) preserves the portions of the logic that share the syntax of C.

2.1 Preliminaries

Begin with the sorted, first-order predicate calculus with equality, as described, for example, in [19]. `True` and `False` are 0-ary predicates (boolean constants). Include the usual logical participles: negation ($\sim$), the boolean connectives $\wedge$, $\vee$, and $\Leftrightarrow$ for conjunction, disjunction, and equivalence, and the universal ($\forall$) and existential ($\exists$) quantifiers. The two symbols $\Rightarrow$ and ∇ form a ternary boolean connective defined as follows: $\theta \Rightarrow \varphi \nabla \phi$, read "if θ then φ else ϕ," is equivalent to $(\theta \wedge \varphi) \vee (\sim\theta \wedge \phi)$. The formula $\theta \Rightarrow \varphi \nabla$ `True` is abbreviated to $\theta \Rightarrow \varphi$. Parentheses are used in the normal fashion to associate formulae. When parentheses are absent, association is derived by the precedence: $\sim$, $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$ ∇, $\exists$,

www.manaraa.com

∀. For convenience, a quantifier can apply to a list of variables separated by commas. Thus, `(∃x,y,z)` is short for `(∃x)(∃y)(∃z)`.

As much as possible, this work follows the terminology and conventions of traditional logic. A *term* is a constant, a variable, or the application of a function to other terms. An *atom* is the application of a predicate to terms. The terms to which a function or predicate is applied are called its *arguments*. A *positive literal* is an atom, and a *negative literal* is the negation of an atom. A *formula* is a literal, a quantified formula, or formulae combined with the boolean connectives. Nested quantifiers may not share quantified variables, and if a variable is quantified in a formula, then everywhere it appears must be within the scope of one of its quantifiers. A variable is *bound* in a formula if it appears only within the scope of its quantifiers, otherwise it is *free*. A logical *object* is a constant, function, or predicate. The *signature* of an object is its sort together with the sorts required of its arguments, if any. The sort of a term is the sort of its outermost function. In forming terms, the arguments of a function or predicate must match in number and sort the object's signature. (Note, though, that function and predicate symbols are sometimes overloaded. When it is said, for example, that equality applies to any two terms of the same sort, the reader is expected to understand that formally there is an equality predicate for each sort.)

The logic includes an infinite hierarchy of sorts. This hierarchy is formed from the repeated application of a few *sort constructors* to a few *base sorts*. A *sort expression* identifies a sort by name or by the method of its construction. Sort expressions are *not* formulae in the logic; rather, they serve to type terms in the logic. The next section describes the sort hierarchy.

Formally, a sorted logic has disjoint name spaces for constants, variables, functions, and predicates, and the name of each object tells its sort and the number and sort of its arguments. In practice, this would require too awkward a segmentation of the only name space that is actually available, to wit, character strings. Like C, an *identifier* is any string of letters (including the underscore) and digits, beginning with a letter. Identifiers are pressed into service as variable, function, and predicate symbols, and as sort names. The sort of a variable is

www.manaraa.com

declared explicitly at the beginning of its lexical scope, which is either the first time it appears as a free variable, or where it is quantified. A *sort declaration* has the form **`x:sexp`** or **`sexp x`** where **`x`** is the name of the variable and **`sexp`** is a sort expression.

Three of the base sorts are **`integer`**, **`character`**, and **`floatingPt`**. The integers and floating point numbers have the usual functions for addition (+), subtraction (–), and multiplication (*), and the binary predicates **`<`**, **`>`**, , and provide the usual order. The domain of the character sort is a finite alphabet, Γ. A character constant is written as in C, for example, **`'a'`** and **`'Z'`**. Characters have a lexical order, and the symbols **`<`**, **`>`**, , and are overloaded for this purpose, also.

The symbol for equality is ≅. The symbol = is used for a weaker, but more frequently used, predicate that is true if two data structure instances are equal in their data, but not necessarily in their addresses. (Data structures are explained below.)

2.2 Deductive Apparatus

An axiomatization of the integers along standard lines is assumed, for example, the system Q= in [27]. Any first-order axiomatization of the integers is semantically incomplete, but because this work does not stray into arcane mathematics, it is unconcerned with assertions that are independent of the traditional first-order axiomatizations. The presentation assumes axiom schema that permit induction of all formulae based on the natural numbers.

This work also assumes a first-order axiomatization of floating point numbers that includes the appropriate functions for them. Such axiomatization is clearly dependent on the characteristics of the floating point machine of concern. Fortunately, the problems of data structures are not tightly coupled to the issues and algorithms of floating point arithmetic. In what follows, nothing is assumed about floating point numbers except that they can be tested for equality and have an order.

www.manaraa.com

The lexical order of characters is trivially axiomatized with a finite number of axioms of the form **'a'<'b'**, an axiom for the transitive closure of **<**, and axioms defining **>**, , and in terms of **<**. This work assumes nothing about characters except for equality and order.

The focus of this work is how the base sorts are composed into more complex structures. The characteristics of the integers are important, because they are used to represent lengths and offsets, and because they provide the basis for induction. Floating point numbers and characters are carried along as uninterpreted data, though the availability of identity and order makes them available as keys. Floating point functions are needed so that programmers can lay floating point algorithms on top of data structures, but this work makes no assumptions about the behavior of these functions. Axioms for the complex sorts are given in the following sections of this chapter.

2.3 The Sorts

In addition to **integer**, **float**, and **character**, there is one other base sort. The domain of the **address** sort is a denumerable set, each element of which can be viewed as a unique tag. There are no functions that apply to this sort, and equality is the only relevant predicate. The constant **NULL** is a distinguished address value. The base sorts are listed below.

Sort Name	*Domain*
integer	The usual integers.
floatingPt	Floating point numbers.
character	A lexically ordered, finite alphabet, Γ.
address	A denumerably infinite set of tags.

An infinite hierarchy of sorts is constructed from these base sorts. This hierarchy is shown in Figure 1, below. The explanation of the sort hierarchy will follow the paths in this diagram.

www.manaraa.com

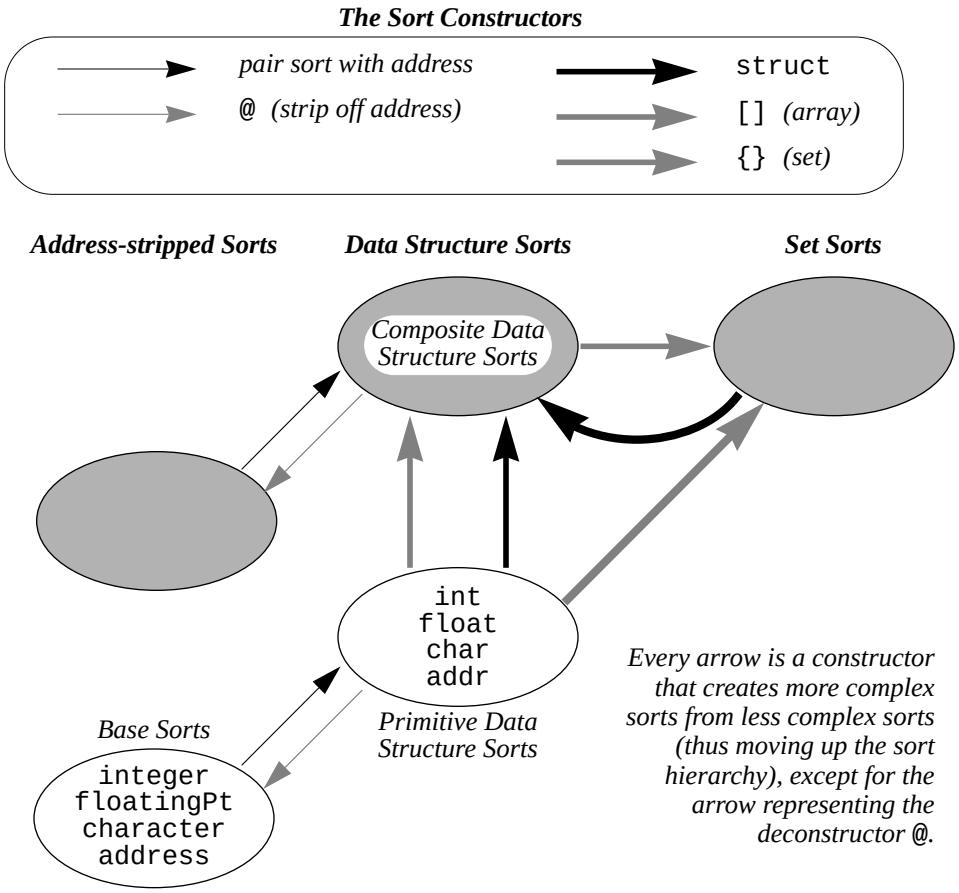


Figure 1: The sort hierarchy.

The least elements in the sort hierarchy are the base sorts. They are shown in the lower left oval of Figure 1. These do not count as even primitive data structures, because they lack an address.

An element in a data structure sort is an ordered pair $<a, d>$. The first value, a, is an **`address`**, and it is called the *address value*. The second value, d, is called the *data value*. Unary functions "**`&`**" and "**`@`**" apply to data structure sorts, projecting the address and data values. Thus, if **`x`** is a variable of a data structure sort, **`&x`** is the address value of **`x`**, and **`@x`** is the data value of **`x`**. The primitive data structure sorts are created by pairing an address with each of the base sorts. These are listed below, and are shown in the lower, middle oval of Figure 1.

www.manaraa.com

Data structure sort	*Data value*
int	**integer**
float	**floatingPt**
char	**character**
addr	**address**

The sort names **int**, **float**, and **char** and the address projection function **&** are purposely pulled from C. As there, if **x** is an **int**, **&x** is an address, and the value of **x** is an integer.

Two other sort constructors resemble type constructors available in C.[1] These are the array constructor and the **struct** constructor. They both define a data structure sort in terms of one or more other data structure sorts.

Given any data structure sort expression **Sexp**, **Sexp[]** is also a data structure sort expression, called the array of **Sexp**. **Sexp** is called the element sort of the array. There are three basic functions on arrays: **len()**, which returns the array length; an indexing function, which returns a value in the element sort; and a function that extracts subarrays. If **x** is an array of **Sexp**, declared **x: Sexp[]**, and **t**, $\mathbf{t_0}$, and $\mathbf{t_1}$ are integer terms, these functions are written as shown below.

Function	*Notation*	*Sort*
array length	**len(x)**	**integer**
element projection	**x[t]**	**Sexp**
subarray projection	$\mathbf{x[t_0..t_1]}$	**Sexp[]**

In the last function, if $\mathbf{t_0}$ is omitted, it is assumed to be zero, and if $\mathbf{t_1}$ is omitted, it is assumed to be **len(x)-1**. A small abuse of notation is introduced to make the syntax more like that of C. Instead of **x:Sexp[]** or **Sexp[] x**, it is permissible to write **x[]:Sexp** or **Sexp x[]**.

The **struct** constructor aggregates a finite set of values, and allows their selection by *field name*. It optionally associates a name with the sort. The syntax and meaning follows C. For example, the sort expression below defines a structure that has an address, an array of characters, and an integer, and associates

[1] This resemblance is intended. In implementation, instances of these sorts are represented by instances of the corresponding C data types, as described in Chapter 3.

www.manaraa.com

the name **WordNode** with this sort.

```
struct WordNode {
   addr Next;
   int ChrCount;
   char Word[];
}
```

The field names are postfix functions on the sort. Thus, if **x** is a variable of sort **WordNode**, declared **x:WordNode** or **WordNode x**, then **x** is available to the functions **Next**, **ChrCount**, and **Word**, and the application of these functions to **x** is written **x.Next**, **x.ChrCount**, and **x.Word**. These terms have the sort **addr**, **int**, and **char[]**, respectively.

Instances of the composite data structure sorts — array and **struct** sorts — are ordered pairs, each comprising an address value and a data value. As with the primitive data structure sorts, the unary functions "**&**" and "**@**" project these values. As always, **&x** is an **address**. If **x** has the sort **Sexp**, then the sort of **@x** is written **@Sexp**. If **Sexp** is an array or **struct**, **@Sexp** is the only way to express the sort of its data value. Used in sort expressions, the operator **@** is a sort deconstructor, as shown by the downward arrows of Figure 1. Since **@** just returns the sort of the data value, **@int** is **integer**, and similarly **@** returns the base sort of the other primitive data structure sorts. The sorts given by the **@** operator are collectively called the *address-stripped sorts*.

The logic is also concerned with data structures whose elements are scattered in memory, tied together only through pointer reference. To describe these kinds of data structures, a *set* constructor[2] is used. Syntactically, the set constructor works like the array constructor, except that curly braces, **{}**, are used instead of square ones. Also like an array, a set aggregates a finite number of values from some data structure sort. Unlike an array, a set is not indexed, and a set does not have an address for the set as a whole. For example, if **x** is an instance of **int{}**, **x** could comprise three **int** values, with no order specified.

---

[2] The traditional problems of mathematical set theory do not arise, because sets are stratified by the sort hierarchy. The intended model has only finite sets.

www.manaraa.com

As usual, the predicate for set membership is written **a∈x**. The sort of **x** must be **Sexp{}**, where **a** has the sort **Sexp**, called the element sort. The element sort of a set must be a data structure sort. (Sets of sets and sets whose elements have a base sort are not allowed.) The basic set functions are listed below.

Function	*Notation*	*Sort*
set cardinality	**|x|**	**integer**
set construction	**{a, b}**	**Sexp{}**
set union	**x∪y**	**Sexp{}**
set intersection	**x∩y**	**Sexp{}**
set difference	**x-y**	**Sexp{}**

Sets are not data structures. But a set can be included as a field in a **struct**, and through this, become part of a data structure sort. Every **struct** must have at least one field that is not a set. The inclusion of a set in a data structure is conceptually important. As the next chapter will discuss, it provides the programmer a way to express what non-contiguous pieces belong with a data structure, and through this, how memory is allocated and released. A set's inclusion in a **struct** does not constrain how the set's elements are laid out in memory. (Thinking close to the implementation level, a programmer should view the layout of a **struct** as unchanged if set fields are added or deleted. Data representation is discussed in Chapter 3.)

Sets complete the traversal of the sort hierarchy as shown in Figure 1. In that figure, every arrow except that for the deconstructor **@** creates a more complex sort whose instances are aggregates of instances of simpler sorts. Thus, the hierarchy is one of *composition*. (It is *not* an abstraction or inheritance hierarchy.) Domains for the sorts are discussed below. Later sections give the axioms that characterize these domains and the relationships between data structure instances, their parts, and their addresses. These axioms will reflect much of a programmer's intuitive understanding about how data structures and arrays are laid out in memory.

The array, structure, and set functions described above are those that are common for these constructs. Section 2.7 describes some functions for modifying parts of data structures and sets.

www.manaraa.com

It is important to keep straight the role played by sorts in a logic. The sorts are *not* defined in the logic. Rather, they are a part of the logic's definition. They syntactically characterize terms, and semantically characterize the intended domains. When a variable's sort is declared, for example, `pt: struct {int V; int H;}`, this does *not* define a new sort. The unique sort that has the postfix functions `V` and `H` returning `int` values, and no other postfix functions, is permanently part of the sort hierarchy. The declaration merely says that the variable `pt` has this sort, and such declaration is needed only because it is too cumbersome to reserve disjoint sets of variable names to each sort. Associating a sort name with a `struct` definition is merely a notational convenience that saves repeatedly writing out the structure definition.

2.4 The Intended Model for the Logic

The standard integers are the intended domain for the `integer` sort. A finite character alphabet and a set of floating point numbers convenient to the implementation are the intended domains for the `character` and `floatingPt` sorts. The natural numbers and the special value `NULL` form the intended domain for the `address` sort. These are the *base domains*.

Structures and arrays aggregate simpler data values sequentially. Sets aggregate simpler data values without regard to sequence. The intended domains for the sort hierarchy are created from finite lists and sets, using the domains above for the base sorts. Denote a list with angle brackets — $<a, b, c>$ — and a set with curly brackets — $\{a, b, c\}$. Sequence is significant in the first but not the second. Thus, $\{a, b\} = \{b, a\}$, but $<a, b> \neq <b, a>$. The *intended universe* is the smallest collection that satisfies the following recursive definition. (1) The empty set and the empty list are in the intended universe. (2) Elements of the base domains are in the intended universe. (3) Every finite list of and finite set of elements in the intended universe is also in the intended universe. The narrative below carves this universe into domains for the sorts. It also gives an intended interpretation to the functions discussed so far.

www.manaraa.com

The sort **int** has as its intended domain all ordered pairs $<a, \chi>$ where a, the address value, is an address other than **NULL**, and χ, the data value, is an integer. For **char**, **float**, and **addr**, the data value ranges over the character, floating point, and address domains, respectively.

The array sort **Sexp[]** has as its intended domain all ordered pairs of the form $<a, <e_0,...e_{n-1}>>$, where a is an address other than **NULL**, $e_0=<a, v>$ (if e_0 is present), each e_i is in the intended domain of **Sexp**, and no address value b appears in both e_i and e_j when i j. If **x** denotes $<a, <e_0,...e_{n-1}>>$, then **&x** denotes a, **len(x)** denotes n, and **x[i]** denotes e_i for 0 i<n. Similarly, the structure sort with fields $\mathbf{f_0},...\mathbf{f_n}$ whose sorts are $\mathbf{Sexp_0},...\ \mathbf{Sexp_n}$ has as its intended domain all ordered pairs of the form $<a, <e_0,...e_n>>$, where $e_i=<a, v>$ for $\mathbf{f_i}$ the first non-set field, each e_i is in the intended domain of $\mathbf{Sexp_i}$, and no address value b appears in both e_i and e_j when i j. If **x** denotes $<a, <e_0,...e_n>>$, then **&x** denotes a and $\mathbf{x.f_i}$ denotes e_i. The set sort **Sexp{}** has as its intended domain all finite sets $\{e_0,...e_{n-1}\}$, where each e_i is in the intended domain of **Sexp** and no address value b appears in both e_i and e_j when $e_i e_j$. If **x** denotes $\{e_0,...e_{n-1}\}$ and **y** denotes e_i, then **|x|=n** and **y**$\in$**x** hold.

Notice that each value in the intended domain of a data structure sort is an ordered pair whose first element is an address. If $<a, \chi>$ is a value in the intended domain of the data structure sort **Sexp**, then χ is a value in the intended domain of **@Sexp**. If χ is composite, i.e., if **@Sexp** is not one of the base sorts, then a is also the address of the first non-set component of χ.

Consider an **int** array **x**, of length two, whose first integer value is 53 and whose second integer value is 32. The value of the array **x** in the intended domain is a list, shown in the table below, where the a_i are address values. Other terms involving **x**, their sorts, and their values, are also shown.

term	*sort*	*value in intended domain*
x	**int[]**	$<a_0, <<a_0, 53>, <a_1, 32>>>$
&x	**address**	a_0
@x	**@int[]**	$<<a_0, 53>, <a_1, 32>>$
x[0]	**int**	$<a_0, 53>$

www.manaraa.com

&x[0]	**address**	a_0
@x[0]	**integer**	53

As a second example, consider the sort **struct Foo {int count; int values{};}**. An instance **y** of this sort is a structure whose first element is an integer, say 2, and whose second element is a set of integers, say {42, 53}. Some terms involving **y** are shown below.

term	*sort*	*value in intended domain*
y	**Foo**	$<a_0, <<a_0, 2>, \{<a_3, 53>, <a_2, 42>\}>>$
&y	**address**	a_0
@y	**@Foo**	$<<a_0, 2>, \{<a_3, 53>, <a_2, 42>\}>$
y.values	**int{}**	$\{<a_3, 53>, <a_2, 42>\}$

Axioms that characterize the logic are given in the remainder of this chapter. The above domains and interpretation of functions, together with the usual understanding of equality on sets and lists, satisfy these axioms, and thus provide a model of the logic. Models of the logic are discussed in more detail in Chapter 4.

2.5 Axiomatic Characterization of the Sorts

The axioms below characterize the compositions that form the sort hierarchy. These axioms describe when composed sorts are equal, and they describe the behavior of the functions that project the sorts' components. We assume that the base sorts are already axiomatized. (Remember that the symbol ≅ is used for strict equality.)

For each data structure sort, characterize equality and forbid **NULL** addresses:

(1) `x≅y ⇔ &x≅&y ∧ @x≅@y`

(2) `&x NULL`

For each array sort, characterize equality and define subarrays:

(3) `len(x) 0`

www.manaraa.com

(4) `@x≅@y ⇔`
`len(x)≅len(y)`
`∧ (∀i:integer)(0 i<len(x)⇒ x[i]≅y[i])`

(5) `0 j k<len(x) ⇒ len(x[j..k])=(k–j+1)`
`∇ len(x[j..k])=0`

(6) `(0 i<len(x[j..k])⇒ (x[j..k])[i]=x[j+i]`

For each structure sort, with fields $\mathbf{f}_0$, ... $\mathbf{f}_n$, characterize equality

(7) $\texttt{@x≅@y} \Leftrightarrow \texttt{x.}\mathbf{f}_0\texttt{≅y.}\mathbf{f}_0 \wedge \ldots \wedge \texttt{x.}\mathbf{f}_n\texttt{≅y.}\mathbf{f}_n$

For each set sort with element sort **Sexp**, and every singleton **{a}**, characterize equality and the set operations:

(8) `x≅y ⇔(∀z:Sexp)(z∈x ⇔ z∈y)`

(9) `(∀z:Sexp)(z∈x∪y ⇔ (z∈x ∨ z∈y))`

(10) `(∀z:Sexp)(z∈x∩y ⇔ (z∈y ∧ z∈y))`

(11) `(∀z:Sexp)(z∈x–y ⇔ (z∈x ∧ ~z∈y))`

(12) `|x| 0`

(13) `|EMPTY| = 0`

(14) `|{a}| = 1`

(15) `|x∪y| = |x|+|y|-|x∩y|`

Like most of the axioms that will follow, the above axioms are actually axiom schema. Each axiom is repeated for each sort to which it applies. The intended universe trivially satisfies the above axioms, when the functions are interpreted as described in Section 2.4.

The next set of axioms define several predicates that relate composite data structure instances to their parts. These axioms formalize the important notion of a data structure part being *reachable* by offset calculation and pointer chasing. These predicates can be organized according to their logical strength, indicated by the arrows in Figure 2. (One predicate is said to be *logically stronger* than a second if it entails the second on the same arguments.)

www.manaraa.com

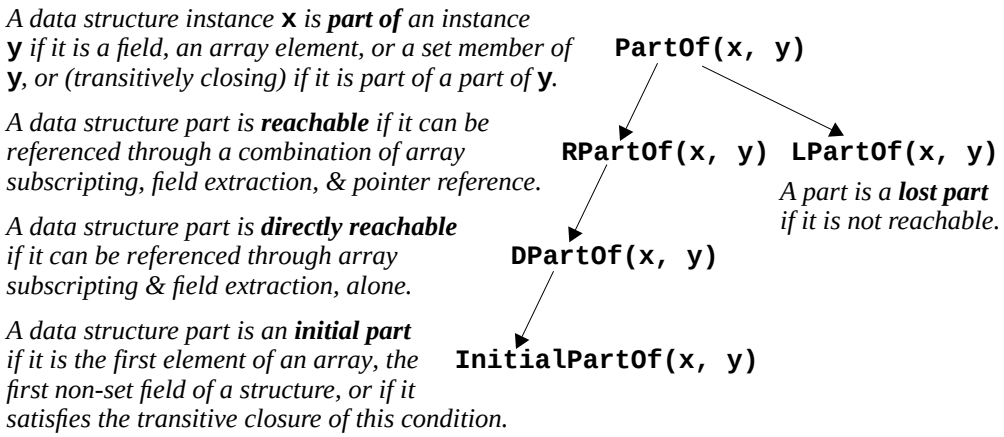


Figure 2: The parts predicates for data structures.

The axioms for these predicates are given below. For one data structure instance **x** being a part of a second instance **y** (a notion formalized in the predicate **PartOf(x, y)**), an initial part of another (**InitialPartOf(x, y)**), and a part of another directly reached by offset calculation (**DPartOf(x, y)**), the axioms are organized by the sorts of the arguments. The predicate **RPartOf** extends the predicate **DPartOf** with reachability through pointer reference. A data structure part is *lost*, a notion formalized in the predicate **LPartOf**, if it is not reachable.

For **x** of data structure sort **Sexp**, and **y** of sort **Sexp []**:

(16) $\texttt{InitialPartOf (x, y)} \Leftrightarrow \texttt{0<len(y)} \wedge \texttt{x} \cong \texttt{y[0]}$

(17) $\texttt{PartOf (x, y)} \Leftrightarrow (\exists \texttt{i:integer})(\texttt{0 i<len(y)} \wedge \texttt{x} \cong \texttt{y[i]})$

(18) $\texttt{DPartOf (x, y)} \Leftrightarrow (\exists \texttt{i:integer})(\texttt{0 i<len(y)} \wedge \texttt{x} \cong \texttt{y[i]})$

For **y** of a structure sort, the first non-set field $\mathbf{f}_F$, all the fields **f**, **g**,...**h** of **y** that have the same sort as **x**, where Axioms (19) and (21) apply only if **x** is not a set:

(19) $\texttt{InitialPartOf (x, y)} \Leftrightarrow \texttt{x} \cong \texttt{y.}f_F$

(20) $\texttt{PartOf (x, y)} \Leftrightarrow \texttt{x} \cong \texttt{y.f} \vee \texttt{x} \cong \texttt{y.g} \vee \ldots \vee \texttt{x} \cong \texttt{y.h}$

(21) $\texttt{DPartOf (x, y)} \Leftrightarrow \texttt{x} \cong \texttt{y.f} \vee \texttt{x} \cong \texttt{y.g} \vee \ldots \vee \texttt{x} \cong \texttt{y.h}$

For every data structure sort (and for Axiom (23), every set sort):

(22) $\texttt{InitialPartOf (x, x)}$

(23) $\texttt{PartOf (x, x)}$

(24) $\texttt{DPartOf (x, x)}$

www.manaraa.com

PartOf subsumes membership for sets. For **x** of sort **Sexp** and **y** of sort **Sexp{}**:

(25) `PartOf (x, y) ⇔ x∈y`

For **x** and **y** with data structure sorts (or **y** a set sorts, for **PartOf**), where the sort of **x** is lower in the sort hierarchy than the sort of **y**, but none of the cases above:

(26) `InitialPartOf (x, y) ⇔`
`(∃z)(InitialPartOf (x, z) ∧ InitialPartOf (z, y))`

(27) `PartOf (x, y) ⇔ (∃z)(PartOf (x, z) ∧ PartOf (z, y))`

(28) `DPartOf (x, y) ⇔`
`(∃z)(DPartOf (x, z) ∧ DPartOf (z, y))`

For **x** and **y** where the sort of **x** is *not* lower in the sort hierarchy than that of **y**, nor where **x** and **y** have the same sort:

(29) `~InitialPartOf (x, y)`

(30) `~PartOf (x, y)`

(31) `~DPartOf (x, y)`

For **y** having a data structure sort, **x** having a set or data structure sort lower in the sort hierarchy, and every sort **Sexp** lower in the sort hierarchy than that of **y**:

(32)
```
RPartOf (x, y) ⇔ DpartOf (x, y)
    ∨ (∃p:addr)(∃u:Sexp)(
          DpartOf (p, y) ∧ p=&u
       ∧  PartOf (u, y) ∧ RPartOf (x, u)
       )
    )
```

For **y** having a data structure sort, **x** having a set or data structure sort lower in the sort hierarchy:

(33) `LPartOf (x, y) ⇔ PartOf (x, y) ∧ ~RPartOf (x, y)`

The parts predicates above can be used to formalize some common notions about data structure instances. A data structure instance is *navigable* if it has no lost parts. An address **a** *points into* a data structure instance if the instance has a part whose address value equals **a**. A data structure instance has a *dangling pointer* if it (1) has a part that is an address, and (2) the value of this address neither points into the data structure instance nor is **NULL**. Two data structure instances are *disjoint* if they have no common components. These notions are formalized in

www.manaraa.com

predicates defined below.

For **y** having a data structure sort, and every data structure or set sort **$Sexp_0, ... Sexp_n$**, lower in the sort hierarchy:

(34) $Navigable\ (y) \Leftrightarrow ($
$(\forall x{:}Sexp_0)(PartOf\ (x, y) \Rightarrow RPartOf\ (x, y))$
$\wedge\ \ ...$
$\wedge\ \ (\forall x{:}Sexp_n)(PartOf\ (x, y) \Rightarrow RPartOf\ (x, y))$
$)$

For **y** having a data structure sort, and every data structure or set sort **$Sexp_0, ... Sexp_n$**, lower in the sort hierarchy:

(35) $PointsInto\ (p, y) \Leftrightarrow ($
$p{=}\&y$
$\vee\ \ (\exists x{:}Sexp_0)\ (PartOf\ (x, y) \wedge PointsInto\ (p, x))$
$\vee\ \ ...$
$\vee\ \ (\exists x{:}Sexp_n)\ (PartOf\ (x, y) \wedge PointsInto\ (p, x))$
$)$

(36) $HasDangles\ (y) \Leftrightarrow (\exists p{:}\ addr)$
$(\ p\ NULL \wedge PartOf(p, y) \wedge \sim PointsInto(p, y)\ )$

For **x** and **y** having a data structure sort:

(37) $Disjoint\ (x, y) \Leftrightarrow (\forall p{:}\ addr)$
$(\ \sim(PointsInto(p, x) \wedge PointsInto(p, y)))$

The next chapter will show how invariants for data structure classes are defined in the logic. The above predicates are useful in describing desirable properties of data structures. It is generally good if all parts of a data structure are reachable, and if it has no dangling pointers. Thus, a programmer would want to prove in the logic that the invariants defined for a data structure type logically entail these desirable properties.

The axioms given so far place little restriction on address values. As described for the intended model, composite data structure instances have the property that the address of the composite instance equals the address of its first non-set component. Except for this, addresses do not occur multiple times within a single data structure instance. The axiom below characterizes these restrictions.

www.manaraa.com

For **x** and **y** having data structure sorts lower in the sort hierarchy than **z**:

(38) `(PartOf(x,z) ∧ PartOf(y,z) ∧ &x=&y) ⇒`
`(InitialPartOf(x,y) ∨ InitialPartOf(y,x))`

The axioms above characterize the intended model. The next few sections will present axioms that define other functions and predicates, but these axioms will not constrain the intended domains nor the interpretation of functions and predicates defined above.

2.6 Comparing Data Structures

Because the logic treats a data structure's address as a contained piece of information, if two data structures are equal in the logic, they must have the same address. They are the same in the strongest possible sense, for example, in the sense of Fortran equivalence. This notion of strict equality is not very useful for talking about data structures, where one usually wants to discuss instances that may occupy different memory locations. (For this reason, the symbol ≅ denotes strict equality, and the symbol = is reserved for a different predicate described below.) This section defines some weaker but more useful notions of data structure identity.

Data Equality

The predicate for *data equality*, **=**, holds of two values in a sort if they are equal, ignoring any address values they contain. The axioms below define data equality in terms of strict equality. It is defined first for the base sorts, and then for each of the data structure and set sorts.

Define =, for each base sort:

(39) `x=y ⇔ x≅y`

Define =, for each primitive data structure sort:

(40) `x=y ⇔ @x≅@y`

www.manaraa.com

Define =, for each array sort:

(41) `x=y ⇔ (`
`len(x)=len(y)`
`∧ (∀i:integer)(0 i<len(x)⇒ x[i]=y[i])`
`)`

Define =, for each structure sort with fields $\mathbf{f}_0$,... $\mathbf{f}_n$:

(42) $\texttt{x=y} \Leftrightarrow \texttt{x.}\mathbf{f}_0\texttt{=y.}\mathbf{f}_0 \wedge \ldots \wedge \texttt{x.}\mathbf{f}_n\texttt{=y.}\mathbf{f}_n$

Define =, for each set sort **`Sexp{}`**:

(43) `x=y ⇔ (`
`(x≅EMPTY ∧ y≅EMPTY)`
`∨ (∃u, v:Sexp)(u∈x ∧ v∈y ∧ u=v ∧ x-{u}=y-{v})`

The order predicates **`<`**, **`>`**, , and are extended to the data structure sorts **`char`**, **`float`**, and **`int`** by applying them just to the data values of these sorts. Unlike data equality, these are not percolated up the entire sort hierarchy.

The motivation behind reserving the symbol **`=`** should now be clear. Whether **`x`** and **`y`** have the base sort **`integer`** or **`character`**, or the data structure sort **`int`** or **`char`**, the formulae **`x=y`** and **`x<y`** have their customary meaning. The symbols = and ≅ have identical meaning on the base sorts. By convention, the latter is used only with data structure and set sorts, and can be read “equal in both data and location.”

Data Structure Isomorphism

Programmers have a common notion of two data structure instances being “the same” when their non-pointer data values are equal and when their pointer values correspond in the “right sense.” Operationally, this sameness can be viewed as equivalence under the transformation that preserves a data structure when it is relocated in memory and its pointers are updated accordingly. For example, if an element of a linked list is relocated in memory, then the pointer to it from the previous element must be changed to reference the new location. Given this tracking of the relocation by the pointers involved, the new data structure is “the same.”

www.manaraa.com

The informal description in the previous paragraph of what it means for two data structure instances to be "the same" appeals to a programmer's intuition. Unfortunately, this crucial concept is left informal in most texts on programming and programming languages. Consider the move from addresses to "links" made by Knuth, when discussing the representation of a hand of cards on pages 230-231 of [32]:

> "The memory locations in the computer representation are shown here as 100, 386, and 242; these could have been any other numbers as far as this example is concerned, since each card links to the next. ...
>
> "The introduction of links to other elements of data is an extremely important idea in computer programming; this is the key to the representation of complex structures. When displaying computer representations of nodes it is usually convenient to represent links by arrows, ... The actual locations 242, 386, and 100 (which are irrelevant anyway) no longer appear ..."

The "links" that Knuth discusses are an abstraction; they are pointers embedded in data structures as preserved under the appropriate transformation when data structures are relocated in memory. The programmer has developed the right intuition of this if the programmer can make programs that work and can discuss them with other programmers.

This notion of "sameness" is made formal in the logic as a binary predicate called *data structure isomorphism*. Two instances **x** and **y** of a data structure sort are isomorphic, written **x::y**, if they are the "same" in the sense discussed above, i.e., if a C programmer would declare them the same modulo the contingent location of their parts in dynamic memory. This predicate is important to programming in the logic, since it is heavily used to specify changes to data structures, as described in the next chapter.

There are several possible approaches to defining data structure isomorphism. The one used below makes use of two lists of pointers, which for two instances of a data structure sort must act as parallel indices, to wit, the i-th pointer from the first list and the i-th pointer from the second list (1) reference analogous parts of the first and second data structure instances, and (2) are equal to analogous pointers from the first and second data structure instances. A part of the

www.manaraa.com

first data structure instance is analogous to a part from the second data structure instance if it is obtained by using the same array subscripts and the same field references. The definition of "paired indices" begins the formalization of these concepts.

Paired indices are same length address arrays (sort **addr[]**) neither of which repeats values:

(44)
```
PairIdx (xIdx, yIdx) ⇔ (
      len(xIdx)=len(yIdx)
   ∧  (∀i,j: integer)( 0 i,j<len(xIdx)⇒ (
            (xIdx[i]=xIdx[j] ⇒ i=j)
         ∧  (yIdx[i]=yIdx[j] ⇒ i=j)
         )
      )
   )
```

The next few axioms define when data structure instances **x** and **y** of the same sort are analogous relative to paired indices for them. For instances of **char** or **int**, the paired indices must reference **x** and **y** from the same subscript, and **x** and **y** must carry the same data value. For the **addr** sort, the latter condition is different: a common subscript must select an address in the first index that equals **x** and a value in the second index that equals **y**.

Define "analogous relative to paired indices" for **x** and **y** of sort **int**, **float**, or **char**:

(45)
```
AnaP (x, y, xPtrs, yPtrs) ⇔ (
      PairIdx (xPtrs, yPtrs) ∧ x=y
   ∧  (∃i:integer)(
         0 i<len(xIdx)∧ xIdx[i]=&x ∧ yIdx[i]=&y
      )
   )
```

www.manaraa.com

Define "analogous relative to paired indices" for **x** and **y** of type **addr**:

(46)
```
AnaP (x, y, xPtrs, yPtrs) ⇔ (
   PairIdx (x, y, xPtrs, yPtrs)
∧  (∃i:integer)(
      0 i<len(xIdx)∧ xIdx[i]=x ∧ yIdx[i]=y
   )
∧  (∃i:integer)(
      0 i<len(xIdx)∧ xIdx[i]=&x ∧ yIdx[i]=&y
   )
)
```

The next three axioms extend the notion of a pair of data structure or set instances being "analogous relative to a pair of pointer lists" to structures, arrays, and sets.

Define "analogous relative to paired indices" for **x** and **y** in a structure sort with fields $\mathbf{f_0}$,...$\mathbf{f_n}$:

(47)
```
AnaP (x, y, xPtrs, yPtrs) ⇔ (
   AnaP (x.f0, y.f0, xPtrs, yPtrs)
∧  ...
∧  AnaP (x.fn, y.fn, xPtrs, yPtrs)
)
```

Define "analogous relative to paired indices" for **x** and **y** in an array sort **Sexp[]**:

(48)
```
AnaP (x, y, xPtrs, yPtrs) ⇔ (
   len(x)=len(y)
∧  (∀i:integer) (0 i<len(x)⇒
      AnaP (x[i], y[i], xPtrs, yPtrs)
   )
)
```

Define "analogous relative to paired indices" for **x** and **y** in a set sort **Sexp{}**:

(49)
```
AnaP (x, y, xPtrs, yPtrs) ⇔ (
   (∀u:Sexp)(u∈x ⇒
      (∃v:Sexp)(v∈y ∧ AnaP (u, v, xPtrs, YPtrs))
   )
∧  (∀v:Sexp)(v∈y ⇒
      (∃u:Sexp)(u∈x ∧ AnaP (u, v, xPtrs, YPtrs))
   )
)
```

www.manaraa.com

If a pair of sets are analogous relative to paired indices, then there is a bijective mapping between the sets. The indices match elements from the two sets according to shared index subscripts. Two data structure instances are isomorphic if there exist a pair of indices with respect to which they are analogous.

Define data structure isomorphism, for each data structure sort:

(50)
```
x::y ⇔ (∃xPtrs, yPtrs: addr[])(
   AnaP (x, y, xPtrs, yPtrs)
)
```

Note that analogous pointers from **x** and **y** must simultaneously (1) equal **NULL**, (2) reference analogous parts of **x** and **y**, or (3) fail to reference any part of **x** and **y**.

This section has defined three different binary predicates for comparing data structures. They are summarized in Figure 3, below. The arrows indicate the strength of these predicates.

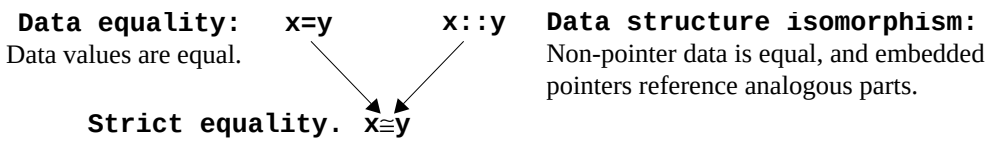


Figure 3: Different relationships between data structures **x** and **y**.

Note that data structure isomorphism is *not* logically stronger than data equality. If a pair of data structure instances have embedded pointers, data equality requires analogous pointers to be equal, while isomorphism requires these pointers to reference analogous parts of their respective data structure instances. Strict equality is, of course, stronger than both.

2.7 Functions that Modify Composite Values

The functions described in Section 2.3 for data structures decompose the composite sorts, that is, they return pieces of a composite value. The set functions described there are the usual ones for combining sets. Functions that

www.manaraa.com

generate data structure and set instances by modifying parts of existing data structure and set instances are very important to programming with the logic. These functions are described below.

Let **A**, **i**, and **x** be terms whose sorts are **Sexp[]**, **integer**, and **Sexp**, where **Sexp** is an arbitrary data structure sort. The array valued function below replaces the data of an array element:

Function	*The result is ...*
A{[i] = x}	**A**, with the data of its i-th element replaced by **x**.

The structure-valued function defined below replaces a field. Let **R** be a structure term with a field **f**, whose sort is **FSort**, and let **x** be a term of sort **FSort**. The notation for replacing a field is shown below.

Function	*The result is ...*
R{f = x}	**R**, with the field **f** replaced by **x**.

Like all functions, those above can be chained. Thus, **A{[0]=1}{[1]=2}** is an array that has 1 and 2 as its first two elements, and with all other elements identical to those of **A**. Again, notation can be abused to make it easier to write these chained functions, and to move the syntax closer to C. First, adjacent right and left curly brackets, **}{**, can be replaced with a semi-colon. The term above is then written **A{[0]=1; [1]=2}**. Second, the left curly brace can be move left past indices, set elements, and field names, with dots inserted where needed to delimit field names. Thus, **A[0]{color='r'}** can be written **A{[0].color='r'}**. These short-cuts are most useful when combined. Consider, for example, the term below, which replaces an element in an array in a structure, and then sets an index to that array element, in the same structure.

```
A {
   list [j] = x;
   mostRecent = j;
}
```

The next axiom defines the array replace function. The variables **i** and **j** are **integer**, and the other free variables belong to the concerned array sort. The results of the function are not defined for array indices that lie outside the array's

www.manaraa.com

range. The function **a{[j]=x}** creates a value that is identical to **a**, except that the data value of the j-th element equals **x**.

For each array sort:

(51) b≅a{[j]=x} ⇔ (
j<len(a) ⇒ (
len(a)=len(b)
∧ (∀i)((0<i len(a)∧ i j)⇒ a[i]=b[j])
∧ &a[j]=&b[j] ∧ b[j]=x
)
)

The two axioms below define the function that replaces a field in a structure. The variables **a** and **b** belong to a structure sort, **f** is a field in the sort, $\mathbf{g_0}$,...$\mathbf{g_n}$ are the other fields, and **x** is a variable that has the same sort as **f**. As above, **b** is defined as strictly equal to **a**, except for the field **f**, which becomes data equal to **x**. (If **f** is a data structure rather than a set, then it retains its address value. The first axiom handles the case where **f** is a data structure, the second, where **f** is a set.)

For each structure sort with fields **f** and $\mathbf{g_0}$,...$\mathbf{g_n}$, where **f** is not a set.

(52) b≅a{f=x} ⇔ (
$a.g_0=b.g_0 \wedge \ldots \wedge a.g_n=b.g_n$
∧ &a.f=&b.f ∧ b.f=x
)

For each structure sort with fields **f** and $\mathbf{g_0}$,...$\mathbf{g_n}$, where **f** is a set.

(53) b≅a{f=x} ⇔ (
$a.g_0=b.g_0 \wedge \ldots \wedge a.g_n=b.g_n$
∧ b.f=x
)

There are some more notational shortcuts that need to be discussed. First, whenever a component appears on the right-hand side of an equal sign in the above functions, it refers to the value of the term to which the function applies. Consider the term below.

A { list[k] = list[i] }
{ list[l] = list[j] }

www.manaraa.com

The term **list[i]** is short for **A.list[i]**. The term **list[j]** is short for **A{list[k]=list[i]}.list[j]**. Thus, if **j=k**, the result of the two function applications will be a term where **list[l]=list[k]**. Another way of viewing this is that the modifications are applied in the order listed. Note that the convention about replacing **}{** with **;** makes the term above equivalent to the term below.

```
A  {  list[k] = list[i];
      list[l] = list[j] }
```

Modifying Sets

With sets, it is convenient to use notation like that for the array and structure functions to denote replacing an element. Thus, **S{e=x}** is taken to be **(S-{e})∪{x}**. This is extended to the fields or array elements of **e**, so that **S{e.f=x}** is taken to be **(S-{e})∪{e{f=x}}**. In the term **{e{f=x}}**, notice that the outer brackets indicate set construction, and the inner brackets indicate the application of a replace function.

As with structures, several elements of a set are often modified in the same term. Because of this, it is useful to define notation for adding and removing elements from sets.

Function	*Meaning*
A{«x}	**A∪{x}**
A{»x}	**A–{x}**

The conventions regarding the use of curly braces that apply to structure and array updates carry over to the above notation for sets. This results in a more terse and intuitive notation. Consider the term below.

```
A {
   « x;
   y.f=g;
   » z;
   }
```

This can be read: “the set **A**, with the element **x** added, the field **f** of the element **y** replaced by **g**, and the element **z** removed.”

www.manaraa.com

The use of curly braces for both set construction — e.g., **A∪{x}** — and to indicate function application — e.g., **A{«x}** — is an unfortunate compromise in the notation. It is ameliorated by the fact that set construction is almost never used. The replacement of **}{** by **;** is only allowed when the curly braces are used for function application. When the curly braces are not needed to associate several function applications, they can be omitted for the set function s above. For example, **A{«x}** can be abbreviated to **A«x**.

2.8 Characterizing Memory States

The logic is intended to characterize and support reasoning about data structures. There are two contexts in which programmers typically reason about data structures. First, they can reason about data structures generally, without regard to any particular memory state. Second, they can reason about data structures with reference to a particular memory state. The logic described so far reflects this first context. The intended model contains as its "domain of discourse" all data structure instances that reside in *any* memory state at *any* time. This model contains too many values to characterize a fixed memory state. For example, for an address a, the two **int** values, $<a, 3>$ and $<a, 4>$ are both in the intended model. Given a fixed memory state, the value at the address a could be 3 or 4, but not both. When reasoning about a fixed memory state, we would like to be able to conclude from the fact that two **int** values have the same address, that therefore they have the same value.

An axiom is given below that, when added to the logic, characterizes a fixed memory state. This axiom is called the *specific state axiom*. It is not considered part of the logic, except when expressly noted.

Specific State Axiom: For **x** and **y** with data structure sorts:

(54) `&x=&y ⇔ InitialPartOf (x, y)`

Note that the above axiom constrains models of the logic. As noted above, the intended model of the logic does not satisfy the specific state axiom, because it includes too many data structure instances.

www.manaraa.com

Given a model of a sorted logic, a submodel is an interpretation of the logic (1) that has for each sort a domain that is a subset of the sort's domain in the larger model, (2) that preserves the interpretation of constant, function, and predicate symbols, and (3) that makes true all theorems. For the logic presented here, (1) and (2) imply (3), because none of the axioms present existential requirements on the domains.

Memory states are identified with *specific state submodels* of the intended model. A *specific state submodel* is any finite set of mutually disjoint data structure instances augmented with (1) all the components of these instances, to make the domains closed under the decomposition functions, and (2) all valid finite compositions of these instances. The original finite set of data structure instances is called the basis. Specific state submodels satisfy the specific state axiom because the initial set of data structure instances are mutually disjoint, so any common addresses must come from data structure composition.

Consider a memory state as (1) any finite set $\mathcal{A}$ of addresses, called the address space, and (2) a *content function* v that maps this set into the discriminated union of the base sorts. Any such memory state generates a unique specific state submodel that has as its basis all values $<a, v(a)>$. Conversely, any specific state submodel can be mapped into a unique memory state. A specific state submodel assigns through its primitive data structure instances a unique data value χ to each address value a. The set of addresses in the primitive data structure instances are the address space, and this assignment is the mapping v. Figure 4 shows this mapping for a specific state submodel that has one data structure instance as its basis.

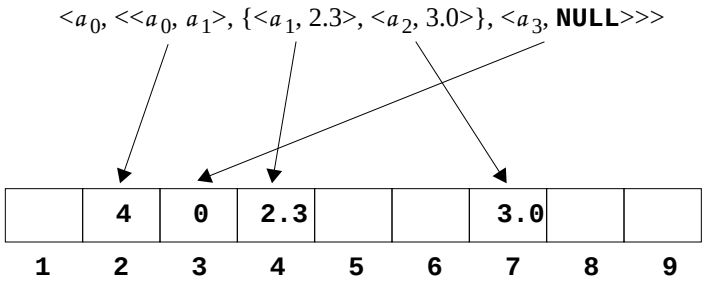


Figure 4: A specific state submodel whose basis has one data structure instance.

www.manaraa.com

The above discussion shows that there is an equivalence between (1) memory states as mappings from finite address spaces to values in the base sorts, (2) finite sets of mutually disjoint data structure instances, and (3) a particular class of submodels of the intended model of the logic. This equivalence is used in discussing the procedural semantics of programs, in Chapter 5.

www.manaraa.com

3. Programming with the Logic

A logic alone does not provide the programmer enough expressive power to describe a computation. Even a logic programming language such as Prolog, in which a program is a set of clauses in Horn clause logic, adds more support for programming than might appear at first glance. Prolog relies on a particular search algorithm that attempts to satisfy goals according to their textual order (something that is logically irrelevant), and provides the programmer with a variety of non-logical constructs for modifying the program's execution, such as `!`, `assert()`, and `retract()`. These constructs are syntactically treated as predicate symbols, but semantically they act to modify a Prolog program's execution rather than to express a logical assertion.

Kowalski [33], discussing the move from procedural languages to logical languages, understood that *something* must be added to logic in order to create programs when he wrote:

Algorithm = Logic + Control

One goal in the design of Galois was to permit the programmer to add as little *something* as possible to create data structure programs. In thinking about this, the Hoare calculus[25] became an inspiration. In it, the behavior of a program *P* is expressed using a precondition and postcondition expressed in logic:

{Precondition} *P* {Postcondition}

This can be taken as a form of program specification. A programming environment that generates code from this specification would generate the program *P* given the precondition and postcondition. The logic of the previous chapter provides the expressive power necessary to write preconditions and postconditions for data structure programs. Galois adds to the logic the ability to specify a program in a form very similar to the Hoare calculus. Thus:

Galois = Logic (Chapter 2) + Program Specification (Chapter 3)

www.manaraa.com

The compilation process turns a program specification into an executable procedure (C function). In Galois, the postcondition specified for a program is always a predicate defined in the logic. The desired program is said to *realize* its predicate and to *satisfy* its specification.

The design of Galois reflects the philosophical principle that, in turning logic to the purpose of programming, the syntax and semantics of the underlying logic should not be sullied. Predicates are defined in a purely logical framework, and are then used to specify programs. Program specification does not affect the logical meaning of the predicates. In this regard, Galois provides a more pure form of logic programming than Prolog. As noted above, Prolog relies on programming constructs that are made to syntactically appear as part of the underlying logic, even though they have no meaning in the logic, and these affect the meaning of the clauses in which they appear in subtle ways.

Section 3.1, immediately following, takes a closer look at program specification. The subsequent two sections discuss how Galois programs are used in applications and how they are prepared. Section 3.4 gives the syntax and explains elements that were not previously described. Section 3.5 gives some simple examples and provides further discussion. Section 3.6 discusses pointers and memory management. It describes how programming in Galois can eliminate the common programming errors related to these. Section 3.7 states some restrictions on writing programs in the logic.

3.1 Introduction to Program Specification

As adumbrated above, a Galois specification for a program P resembles an assertion in the Hoare calculus: {precondition} P {postcondition}. The Hoare calculus does not express termination properties, nor some other execution properties that are described below. In Galois, *execution properties* are appended to the end of a program specification. The example below shows a Galois specification for a program that searches a sorted list. (The preconditions and postconditions for this program are defined in Section 3.5, below.)

www.manaraa.com

```
{AscPts (List)}
doAscPtsSearch (in List, in Key, in Prev, out Here)
{AscPtsSearch} terminates.
```

This example shows the five pieces of information in a program specification. These are:

(1) *Program name:* **`doAscPtrSearch`**. This is used to identify the program. Often, a program that realizes the predicate **`Foo`** is named **`doFoo`**.

(2) *Argument modes:* **`in`** & **`out`**. The first three arguments to the program are input arguments, while the last is an output argument. In the Hoare calculus, the text of a program determines variable use. Because Galois generates a program from the specification, the specification must describe the use of the program's arguments.

(3) *Precondition*: **`AscPts(InList)`**. As in the Hoare calculus, the precondition is a logical formula that must hold on entry to the program. The free variables in the precondition must be a subset of the input arguments to the program.

(4) *Postcondition:* **`AscPtsSearch`**. The postcondition is the name of a predicate that has the same arguments as the desired program. The generated program, when executed, attempts to make the postcondition true. If it does so, it *succeeds*, and returns status indicating success. (Program execution is discussed in more detail below.)

(5) E*xecution properties:* **`terminates`**. This requires the specified program to terminate even if, for a particular execution, there are no values for its output arguments that satisfy the postcondition.

An argument's *mode* describes how the argument is used when the program is executed. There are three argument modes:

in	A *pure input* argument passes a value to the program when the program is executed. Its value is not changed by execution.
inx	A *consumed input argument* passes a value to the program, and the value's memory is consumed (deallocated) by the program's execution. A consumed input argument must have an addressed sort.
out	An *output* argument returns a value from the program.

The pure input arguments and the consumed input arguments are collectively called the *input arguments*, and this set is written I. A set of values bound to the input arguments is called an *input tuple*. The pure input arguments

www.manaraa.com

and the output arguments are called the *result arguments*, and this set is written Q. A set of values bound to the result arguments is called a *result tuple*. The set of consumed input arguments and the set of output arguments are written $\mathcal{D}$ and O, and values bound to these sets of arguments are called a *consumed input tuple* and *output tuple*. Notice that I and Q provide complete information about the arguments' modes, since the pure input arguments are $I-Q$, the consumed input arguments are $I \cap Q$, and the output arguments are $Q-I$.

Input values must be bound at the start of the program's execution, and result values are bound on its return. A result tuple is *valid* for a particular execution if it equals the input tuple on their common arguments and together with the input tuple it satisfies the program's predicate, i.e., the postcondition. A program performs a destructive update by creating output values from consumed input values.

The programmer can specify several kinds of execution properties for a program. A program is:

sound	if it generates only valid output tuples.
effective	if it generates *one* output tuple, whenever a valid output tuple exists.
complete	if it generates *all* valid output tuples (up to data structure isomorphism).
terminates	if it never enters an endless loop.

It is assumed that the programmer always desires a sound implementation. In addition, the programmer can specify that the program is effective or complete, but not both. These execution properties are defined formally in Chapter 5.

3.2 Using Galois Programs

Programs created through Galois are linked into and invoked by an application written in a traditional programming language. Chapter 7 shows code generation targeting C. Each Galois program becomes a C function. Though this

www.manaraa.com

work uses C as the example target language, there is nothing that prevents code generation in Pascal, Ada, or other similar languages.

When used in an application, Galois programs are guaranteed to return only output values that satisfy their postcondition, providing the precondition holds when the Galois program is executed and providing the application abides by conventions for data representation and manipulation. (This is discussed some below and in more detail in Chapter 5.) Thus, Galois provides a way to build verified component libraries for manipulating data structures that are used in applications written in traditional languages.

A Galois program realizes a predicate. A new predicate is often defined in terms of existing predicates. When such a predicate is used as a postcondition in a program specification, the compiler builds a program that invokes programs that were previously compiled for the component predicates. (See Figure 1, below.) The programmer does not have to worry about whether these program's preconditions hold or whether the calling program abides by the conventions for data representation and manipulation. When the compiler constructs a Galois program that calls other Galois programs, it guarantees these things. In this fashion, Galois can be used to build verified component libraries that deal with complex and intricate data structures, and the application programmer need only worry about correctly using the programs that form the interface.

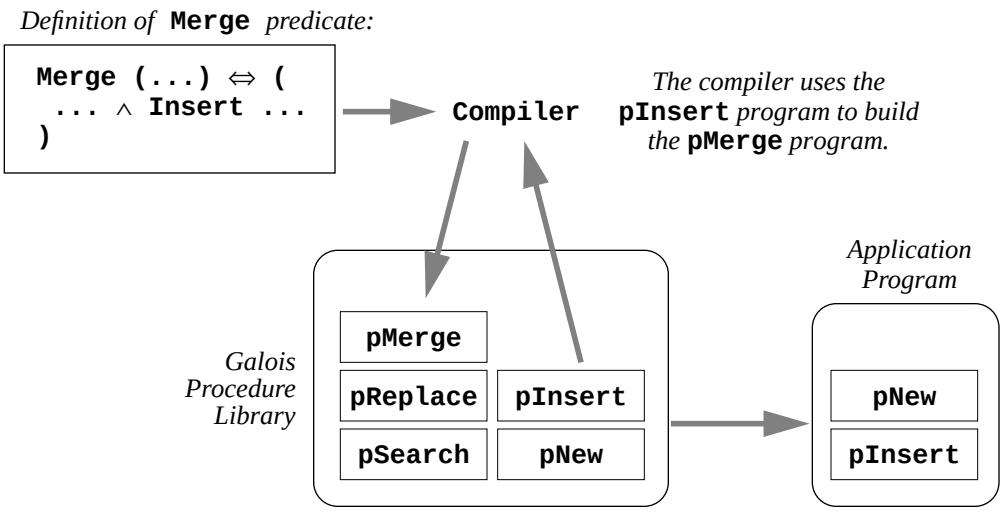


Figure 1: The use of Galois programs.

www.manaraa.com

For Galois programs targeted to C, there is a close correspondence between the logical sorts and C types. Values in the sorts **addr**, **char**, **int**, and **float** are represented by C values with type **(void *)**[1], **char**, **int**[2], and **float**[3]. Arrays are represented as C arrays. A value in a structured sort is represented by the corresponding C structure, with set-valued fields omitted. Each element of a set is represented by a C structure.

When a Galois program executes, it deallocates the memory occupied by its consumed input arguments and allocates memory for its output arguments. This memory management takes place in the C heap using standard C library functions. Thus, any value the application passes as a consumed input to a Galois program must have been allocated on the heap. (The easy way for the application programmer to guarantee this is to pass for consumption only values that have been produced by other Galois programs.)

Data representation is not an issue for Prolog, LDL, or most other logic languages because they are self-contained and each implementation can choose its own data representation. Galois is meant to work with data structures that are understood by C programs (and C programmers!).

3.3 Program Preparation

Ideally, the Galois compiler would be able to accept any predicate definition and any program specification and produce a program that satisfies the specification. The compilation process is not so simple, for several reasons. Most

---

[1] This work uses C as described in [29]. It includes the **void*** type.

[2] The C **int** type has an obvious failing as a representation of the logical **int**: the former has a finite domain while the latter conceptually holds any integer value. This problem is ignored in this work. A clean implementation requires use of an **int** type that is arbitrary precision, or that at least raises an exception when the implemented domain is exceeded.

[3] The **float** sort in the logic is assumed to have axioms that reflect the behavior of the C **float** type. (This is not practical for the **int** sort, because true integers are needed in the logic for induction proofs.)

www.manaraa.com

glaringly, some specifications are impossible to satisfy. Consider, for example, a predicate that defines whether a universal Turing machine halts and a specification that demands a program that both realizes this predicate and terminates on all inputs. Or consider a specification for a program that must find an unreachable piece of data. It is to Galois's credit that the programmer can logically define the concerned predicates, but will discover during the compilation process that the desired programs cannot be created. (This is far better than what happens with a traditional language, which permits the programmer to build a program that executes, though incorrectly, leading the programmer to work at fixing the program.)

Within the realm of what is computable, more practical concerns arise. The programmer can define a predicate that can be realized by a program, but that is too abstractly defined to permit immediate compilation. In such a case, the programmer will have to define a logically equivalent predicate that presents the compiler an easier task. (This is discussed more in Chapter 5.)

As discussed above, the compiler will guarantee that any Galois program composed from other Galois programs will guarantee the preconditions of these. This requires the compiler to analyze the preconditions of included programs relative to the context in which they are used and the precondition of the program being compiled. The procedural calculus in Chapter 5 shows how this analysis is done, yielding a *prerequisite assertion* which, if true, validates the compiled program. The compiler relies on the programmer using theorem proving tools to verify the prerequisite assertion. (See Figure 1.) Theorem proving is beyond the scope of this work; we assume that this task is practical. (As importantly, Galois seems to increase program reliability even when the programmer verifies the prerequisite assertion by inspection. The prerequisite assertion makes explicit the assumptions that are made by program execution, which assumptions are often overlooked in more traditional programming. Patrick Ray [58] investigates the use of Galois in the absence of a theorem prover.)

www.manaraa.com

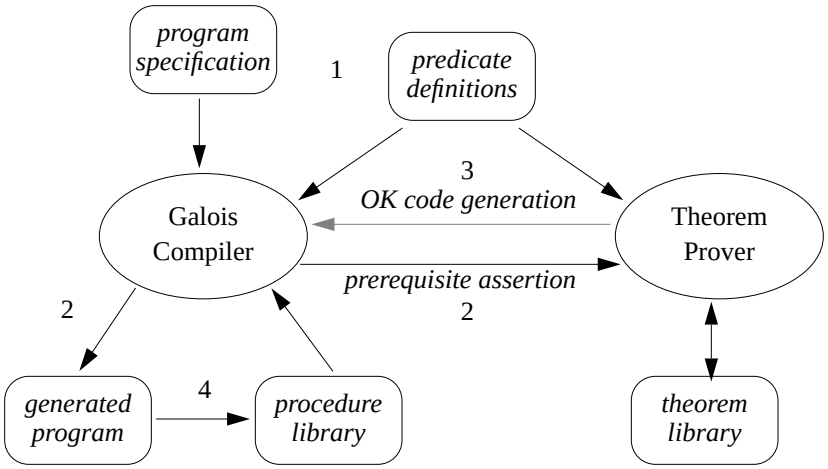


Figure 2: Interaction between compiler & theorem prover.

Figure 1 shows the data and control flows between the compiler, the theorem prover, and the various data elements. The process of creating a Galois program is outlined below. Each step explains part of Figure 1.

(1) The programmer defines a predicate and specifies a desired program. The predicate definition adds to the set of predicate definitions, and the program specification is the primary input to the compiler.

(2) The compiler generates a composition for the program and produces the prerequisite assertion that validates it. If the compiler fails, it displays the failure points to the programmer so that the programmer can modify the program specification or predicate definition. (This step is detailed in Chapter 5.)

(3) The prerequisite assertion is verified. The theorem prover may use and update a theorem library concerning predicates during this step. If the prerequisite assertion cannot be verified, then the programmer must revise the program specification (perhaps strengthening the precondition) or the predicate definition (adding run-time checks).

(4) Once the prerequisite assertion is verified, the compiler generates code in the target language and adds the procedure to the procedure library.

This process is a more complex route to executable code from logic than exemplified in Prolog, where any syntactically correct program executes. The greater preparatory effort required by Galois buys three improvements over Prolog. *Correctness:* The programmer can select, and the compiler verify, very strong execution properties. For example, an effective or complete Galois program

www.manaraa.com

never fails to give a tuple that satisfies its predicate, when one exists. In contrast, Prolog programs without control constructs are only sound, and with control constructs they can give wrong answers. *Expressiveness:* Galois programs can express and manipulate pointer-based data structures. *Efficiency:* Galois programs execute efficiently and are capable of destructively updating pointer-based data structures.

3.4 The Elements of Galois

At the top level, there are three elements of the programming language: predicate definitions, sort definitions, and program specifications. Predicate definitions are made in the logic, as described in Chapter 2. Sort definitions merely provide a way to attach a name to sort expressions for convenient use. Program specifications play the role discussed in the previous sections.

These are the only elements of the language. Their description below includes a BNF grammar. Five non-terminals of this grammar reference syntactic parts of the logic in Chapter 2: **<sortExpr>** is a sort expression, **<identifier>** is a sort, predicate, or variable name, **<arg>** is an argument declaration, **<formula>** is a formula, and **<pliteral>** is a positive literal. These non-terminals are not defined in the BNF grammar below. (All other non-terminals begin with a capital letter.)

The Predicate Definition

A predicate definition introduces a new predicate name and a recursive definition for the predicate. It is a formula in the logic with the following form:

```
<PredDef> := <PredName> (<PredArgList>) ⇔ (<Body>)
<PredArgList> :=   | <arg> | <arg> , <PredArgList>
<PredName> := <identif ier>
<Body> := <formula>
```

That is, a predicate definition is a formula of the form

www.manaraa.com

p(a$_0$,...a$_n$) $\Leftrightarrow$ Φ, where each argument **a$_i$** is a variable with a sort declaration as described in Chapter 2. The *body* of a predicate definition is a formula in the logic, Φ, whose free variables are the predicate's arguments. The body may include uses of the predicate being defined, **p**. To ease parsing, the body must be enclosed in parentheses. Because a predicate definition is just a formula in the logic, as described in Chapter 2, its syntax and meaning are not discussed further here.

The Sort Definition

A sort definition is very much like a C **typedef**. It associates a name with one of the logic's sorts, so that the name can be used as a sort declaration for an argument or variable. The format of a sort definition is:

```
<SortDef> := sortdef <sortExpr> <SortName>;
<SortName> := <identif ier>
```

Sort expressions are described in Chapter 2. Two examples of sort definitions follow.

```
sortdef struct {Next: addr; X: f  loat; Y: f loat;} Point;
sortdef struct {
   addr Head;
   Point Pts{};
} PointList;
```

The first sort definition above describes a structured sort, named **Point**, that has floating point fields **X** and **Y**. The second sort is also a structured sort, named **PointList**, that contains a set of **Point** elements and an address (intended to reference the first point.) As in Chapter 2, a small abuse of notation allows a syntax that is more like C: the square and curly brackets that specify arrays and sets can follow a field name or sort name.

The Program Specification

The programmer specifies a program in a *program specification*. In addition to the predicate that the program realizes, the programmer must provide a

www.manaraa.com

name, argument modes, precondition, and execution properties. The syntax for a program definition is given below.

```
<ProgSpec> :=
   {<Pre>} <ProgName> (<MAList>) {<PredName>} <PropList>.
<ProgName> := <identif ier>
<PredName> := <identif ier>
<Pre> :=  { <formula> }
<MAList> :=   | <ModeAssign> | <ModeAssign> , <MAList>
<ModeAssign> := <Mode> <ArgName>
<Mode> := in | inx | out
<ArgName> := <identif ier>
<PropList> :=   | <Property> | <Property> , <PropList>
<Property> := sound | effective | complete | terminates
```

Before further explanation, an example might be helpful. The program declaration below says that the program **foo** generates all values of **y** that satisfy the predicate **p** for a given value of **x**, providing that **x** satisfies the precondition **q** when **foo** is invoked.

```
{q(x)} foo (in x, out y) {p} complete.
```

There are several syntactic rules that apply to program definitions, all of which are intuitive. The arguments given the program must correspond in name and place to those for the predicate it realizes. Only arguments that have a structure or array sort can have the mode **inx**. The free variables of the precondition must be a subset of the program's input arguments.

The same predicate can be used to define several programs. One program might calculate one set of output arguments, and a second might calculate a different set of output arguments, while both realize the same logical predicate. Some predicates are not realized by any program. They serve as preconditions, or to define other predicates.

The Module and Miscellaneous Syntax

A module is a coherent collection of predicate definitions, sort

www.manaraa.com

definitions, and program definitions. The scope of a sort definition is its module; that is, the defined sort name can be used in any predicate definition in the module, but does not apply outside the module. Program and predicate definitions can use predicates defined in the module. (Nonetheless, a recursive cycle can only be a single loop. Predicate **p** is said to be *defined in terms of* **r** if the body of **p** refers to **r**, or refers to a predicate **q** that is defined in terms of **r**. The predicate **p** can be defined in terms of itself, but **p** cannot be defined in terms of any *other* predicate that is defined in terms of **p**.)

While not shown in the BNF grammar, Galois permits comments in the style of C++. It also allows integer constants for the module to be parameterized in the style of C++. For example, the line below defines **N** to be **20**.

```
int const N=20;
```

Modules are entirely independent of one another. Thus, the names in one module never collide with the names in another module, nor can definitions in one module refer to predicates or sorts defined in another module. Each module has its own procedure library and theorem library. This organization and name scoping is not conducive to programming in the large, but it suffices for the purposes of this work.

3.5 Examples and Discussion

Typically, some predicates in a module serve only to define a data structure type. For example, the predicate below defines an ordered point list. This data structure type is a linked list of points ordered on their x-value. (See part A of Figure 3.) It uses the **PointList** sort defined in the previous section.

```
AscPts (Ex: PointList) ⇔ (
   Ex.Head=NULL ⇒ Ex.Pts=EMPTY
∇  (∃First: Point)(
      First∈Ex.Pts ∧ &First=Ex.Head
   ∧  First.Next NULL⇒ (∃Scnd: Point)(
         Scnd∈Ex.Pts ∧ &Scnd=First.Next ∧ Scnd.X>First.X
```

www.manaraa.com

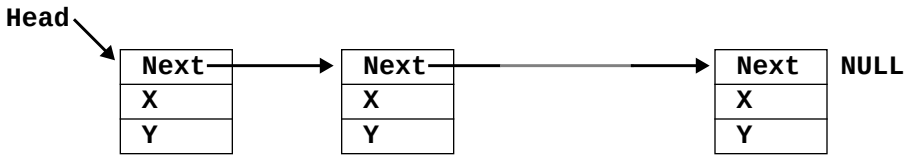
```
        )
    ∧   AscPts ( Ex {    Head = First.Next;
                         Pts = Pts{»First} }
        )
)
```

The **PointList** sort is a structure that contains a pointer, **Head**, together with a set, **Pts**. The elements of **Pts** form a singly linked list, with elements increasing in **X** value down the list.

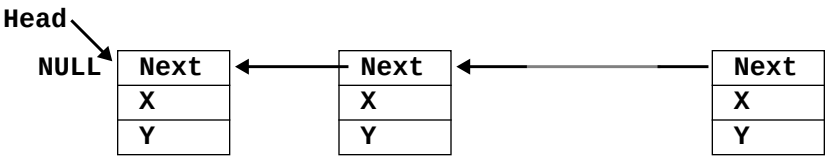
All the elements of a **PointList** data structure can be reached by chasing the **Next** pointers. (See part A of Figure 3.) The following assertion, which is provable in the logic from the above definition, expresses this fact.

AscPts (x) ⇒ ((z∈x.Pts) ⇒ RPartOf (z, x))

One can *logically* define data structures where it is impossible to reach all pieces of the data structure. Consider, for example, a linked list with pointers running backwards. (See part B of Figure 3.) Practically, this is a terrible blunder. Such a blunder is revealed when the programmer tries to compile a program involving this data structure. The compiler will be unable to generate code that does anything useful. If a program is specified that requires accessing unreachable elements, the compiler will either issue errors, or it will ask the theorem prover or programmer to verify assertions that are false.



A: The AscPts data structure. *ascending* **X** *values* →



B: A backward linked list, a very stupid data structure.

Figure 3: Examples of data structures.

www.manaraa.com

An alternate definition of the **AscPts** data structure type is shown below. It is more terse than the previous definition. It asserts the following invariant: (1) if the point set is not empty, there is one point that has a null **Next** pointer, (2) all elements of the point set are reachable, and (3) the **Next** pointer, if not **NULL**, always points to something with a greater **X** value. This definition might be preferred for proving assertions about the data structure type or about programs that use it. It is provably equivalent to the former definition.

```
AscPts (Ex: PointList) ⇔ ( // Alternate def inition
   Ex.Head NULL⇒ (∃pt∈Ex.Pts)(pt.Next=NULL)
∧  (∀pt∈Ex.Pts)(
      RPartOf(pt, Ex)
   ∧  pt.Next NULL⇒
         (∃pt2∈Ex.Pts)(pt.Next=&pt2 ∧ pt.X<pt2.X)
   )
)
```

The formula above makes use of a frequent abbreviation. Given a term **t** whose sort is **Sexp{}**, the formulae **(∀x:Sexp)(x∈t ...)** and **(∃x:Sexp)(x∈t ...)** are ubiquitous. These are abbreviated to **(∀x∈t)** and **(∃x∈t)**.

The next predicate defines a search of an ordered point list. It is the body of a program that given a list (**InL**), a key value (**Key**), and a default address (**Prev**), returns the address (**Here**) of the point furthest down the list whose **X** value is less than or equal to the key value, or the default address if all points in the list have **X** values greater than the key value. (The program definition is given later.)

```
AscPtsSearch
(InL: PointList, Key: f loat, Prev: addr, Here: addr)⇔ (
   InL.Head=NULL ⇒ Here=Prev
∇  (∃pt∈InL.Pts)(
      &pt=InL.Head
   ∧  (  Key<pt.X ⇒ Here=Prev
      ∇  AscPtsSearch ( InL {Head=pt.Next, Pts{»pt}},
            Key, Here, &pt)
      )
```

www.manaraa.com

```
    )
)
```

The next predicate defines a relationship between (1) an ordered point list, (2) a new point whose x-value is not in the first list, and (3) an ordered point list that includes both the points from the first list and the new point. In short, it defines the input-output relation for the operation that inserts into an ordered list. It uses the search predicate.

```
AscPtsInsert
(InL: PointList, Pt: Point, OutL: PointList)⇔ (
   (∃New: Point, Here:addr)(
      AscPtsSearch (InL, Pt.X, NULL, Here)
   ∧  Here=NULL
      ⇒ (  New::Pt{Next=InL.Head} // Insert at head
         ∧  OutL::InL {Head=&New, Pts≅Pts{«New}}
         )
      ∇  (∃HPt∈InL)( // Insert after HPt
            Here = &HPt
         ∧  HPt.X   Pt.X // Fail if keys are equal
         ∧  New::Pt{Next=HPt.Next} // Copy ptr to New
         ∧  OutL::InL{Pts≅Pts{«New; HPt.Next=&New}}
      )
   )
)
```

The predicate above has a structure almost identical to a list insert operation written in C or Pascal. It can be given the following interpretation. “To insert into an ordered point list, first search for the greatest existing point whose key value is less than or equal to the new point. If no such point exists, insert the new point at the head of the list. If the found point equals the new point in key value, then the insert fails. Otherwise, insert the new point after the found point.” Even though the program deals with pointer-based data structures and uses pointer manipulations familiar to a C programmer, and even though the compiler will generate the expected C code, the predicate definition is a formula in a first-order logic. Assertions logically derived from the predicate definition will accurately qualify the output tuples produced by the procedure.

www.manaraa.com

The following program specifications qualify the desired list search and insert programs. In both cases, the precondition is merely that the input data structure is an ordered list of points.

```
{AscPts (InL)}
doAscPtsSearch (in InL, in Key, in Prev, out Here)
{AscPtsSearch} effective, terminates.
{AscPts (InL)}
doAscPtsInsert (inx InL, in Pt, out OutL)
{AscPtsInsert} effective, terminates.
```

For the insert program, the mode for the input list specifies that it is consumed, which is to say, it is destructively updated in producing the output list. In the predicate definition for **`AscPtsInsert`**, note the use of data structure isomorphism to define the output data structure (**`OutL`**) as equivalent to the input data structure (**`InL`**) with some modifications applied. This technique will be used throughout the examples. Chapter 6 describes how this is realized as a destructive update.

The same predicate could be used in a program definition that preserves the input list, copying it to produce the output list. The two programs would realize the same logical predicates, but their program specifications would impose different argument modes.

When the compiler is generating code for a program, and the defining predicate references other predicates, the compiler searches the procedure library for previously compiled programs that realize these other predicates. The compiler searches for a program whose argument modes and execution properties are useful in generating code for the program on which it is currently working. In particular, the compiler insures that consumed values are not used after they are consumed. Thus, the compiler guarantees that data structure allocation, use, and destruction are properly sequenced. (At the interface between the application and Galois programs, it remains the programmer's responsibility to insure that the application abides by conventions regarding data manipulation. These are described in detail in Section 5.4.)

www.manaraa.com

3.6 Pointers, Sets, and Memory Management

Because a set value in the logic is represented in C by an arbitrary collection of disjoint C data structures, it is impossible to find an element of a set unless there is a pointer to it. Pointers into sets are chased by formulae such as the one below.

```
(∃x∈Sexp)(&x=pt ∧ ...)
```

The variable **pt** is the pointer that is dereferenced, and the elided portion of the formula makes use of the referenced element.

The astute reader may ask: since there is no explicit structuring of elements in a set, except for the internal references that are defined by the programmer, what does it mean — operationally — to add elements to or remove elements from sets? The answer is that these things describe when memory is allocated and released.

Normally, when the compiler generates code for an existential formula, it generates code that (1) allocates space for the existentially quantified variable and gives it a value, and that then (2) deallocates the memory occupied by the variable at the end of the existential scope. This second step is omitted if the existentially quantified variable is added to a set that is part of an output argument. Consider the **AscPtsInsert** predicate from Section 3.5. The variable **New** is allocated, assigned a value, and then deallocated unless it is added to the set **InL.Pts** and returned in the guise of **OutL**. Conversely, a predicate that deleted an element from the ordered linked list would eject an element from a set, and the generated code would appropriately deallocate the memory that had been allocated for it.

In writing programs, the programmer does not have to think about memory allocation and deallocation. Instead, the task is to define data structure invariants that have no lost parts or dangling pointers, and to define predicates for input-output relationships that preserve these invariants. A memory leak occurs

www.manaraa.com

when a program creates a data structure with lost parts. An erroneous memory deallocation occurs when a program creates a data structure with a dangling pointer. In the example for the ordered point list, the programmer would want to prove, in the logic, the following two assertions.

- **AscPts(x)** $\Rightarrow$ **Navigable (x)** $\wedge$ **~HasDangles (x)**
- **(AscPts(InL)** $\wedge$ **AscPtsInsert(InL, Pt, OutL))**
  $\Rightarrow$ **AscPts(OutL)**

If these assertions are true, then the compiler will generate code that allocates and deallocates memory correctly. The same is true for arbitrarily complex data structures. If the programmer proves (1) that the data structure is navigable and has no dangling pointers and (2) that all programs concerned generate outputs that satisfy the data structure's invariant, then memory leaks and dangling pointers are eliminated.

Regardless of whether the programmer does (1) and (2), the compiler will not generate code that chases dangling pointers. Wherever a pointer is chased, the compiler generates an assertion that asks the programmer to prove from the preconditions of the program and "what the program has made true so far" that the pointer dereference is valid. This assertion is made part of the prerequisite assertion that must be verified before the compiler produces code for the specified program. (The notion of "what the program has made true so far" is made concrete in Section 6.5.)

Failure to allocate and deallocate memory correctly and chasing invalid pointers are problems that plague C programming. These bugs are subtle, difficult to track down, and often persist in software throughout its commercial life. Galois gives these issues a formal expression, so that formal reasoning about them is supported, and producing programs free of these bugs is reduced to verifying in a first-order logic a set of assertions that are automatically generated.

www.manaraa.com

3.7 Restrictions on Use of the Logic for Programs

As previously mentioned, the compiler will balk at some predicates, even though they are syntactically correct. Chapter 1 briefly described how programs whose predicates are expressed too abstractly must have their predicates written more concretely in order for the compiler to successfully process them. These limitations in using the logic are discussed below. (Chapter 5 and Chapter 7 provide more detailed explanation.) It is important to keep in mind that not every predicate is intended to be realized by a program. For example, a predicate that serves as a data structure invariant is useful in proofs and preconditions, even though it is never realized by a program.

Syntactic Restrictions

To be realized by a program, a predicate that is recursively defined must have recursion only at an even negation depth. This is formally defined in Chapter 4. Roughly, this means that determining whether a tuple satisfies the predicate cannot depend on determining that some other tuple *fails* to satisfy the predicate.

A second restriction concerns universal quantification. A universally quantified formula will compile only if it has the form $(\forall \mathcal{A})(\theta \Rightarrow \phi)$, θ is realized as a complete procedure whose output variables are $\mathcal{A}$, and ϕ has no output variables. In short, universal quantification can be used only to iterate over a set of values that are explicitly generated and to verify a condition on this set.

Context-Sensitive Restrictions

Pointers can be dereferenced only as discussed in the previous section. Sets must be used, as described there, to specify the addition of pieces to or the removal of pieces from a data structure instance.

www.manaraa.com

Primitive data structures (**int**, **float**, **char**, and **addr**) can be manipulated freely in the logic. But more complex data structures can be used only in certain ways. A complex data structure instance is created in one of two ways. First, a data structure instance is created and bound to a variable of the concerned sort that is existentially quantified. This instance is valid within the quantifier's scope. Second, an output variable can be defined as an isomorphic copy of a data structure instance, possibly modified by the data structure functions. The concerned literal has the form **y::x{...}**, which can be read "**y** is an isomorphic copy of **x**, with the following modifications ...". If **x** is destructively consumed, and it is no longer needed in the program, then this literal acts to destructively update **x** and bind the result to **y**. Otherwise, the literal acts to perform a deep copy of **x**, modify the result, and bind the modified instance to **y**.

The compiler cannot generate code that performs a deep copy of arbitrary data structures. In general, if this is needed, the programmer must specify a program that effectively realizes a predicate **r(x, y)**, with **x** as an input variable and **y** as an output variable, for all **x** in the concerned data structure class, such that **r(x, y)** $\Rightarrow$ **x::y** is a theorem of the logic. Such a program is called a *copy constructor*. In essence, it tells the compiler how to copy **x**. A copy constructor is also needed to deallocate a data structure instance, for example, when a data structure instance is consumed by a program that does not recycle the consumed instance in producing an output instance. (Deallocation can be viewed as copying to the null device, which is why a copy constructor enables deallocation.) Memory allocation and deallocation are discussed in more detail in Chapter 7.

A final restriction concerns input argument consumption. Programs that potentially generate multiple output tuples, i.e., complete procedures that are not functions, may not have consumed input arguments. The intuition behind this is that the entire input tuple must survive until the last result tuple is produced.

www.manaraa.com

4. Relational Models of Logic

This chapter presents a declarative semantic for the logic, i.e., an intended model. The first two sections reprise the relational algebra and show how it can provide a semantic interpretation of the predicate calculus. This is old hat, and the purpose is solely to prepare the way for what follows. The third section shows how a logic and its model can be extended in a natural and minimal fashion given recursive predicate definitions that satisfy certain syntactic constraints. The expressive power of recursive predicate definitions are related to Horn clause programs.

4.1 Review of Relational Algebra

This section reviews the relational algebra[1] to acquaint the reader with the notation that will be used here. The relational algebra builds on a collection of *domains*, $_0$, $_1$, $_2$, ..., each of which has a denumerable set of *values*, and a set ***V*** of *variables*. Each variable x is assigned to a domain (x). The variables assigned to a domain $_i$ are denoted $V(_i)$. (Variables are more commonly called attribute names or column labels, but since they will correspond to variable symbols in the logic, the former terminology is more convenient here.)

An *assignment* is an ordered pair whose first member is a variable and whose second member is a value from its domain. An assignment with variable x and value v is written $x \rightarrow v$ or v/x. (This latter notation is read "v replaces x.") Two assignments are *disjoint* if they have different variables. A finite set of pairwise disjoint assignments is called a tuple, and can be written $\{v_0/x_0,...v_n/x_n\}$ or $\{x_0 \rightarrow v_0,...x_n \rightarrow v_n\}$. The values $v_0,...v_n$ need not be distinct.

The *relational signature* $\tau(t)$ of a tuple t is the set of variables to which

[1] The relational algebra is described in many standard database texts, such as [30].

www.manaraa.com

it assigns values.Thus, the relational signature of $\{v_0/x_0,...v_n/x_n\}$ is $\{x_0,...x_n\}$. Two tuples are disjoint if their relational signatures are disjoint as sets. The *maximal relation* M_τ for a signature τ is the set of all tuples that have the signature τ.

A *relation* R is a signature $\tau(R)$ and a set of tuples all of which have the signature $\tau(R)$. In a slight abuse of notation, the same symbol is used for both the relation and its set of tuples. (A relation cannot be defined as a set of tuples with the same signature, since then the empty relation would have no signature. It will be important that each signature has its own empty relation.) In database theory, relations are sometimes required to be finite, but this restriction does not serve the purpose of interpreting logic, and so it is not assumed in this work.

Column projection and subtraction are basic operations that apply to both tuples and relations. One can view them as essentially tuple operations that extend to relations by application to each member tuple (and by application to the relation's signature), or as essentially relational operations that apply to tuples when these are viewed as relations with one member. They are defined below for tuples, and their extension to relations is assumed.

$\pi_{\mathcal{C}}(t)$	**Tuple projection.** Let $\mathcal{C}$ be a relational signature. $\pi_{\mathcal{C}}(t)$ is the subset of t such that $\tau(\pi_{\mathcal{C}}(t))=\mathcal{C}\cap\tau(t)$, i.e., the assignments in t to those of its variables that are in C.
$t\backslash D$	**Column subtraction.** D is a signature. $t\backslash D=\pi_{(\tau(t)-D)}(t)$.

This expositions will use several other operations of the relational algebra. These are defined below.

$s\cdot t$	**Tuple concatenation.** If s and t are disjoint tuples, $s\cdot t$ assigns to x the value a iff $a/x \in s$ or $a/x \in t$. Note that columns are labeled rather than ordered, so concatenation is commutative: $s\cdot t=t\cdot s$.
$\sigma_i(R)$	**Relational selection.** This selects all tuples of R whose values equal i in their common variables: $\sigma_i(R) = \{ t \mid t\in R \wedge \pi_{\tau(i)}(t)=\pi_{\tau(t)}i \}$.
$R \bowtie Q$	**Natural join.** $R \bowtie Q = \{ t \mid (\exists q)(\exists r)(\exists s)(t=q\cdot r\cdot s \wedge r\cdot s\in R \wedge q\cdot s\in Q \wedge \text{disjoint}(\tau(r),\tau(q))) \}$. Note that the natural join is the cross product when R and Q share no variables.
R/Q	**Relational divide.** $R/Q = \{ t \mid (\forall u)(u\in Q \Rightarrow t\cdot u\in R) \}$.

www.manaraa.com

R+Q	**Relational union.** If $\tau(R)=\tau(Q)$, then $R+Q=R\cup Q$.
R–Q	**Relational difference.** $R-Q=\{ t \mid t\in R \wedge (\forall u)(u\in Q \Rightarrow \pi_{\tau(R)}(u)\pi_{\tau(Q)}(t)) \}$. Note that this definition allows R and Q to have different signatures, and it removes from R all tuples that are equal to any tuple in Q modulo their common columns.
~R	**Relational complement.** $M_{\tau(R)}-R$.

The above description of the relational algebra leaves open the content of the domains. A particular relational algebra fixes these objects.

4.2 Relational Interpretation of Logic

The next task is the definition of an interpretation for sorted, first-order logic using the relational algebra. An interpretation of a logic L is a relational algebra ***A*** and a denotation function ***D*** that maps the syntactic elements of the logic into the relational algebra. As is common, the application of the denotation function to a formula or term $\mathbf{x}$ in the logic is written $[\mathbf{x}]$, rather than $\boldsymbol{D}(\mathbf{x})$. An interpretation of a sorted logic assigns to each sort $\mathbf{s}$ in the logic a domain $(\mathbf{s})$, to each constant $\mathbf{c}$ a value $[\mathbf{c}]$ in the domain of its sort, and to each function symbol $\mathbf{f}$ with signature $\mathbf{s_0},...\mathbf{s_n} \rightarrow \mathbf{s_f}$ a function $[\mathbf{f}]$ from $(\mathbf{s_0}) \times ... \times (\mathbf{s_n})$ to $(\mathbf{s_f})$. This exposition departs from the usual trail at this point, and does something a little different for predicates and formulae.

First, variable symbols in the logic are identified with variables in the relational algebra; that is, to serve as an interpretation, a relational algebra must have the same variables as the logic, and each variable's sort (in the logic) must be assigned to the variable's domain (in the algebra). This gives rise to a natural interpretation of terms in the logic. Given a term $\mathbf{f(x_0,...x_n)}$ where $\mathbf{x_0},...\mathbf{x_n}$ are variables, the interpretation of this term is a function $[\mathbf{f(x_0,...x_n)}]$ on tuples whose signatures contain $\rho=\{\mathbf{x_0},...\mathbf{x_n}\}$. For a tuple t with $\pi_\rho(t)=\{\mathbf{x_0}/v_0,...\mathbf{x_n}/v_n\}$, define $[\mathbf{f(x_0,...x_n)}](t)=[\mathbf{f}](v_0,...v_n)$. This definition is recursively applied to a general term $\mathbf{f(e_0,...e_n)}$, where $\mathbf{e_0},...\mathbf{e_n}$ are subterms, by: $[\mathbf{f(e_0,...e_n)}](t) = [\mathbf{f}]([\mathbf{e_0}](t),...[\mathbf{e_n}](t))$. Thus, given any term $\mathbf{e}$ in the logic with variables $V(\mathbf{e})$, its interpretation is a function *from* any tuple that assigns values to the variables $V(\mathbf{e})$

www.manaraa.com

to a value in the domain of **e**'s sort.

Each atomic formula $\mathbf{p(e_0,...e_n)}$ of the logic is assigned to a relation $[\mathbf{p(e_0,...e_n)}]$ whose relational signature comprises the free variables of $\mathbf{p(e_0,...e_n)}$, that is, $\tau([\mathbf{p(e_0,...e_n)}])=V(\mathbf{p(e_0,...e_n)})$. Relations assigned to different atomic formulae with the same predicate symbol must be equivalent modulo the calculation of values for their arguments. This is formally expressed by the constraint below. This requirement uniquely determines the relation assigned to an atomic formula $\mathbf{p(e_0,...e_n)}$ given (1) the relation assigned to $\mathbf{p(e_0,...e_n)}$ where $\mathbf{e_0,...e_n}$ are variables, and (2) the denotations for terms.

Consistency of relations assigned to atomic formulae:

Given that both $\mathbf{p(e_0,...e_n)}$ and $\mathbf{p(x_0,...x_n)}$ are well-formed formulae, then $t \in [\mathbf{p(e_0,...e_n)}] \Leftrightarrow \{([\mathbf{e_0}](t))/\mathbf{x_0},...([\mathbf{e_n}](t))/\mathbf{x_n}\} \in [\mathbf{p(x_0,...x_n)}]$.

An interpretation as defined above assigns semantic structures to variables and formulae. The usual notion instead assigns semantic structures to predicate symbols. (Both notions identically treat constants, functions, and terms.) The table below summarizes what has been done so far.

the logical element ...		*is assigned by **D** to the element of **A** ...*
s	a sort	(**s**), a domain
x	a variable symbol	**x**, a variable (column label)
f	a function symbol	[**f**], function on appropriate domains
e	a term	[**e**], a function on tuples whose relational signature includes the variables of e
p(..)	an atomic formula	[**p(..)**], a relation whose signature is V(**p(..)**)

The final step assigns a relation to every well-formed formula in the logic. The boolean constant False is assigned the empty relation with an empty signature, written **F**. The boolean constant True is assigned the relation with an empty signature that contains one (empty) tuple, written **T**. The important properties of these relations from the algebraic viewpoint is that **F** is a zero and **T** an identity of the relational join; that is, for any relation R, **F**×R={}=**F** and **T**×R=R. The relations assigned to arbitrary formulae are calculated from those assigned to atomic formulae according to the rules below.

www.manaraa.com

Calculate ...	*by ...*	*with signature ...*
[**False**] =	**F**	{}
[**True**] =	**T**	{}
[~φ] =	~[φ]	V(φ)
[φ ∧ ξ] =	[φ] ⋈ [ξ]	V(φ) ∪ V(ξ)
[(∀x) φ] =	[φ] / δ(x)	V(φ) – {x}
[(∃x) φ] =	[φ] \ {x}	V(φ) – {x}

It is trivially verified that the rules for the two quantifiers satisfy the equality [(∃x)(φ)] = [~(∀x)~φ(x)], and that conjunction and negation under quantification behave as expected. The other logical participles (⇒, ⇔, and ∨) are handled by rewriting them in terms of conjunction and negation. A tuple t is said to *satisfy* a formula φ under an interpretation precisely when t is in the relation the interpretation assigns to φ.

4.3 Models and Least Fixed-Point Extensions to the Logic

As usual, a closed formula is *true* in an interpretation if it is assigned the value **T**, and an interpretation is *valid* for the logic or a *model* of the logic if all theorems in the logic are true in the interpretation. This narrative now assumes that there is an interpretation of the logic that serves as its intended model. This section shows how a new predicate can be added to the logic, together with a recursive axiom that defines it, in a fashion that gives rise to a unique and minimal extension of the model. Any number of new predicates can be thus added to the logic, one at a time. This definitional method is common in logic, but has not been widely applied in logic programming.

A formula $\Phi_{\mathbf{p}}$ with free variables $\mathbf{x}_0,...\mathbf{x}_0$ of sorts $\mathbf{s}_0,...\mathbf{s}_0$ and perhaps involving a new predicate symbol **p** whose signature is $\mathbf{s}_0,...\mathbf{s}_0$ can be assigned a relation according to the above rules, *given* a prior assignment of some relation P to **p**. The relation thus assigned to $\Phi_{\mathbf{p}}$ is denoted $[\Phi_{\mathbf{p}}](P)$. The set of relations with relational signature $\{\mathbf{x}_0,...\mathbf{x}_0\}$ are partially ordered by set containment, and they form a complete lattice under this ordering. $[\Phi_{\mathbf{p}}]$ acts as an operator on this lattice.

www.manaraa.com

(For any relation P in this lattice, $[\Phi_{\mathbf{p}}](P)$ is another relation in the lattice.) A lattice operator is continuous precisely when it is monotonic [31]. Thus, if $[\Phi_{\mathbf{p}}]$ is monotonic, it then has a least fixed-point, and that least fixed-point, written $[\Phi_{\mathbf{p}}]\uparrow\omega$, is the least upper bound of all finite applications of $[\Phi_{\mathbf{p}}]$ to the empty relation. When this least fixed-point is taken as the interpretation of **p**, then the model so extended is a model of the logic when the predicate symbol **p** is added along with the defining axiom $\mathbf{p}(\mathbf{x}_0, \ldots \mathbf{x}_n) \Leftrightarrow \Phi_{\mathbf{p}}$. This least fixed-point is algebraically expressed by the below recursion.

$$[\Phi_{\mathbf{p}}]\uparrow 0 = [\Phi_{\mathbf{p}}](\{\})$$

$$[\Phi_{\mathbf{p}}]\uparrow i+1 = [\Phi_{\mathbf{p}}]([\Phi_{\mathbf{p}}]\uparrow i)$$

$$[\Phi_{\mathbf{p}}]\uparrow \omega = \cup_{i<\omega} [\Phi_{\mathbf{p}}]\uparrow i$$

The question remains: when is $[\Phi_{\mathbf{p}}]$ monotonic? We now give a characterization of monotonicity in terms of how **p** is used in $\Phi_{\mathbf{p}}$. A negation operator (~) *applies* to a predicate instance if it directly precedes the predicate instance or it directly precedes a formula that contains the predicate instance. The *negation depth* of an instance of the predicate **p** in $\Phi_{\mathbf{p}}$ is the number of negation operators that apply to it, taking universal quantification, conjunction, and negation as the primitive logical participles.[2] In this algebraic framework, it is easy to prove the following necessary, but not sufficient, characterization of monotonicity.

Thm. 1 **(Monotonicity of positively recursive predicates.)** The lattice operator $[\Phi_{\mathbf{p}}]$ is monotonic if every instance of **p** in $\Phi_{\mathbf{p}}$ has an even negation depth.

Proof From the fact that relational join, division by the domain of a variable, and projection preserve increasing and decreasing monotonicity, it follows that if $[\varphi_{\mathbf{p}}]$ and $[\xi_{\mathbf{p}}]$ are both increasing (or decreasing) monotonic, then so are $[\varphi_{\mathbf{p}} \wedge \xi_{\mathbf{p}}]$, $[(\forall \mathbf{x})\ \varphi_{\mathbf{p}}]$, and $[(\exists \mathbf{x})\ \varphi_{\mathbf{p}}]$. Relational

---

[2] Negation counting is the same if disjunction, $\vee$, is included. If material implication, $\Rightarrow$, is included, the antecedent picks up an additional negation depth, e.g., the negation depth of **p** in $\mathbf{p} \Rightarrow \mathbf{q}$ is 1. Using these rules, negation depth is invariant under the usual axioms for the predicate calculus.

www.manaraa.com

complement turns an increasing monotonic operator into a decreasing monotonic operator, and vice versa, i.e., $[\sim\varphi_{\mathbf{p}}]$ is increasing monotonic iff $[\varphi_{\mathbf{p}}]$ is decreasing monotonic. The identity operator is increasing monotonic. Since $[\Phi_{\mathbf{p}}]$ is built from the identity operator using conjunction, quantification, and negations of even depth, the result follows.

Cor.2 If $\mathbf{p}$ occurs at even negation depth in $\Phi_{\mathbf{p}}$, then the intended model for the base logic extended with the assignment of $[\Phi_{\mathbf{p}}]\uparrow\omega$ to the predicate symbol $\mathbf{p}$ is a model of the base logic extended by the predicate symbol $\mathbf{p}$ and the defining axiom $\mathbf{p}(x_0,\ldots x_n) \Leftrightarrow \Phi_{\mathbf{p}}$.

Every stratified Horn clause program ***P*** can be automatically converted into a sequence of recursive predicate definitions of the form $\mathbf{p_i}(x_0,\ldots xn)\Leftrightarrow\Phi_{\mathbf{p_i}}$, where $\Phi_{\mathbf{p_i}}$ refers only to predicates $\mathbf{p_j}$ with j i, and where$\mathbf{p_i}$ occurs at zero negation depth within $\Phi_{\mathbf{p_i}}$. The order of definition of the predicates $\mathbf{p_0}$, ... $\mathbf{p_m}$ is the same as their stratification in ***P***. The formula $\Phi_{\mathbf{p_i}}$ is just the (universally quantified) disjunction whose clauses are the bodies of the clauses in ***P*** whose head is $\mathbf{p_i}$. This proves the following theorem.

Thm. 3 **(Expressive power of stratified Horn clause programs.)** Recursive predicate definitions with even negation depth of recursion have as much expressive power as stratified Horn clause programs.

The predicate calculus permits the explicit use of quantifiers, disjunction, and negation, and recursive predicate definition does not restrict recursion to zero depth. Thus, recursive predicate definition is notationally richer than stratified Horn clause programs.

Another important quality of the extended model is that it preserves initiality. In the terms of Goguen [9], it introduces neither junk nor confusion.

Thm. 4 **(Preservation of initiality.)** If the intended model of the base logic is an initial model, then the extended model is an initial model.

Proof The extended model does not add to the interpretation of terms, and so if the original model lacks values that are not assigned to terms in the logic, then the extended model also lacks unused terms. (No junk is introduced.) The relation $[\Phi_{\mathbf{p}}]\uparrow 0$ includes, by definition, only those tuples

www.manaraa.com

that can be proven to satisfy p given no assumptions about any other tuples satisfying **p**, and assuming that one can prove what tuples satisfy other predicates. Each $[\Phi_{\mathbf{p}}]\uparrow(i+1)$ contains only those tuples that can be proven to satisfy **p** given the tuples that are in $[\Phi_{\mathbf{p}}]\uparrow i$. By induction, a tuple satisfies p only if one can prove that it satisfies p. (No confusion is added.)

4.4 Extensions to The Intended Model

Chapter 2 describes an intended model for the base logic. This model assigns domains to each of the sorts, and relations over those domains to each of the predicates. This model can be extended to general formulae as described in Section 4.2.

The intended model is extended as described in the previous section for any sequence of recursively defined predicates, providing all recursion occurs at even negation depth. As usual, the relation assigned to a predicate **p** is written [**p**], and is called the denotation of **p**.

4.5 The Semantic Map

Given a formula φ, its denotation in the intended model is a relation [φ] whose columns are the free variables of φ, V(φ). Given an input signature I and a result signature Q, where $I \cup Q = V(\varphi)$, a *map* is the triple (φ, I, Q). Define the *semantic map* for (φ, I, Q) to be the function $f_{(\varphi, I, Q)} : {}^{I} \rightarrow 2^{Q}$, where:

$$f_{(\varphi, I, Q)}(i) = \pi_{Q}(\sigma_{i}([\varphi]))$$

This function is often abbreviated f_{φ} when the signatures are implied by the context, or even just f when the entire map is implicit. For any I-tuple i and any Q-tuple q, $q \in f(i)$ iff i and q are equal on their common variables, and $i \cdot q$ is in [φ]. Maps are used in the coming chapters to discuss the semantics of procedures.

www.manaraa.com

5. Program Specification & Compilation

The notions of a program specification, a procedure, a procedure's semantic description, and whether a procedure satisfies a program specification are made formal in the first sections of this chapter. The chapter then proceeds with an exposition of the *procedural calculus*, which gives a set of operators for composing procedures and a set of rules for calculating the composed procedure's semantic description. After this, a compilation algorithm is given that produces a procedure composition that satisfies a given program specification. This chapter closes with a careful statement of the conventions an application program must follow in using Galois programs.

In way of preview, a *procedure* is a unit of computation that operates on a set of variables in a deterministic fashion. The *semantic description* of a procedure is (1) a set of modes describing its use of variables, (2) a precondition, (3) a postcondition, and (4) execution properties describing its behavior relative to the precondition and postcondition. Both the precondition and postcondition are formulae in the logic. These notions are formalized below.

A *program* is a procedure whose semantic description matches a program specification. In the code production described in Chapter 7, procedures are implemented as fragments of C code. Code fragments that realize simple formulae are composed to realize more complex formulae. A fragment that realizes a program specification is wrapped in a function definition, and the resulting C function is the program demanded by the program specification. The procedural calculus provides the mechanism for finding a procedure composition that realizes a specified program. (See Figure 1.)

www.manaraa.com

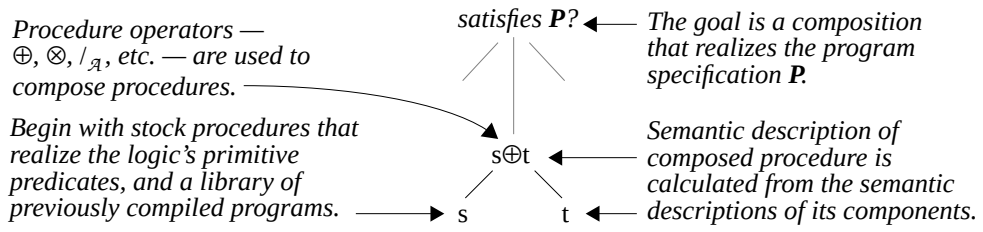


Figure 1: Procedure composition.

Throughout this work, characters in a Gothic typeface, e.g. s, are used to denote procedures. A program specification is denoted by a bold, italicized capital, such as ***P***.

5.1 Procedural Semantics

The definition below formalizes the notion of a procedure.

Def. 2 A *procedure* with inputs from I and results in Q is a function $s : I \rightarrow (\{0, ..|s(i)|\} \rightarrow Q \cup \{\ \})$ that maps any *input tuple* $i \in I$ into a sequence[1] of *result tuples* $s(i)=<q_0, q_1,...>$. Every element of $s(i)$ except perhaps the last is a tuple in Q. $s(i)$ may be finite or infinite. If finite, it may terminate with the end-of-stream value .

Though mathematically defined above, a procedure captures the behavior of a computer program. A procedure accepts an input tuple and generates a sequence of tuples on request. The value is the procedure's signal that it has returned the last available result. If $s(i)$ is infinite, the procedure continues generating output values as long as they are requested. If $s(i)$ is finite but does not end in , the procedure enters an infinite loop after producing the last output value. s is said to *succeed* on input tuple i if s(i) contains a result tuple other than ; otherwise, it is said to *fail*.

[1] A sequence x over the set $\mathcal{A}$ is a function from an initial segment of the natural numbers into $\mathcal{A}$. The domain of x is {k: 0 k<|x|}, where |x| is an ordinal called the *length* of x, |x|ω. The j-th element of x is written x_j. If s is a sequence-valued function, the j-th element of s(i) is written $(s(i))_j$.

www.manaraa.com

The *execution properties* of a procedure s with signatures I and Q relate its behavior to a formula φ that it is intended to realize, or describe when it terminates. The free variables of φ must equal the union of I and Q. Recall from Section 4.5 that f is the function that maps any I-tuple i to the set of Q-tuples each of which together with i satisfies φ. f is used in the definition of execution properties, below.

Def. 3 Let s be s a procedure with input signature I and result signature Q, and let θ and φ be formulae called the precondition and postcondition, where $V(\theta) \subseteq I$ and $V(\varphi) = I \cup Q$. Given that i ranges over all tuples satisfying θ, s, relative to θ and φ:

is *sound*	if $q \in f_{\mathbf{P}}(i)$ for every q in s(i),
terminates	if s(i) ends in ,
is *effective*	if it is sound, terminates, and s(i) contains precisely one output tuple q when $f_{\mathbf{P}}(i) \neq \{\}$,
is *complete*	if it is sound, and for every $q \in f_{\mathbf{P}}(i)$ there is a $q' \in s(i)$ that is isomorphic to q (i.e., q::q'),
is a *function*	if it is effective, complete, and terminates, and
is a *total function*	if it is a function and $\|s(i)\| = 1$.

Note that a procedure can be both effective and complete only if, for each input tuple, the output tuple is unique up to data structure isomorphism. This is the reason such a procedure is labeled a function.

A *semantic description* characterizes the set of procedures with common input and result signatures that realize a formula over a given precondition, displaying similar execution behavior.

Def. 4 A *semantic description* is (1) an input signature I, (2) a result signature Q, (3) a precondition θ, where $V(\theta) \subseteq I$, (4) a postcondition φ, where $V(\varphi) = I \cup Q$, and (5) a set of semantic properties that always includes *sound*.

Def. 5 A procedure *satisfies* a semantic description if it has the stated input and result signatures as the semantic description, and the stated execution properties hold for the procedure relative to the stated precondition and postcondition.

www.manaraa.com

Procedures that satisfy a common semantic description may differ in several ways. Effective procedures that are not functions may return different output tuples for the same input tuple. Complete procedures may present output tuples in different order, and unless termination is specified, some procedures may terminate and others not on the same input tuple. When termination is not specified, some effective procedures may terminate or loop endlessly for input tuples for which there is no output tuple. Note that if a complete procedure terminates, then for each input tuple, the set of output tuples is finite.

A *program specification* is a semantic description, augmented with a program name, where the postcondition is simply a named predicate. A programmer specifies a program that is either effective or complete, but not both. The former is the case when the programmer desires *any* qualified output tuple, if it exists, and the latter when *all* qualified output tuples are desired. A program that produces a data structure is usually effective, e.g., it produces *a* heap, *a* B-tree, or *a* hash table that satisfies the program specification, whenever one exists.

An effective procedure is shown to be a function by proving that it has a unique output tuple for each input tuple. A procedure that has no output variables is necessarily a function. It is called a *characteristic function* because it serves to determine whether an input tuple belongs to a qualified set, i.e., whether an input tuple satisfies some condition which the function is said to *verify.*

Most of the examples in this work show programs that effectively realize their specifications. In order to compose an effective realization of a predicate, it is often necessary to have complete realizations of component predicates. For example, to iterate through an array finding *an* element that satisfies some criterion, there must be a procedure that generates *all* of the possible index values for the array.

While the above notions are straight-forward, a few examples will serve to make them concrete.

Ex. 6 Let c be the procedure with input signature {x, y} and result signature {x, y} such that:

www.manaraa.com

c({a/x, b/y}) = <{a/x, b/y}, >, if x<y and

c({a/x, b/y}) = < >, otherwise.

Notice that there are no output variables. c is complete for the predicate x<y. It terminates, and it is a function, though not a total function. Or to put it another way, it is a characteristic function that verifies the condition x<y.

Ex. 7 Let i be the procedure with input signature {} and result signature {x} that generates the sequence of all prime numbers:

i({}) = < {2/x}, {3/x}, {5/x}, ...>.

This procedure is complete for the predicate **`prime(x)`**. It does not terminate.

Ex. 8 Let w be the procedure with input signature {x} and result signature {x, y} that returns in integer y the sum x+2:

w({a/x}) = <{a/x, a+2/y}, >.

This procedure is both effective and complete for the predicate y=x+2. It is a total function.

The last two procedures will be used in the following section to demonstrate procedural composition. More examples will be developed in the coming section.

5.2 Composing Procedures

The central idea of the procedural calculus is that procedures that realize the component formulae of a larger formula can be composed into a larger procedure that realizes the larger formula. The procedural calculus comprises a few procedure operators and a set of rules that calculate the semantic description of procedures composed with these operators. Each rule concerns one logical construction and describes how to calculate the semantic description of the composed procedure from the semantic properties of the semantic description of the component formulae. The presentation below is organized according to the various logical constructions. First the rules for disjunction are presented,

www.manaraa.com

followed by the rules for conjunction, the other boolean operators, existential quantification, universal quantification, and finally, recursion.

Readers may benefit by reading this section in parallel with Section 7.2, which presents the C code corresponding to the below compositions. This section formally defines what the code there must achieve, and the code there makes concrete the more abstract presentation below.

Disjunction

To produce *an* output tuple that satisfies $\varphi \vee \phi$, a procedure can first try to find a tuple that satisfies φ and, if that fails, try to find a tuple that satisfies ϕ. If there are effective procedures that realize φ and ϕ separately, and the procedure that realizes φ terminates, then these are easily composed in this fashion. This procedure operation is called *choice*.

Def. 9 *Choice.* The procedures s and t must have the same input and result signatures. Define $s \otimes t$ by:

$(s \otimes t)(i) = t(i)$, if $s(i) = < >$, and

$(s \otimes t)(i) = s(i)$, otherwise.

For example, let q and z be procedures with input signature {x} and output signature {x, y}, where:

$q(\{i/x\}) = <\{i/x, j/y\}, >$, if $j^2 = i$,

$q(\{i/x\}) = < >$, if i is not a square, and

$z(\{i/x\}) = <\{i/x, 0/y\}, >$.

Then $q \otimes z$ returns the square-root of x if x is a square and returns zero otherwise. It provides an effective realization of $(y^2 = x \vee y = 0)$.

To produce *all* tuples that satisfy $\varphi \vee \phi$, a procedure can first produce all tuples that satisfy φ, and then, assuming that the former are finite in number, the procedure can produce all the tuples that satisfy ϕ.If there are *complete* procedures that realize φ and ϕ separately, and if the procedure that realizes φ terminates, then

www.manaraa.com

these can be composed to provide a complete realization of $\varphi \vee \phi$ through procedure *concatenation*.

Def. 10 *Concatenation.* The procedures s and t must have the same input and result signatures. Define $s \oplus t$ by:

$(s \oplus t)(i) = <(s(i))_0, \ldots (s(i))_n, (t(i))_0, (t(i))_1, \ldots>$, if $(s(i))_{n+1} =$, and

$(s \oplus t)(i) = s(i)$, otherwise.

This formalizes the notion of "producing all of s and, when that completes, producing all of t." Note that $s \oplus t$ terminates precisely when s and t both terminate. Given a sequence $s_1, \ldots s_m$, the usual notation is used for repeated sums: $\Sigma_{1 \le k \le m} s_k = s_1 \oplus s_2 \oplus \ldots s_m$.

Note that neither choice nor concatenation are commutative, i.e., it is generally the case *neither* that $s \otimes t = t \otimes s$, *nor* that $s \oplus t = t \oplus s$. This is reflected in the calculation of semantic properties for the composed procedures in the lemma below.

Lm. 11 If s and t are procedures with the same input and result signatures, s realizes φ under precondition α, and t realizes ϕ under precondition β, then $s \otimes t$ and $s \oplus t$ realize $\varphi \vee \phi$, $s \otimes t$ under the precondition $\alpha \wedge (\sim\varphi \Rightarrow \beta)$ and $s \oplus t$ under the precondition $\alpha \wedge \beta$. The execution properties of $s \otimes t$ and $s \oplus t$ with respect to $\varphi \vee \phi$ are determined from the execution properties of s with respect to φ and t with respect to ϕ, according to the table below.

the composition ...	if s is ...	and t is ...
$s \otimes t$ is effective	effective & terminating	effective
$s \otimes t$ terminates	terminating	terminating
$s \oplus t$ is complete	complete & terminating	complete
$s \oplus t$ terminates	terminating	terminating

Proof Concatenation requires the precondition $\alpha \Rightarrow \beta$ because both s and t are applied to the input tuple. Choice only requires the weaker precondition $\alpha \wedge (\sim\varphi \Rightarrow \beta)$, because t is applied to the input tuple only if s

www.manaraa.com

fails, i.e., if ϕ does not hold. The rules in the table summarize the comments above about the behavior of these compositions. Note that in the first and third columns, s must terminate to guarantee the effectiveness and completeness of the compositions, because s is applied first, and in order for t to be applied, s must terminate.

Conjunction

The procedural operation that realizes conjunction is *sequential composition*. It realizes the conjunction φ∧ϕ by first producing a result tuple that satisfies φ and, using this, producing a result tuple that satisfies φ∧ϕ. The component procedures must coordinate their use of variables: they must not have common output variables, and the input variables for the second procedure must either be input variables to the composed procedure, or must have been output by the first procedure. The definition of sequential composition first requires the definition of an operation that augments a procedure with new variables that are "passed through" unchanged.

Def. 12 *Signature augmentation.* Let s be a procedure with input signature $\mathcal{I}$ and result signature $\mathcal{Q}$ = {x, ...z}, and let $\mathcal{A}$ = {u,...w} be any signature disjoint from $\mathcal{Q}-\mathcal{I}$. Let i be a tuple over $\mathcal{I}$, and j a tuple over $\mathcal{A}$ such that i and j are equal on their common variables (if any). Then:

$(s\hat{}\mathcal{A})(i\cdot j) = s(i)\cdot j$.

In other words, $s\hat{}\mathcal{A}$ adds the variables in $\mathcal{A}$ to the input and result variables of s, by passing values for them through unmodified. The operation is defined broadly, so that $\mathcal{A}$ may include variables that are already input variables to s. Sequential composition is now defined using the operations of signature augmentation and procedure concatenation, previously defined.

Def. 13 *Sequential composition.* Let s be a procedure with input signature $\mathcal{I}$ and result signature $\mathcal{Q}$, t a procedure with input signature $\mathcal{J}$ and result signature $\mathcal{R}$, and assume that their output signatures are disjoint, i.e., that $(\mathcal{R}-\mathcal{I})\cap(\mathcal{Q}-\mathcal{J})=\{\}$. Let $\mathcal{A} = \mathcal{I}\cup(\mathcal{J}-\mathcal{Q})$ and $\mathcal{B} = \mathcal{R}\cup\mathcal{Q}$. The composition

www.manaraa.com

t (s) with input signature $\mathcal{A}$ and result signature $\mathcal{B}$ is defined at each $\mathcal{A}$-tuple, a, by:

$$(t\ (s))(a) = \Sigma_{0\ k<|s|} t^{\wedge}(Q \cup \mathcal{I})\ (s^{\wedge}\mathcal{A}(a))_k.$$

Operationally, this says: apply s to the I part of a; augment this sequence with the non-I part of a; apply t to each element $(s^{\wedge}\mathcal{A}(a))_k$ of the resulting sequence; augment the result with the non-$\mathcal{I}$ part of $(s(a))_k$; and finally, concatenate these sequences. More simply, it says: feed a tuple to s, feed the first resulting tuple to t, let t produce its output sequence, then go back for the next tuple from s, where all variables not used by s or t are passed through them unchanged.

For example, consider the procedure i that completely realizes **`prime(x)`** and the procedure w that realizes the function **`y=x+2`**. (Both were described in the previous section). The composed procedure w(i) pairs each prime with the number that is two greater:

(w(i))({}) = <{2/x, 4/y}, {3/x, 5/y}, {5/x, 7/y}, ...>

Thus, w(i) is a complete realization of the formula: **`prime(x)`** $\wedge$ **`y=x+2`**. Let j be the characteristic function that verifies that y is prime:

j({a/y}) = <{a/x}, >, if a is prime, and

j({a/y}) = < >, otherwise.

Then j(w(i)) produces all pairs of primes whose difference is two:

j(w(i))({}) = <{3/x, 5/y}, {5/x, 7/y}, {11/x, 13/y}, ... >

In other words, j(w(i)) is a procedure that is a complete realization the formula: **`prime(x)`** $\wedge$ **`y=x+2`** $\wedge$ **`prime(y)`**. Note that i is a *complete* realization of the predicate **`prime(x)`** while j is a *characteristic function* for the predicate **`prime(y)`**. The lemma below summarizes the properties of sequential composition.

Lm. 14 If s realizes φ under precondition α, and t realizes ϕ under precondition β, then if t(s) can be formed, it realizes $\varphi \wedge \phi$ under the precondition $\alpha \wedge (\varphi \Rightarrow \beta)$, and the execution properties of t(s) with respect

www.manaraa.com

to $\varphi \wedge \phi$ are determined from the execution properties of s with respect to φ and t with respect to ϕ, according to the table below.

t(s) is ...	if s is ...	and t is ...
effective	complete	effective
	effective	a total function
complete	complete	complete & terminates
	a function	complete
terminates	terminates	terminates
a function	a function	a function
a total function	a total function	a total function

Proof Each tuple produced by t(s) derives from the application of s, first, to the input tuple, and then of t to one of the result tuples from the *prior* application of s. Because of this order of application, $\alpha \wedge (\varphi \Rightarrow \beta)$ is the weakest precondition that guarantees that (1) α holds prior to the application of s, and (2) for a result tuple of s (together with other input variables to t(s)), β holds.

The rules in the table are fairly simple. In the first two columns, the conditions trivially guarantee that t(s) produce a result tuple whenever one exists. For completeness, in the first case, it is necessary that t terminates, since otherwise t(s) does not produce tuples after the first result instance of s where t fails to terminate. The last two rules follow from the fact that sequential composition generalizes function composition.

Existential Quantification and "True"

A *projection procedure* with input signature I and result signature $Q \subseteq I$ is the total function that merely passes through its result arguments. When $I = Q$, a projection procedure is called an *identity procedure*.

Projection procedures realize existential quantification. Or to frame this

www.manaraa.com

more broadly, existential quantification is only realized constructively. From the programmer's viewpoint, an existentially quantified variable serves to hold an intermediate value that is produced by but not returned from a computation. After the quantified variable is used, a projection function applied through sequential composition removes it, i.e., "projects it off." (Chapter 7 shows how projection procedures are implemented as memory deallocation.) These remarks are formalized in the lemma below.

Lm. 15 If s with result signature Q realizes φ under precondition α, and p is a projection of Q onto Q–$\mathcal{E}$, then p(s) realizes $(\exists \mathcal{E})$φ under precondition α. p(s) is effective, is complete, and terminates precisely when s is effective, is complete and terminates.

Projection procedures can be viewed as realizing the boolean constant **`True`**. $(\exists \mathcal{E})$φ can be viewed as φ∧**`True`**, where the procedure that realizes **`True`** fails to pass on the quantified variables.

The Other Boolean Operators

Given realizations of conjunction, disjunction, and **`True`**, a realization of negation would provide a complete set of boolean operations. This motivates the following definition.

Def. 16 *Procedure Complement*. Given any procedure s with input signature I and result signature Q, the procedure !s is defined at each I-tuple i by:

!s(i) = < > if s(i) = <q, ...> for some$Q$-tuple q,

!s(i) = <i, > if s(i) = < >, and

!s(i) = <>, otherwise.

In short, when s succeeds !s fails, and when s fails and terminates !s succeeds. The procedural complement realizes negation, but it will have a different result signature from s if s has any output variables. This is made rigorous in the lemma below.

www.manaraa.com

Lm. 17 If s with input signature I and result signature Q realizes φ under precondition α, then !s with input and result signature I realizes $\sim(\exists O)\varphi$ under precondition α, where $O=Q-I$ is the output signature of s. !s is a function, and it terminates if s never produces the null sequence, i.e., never enters an infinite loop before producing any output. (This is a slightly weaker condition than saying that s terminates, since it permits infinite sequences.)

Negation can be used directly, but more often, it is implicit in the if-then-else construct, $\varphi\Rightarrow\phi\nabla\theta$. The construct $\varphi\Rightarrow\phi\nabla\theta$ is logically equivalent to $(\varphi\wedge\phi)\vee(\sim\varphi\wedge\theta)$, and it is realized through this rewrite. As pointed out above, procedural complement realizes the negation of a formula— without existential quantification — only when the complemented procedure verifies some condition. Because of this, the realization of $\varphi\Rightarrow\phi\nabla\theta$ requires a procedure s that is a characteristic function for φ, i.e., where $I=V(\varphi)$. This is congruent with the intended purpose of $\varphi\Rightarrow\phi\nabla\theta$, to provide the programmer a construct that corresponds to the **`switch`** statement of C.

Universal Quantification

Universal quantification provides the programmer a way to verify a condition over some set. The only form of universal quantification that can be realized is $(\forall\mathcal{A})\varphi\Rightarrow\phi$, where φ generates values in some set, and ϕ verifies the condition. This formula is realized through the procedural composition defined below.

Def. 18 *Procedure Division.* Given any terminating procedure s with input signature I and result signature Q, a characteristic function t on signature $\mathcal{J}\subseteq Q$, and a signature $\mathcal{A}\subseteq\mathcal{J}$, the procedure $s/_{\mathcal{A}}t$ has input signature I and result signature $Q-\mathcal{A}$ and, for each I-tuple i, $(s/_{\mathcal{A}}t)(i)$ is the $Q-\mathcal{A}$ projection of the subsequence of s(i) where:

$$\pi_{Q-\mathcal{A}}(q)\in(s/_{\mathcal{A}}t)(i)\Leftrightarrow q\in s(i)\wedge(\forall r)((r\in s(i)\wedge\pi_{\mathcal{A}}(r)=\pi_{\mathcal{A}}(q))\Rightarrow r\in t(r))$$

Consider the set of tuples in the sequence s(i) that contain the $\mathcal{A}$-tuple, a, and the set of tuples in the sequence t(i) that also contain the sub-tuple, a. If

www.manaraa.com

these two sets are not equal, remove from s(i) the first set. Continue to pare s(i) by removing such sets for all $\mathcal{A}$-tuples. The result is $(s/_{\mathcal{A}}t)(i)$.

For example, let s be the complete procedure with input signature {b, x} and result signature {b, x, k} that realizes the formula 0 k<b. (The variable x is an integer array that is passed through unchanged by s.) Let t be the procedure with input and result signature {b, x, k} that verifies x[k]>0. Then $s/_{\{k\}}t$ is the procedure with input and result signature {b, x} where:

$s/_{\{k\}}t(\{b_0/b, x_0/x\}) = <\{b_0/b, x_0/x\}, >$, if $x_0[k]>0$ for $0\ k<b_0$, and

$s/_{\{k\}}t(\{b_0/b, x_0/x\}) = < >$, otherwise.

In other words, $s/_{\{k\}}t$ realizes the formula $(\forall k)(0\ k<b \Rightarrow x[k]>0)$. Note the similarity between procedure division and relational division. The former is used to procedurally realize universal quantification, while the latter is used to define the declarative semantics of universal quantification. To realize $(\forall \mathcal{A})\varphi \Rightarrow \phi$, s must completely realize φ, since otherwise some values may not be verified by t. The lemma below adds to the previous discussion a rule for calculating preconditions.

Lm. 19 If s with input signature I and result signature Q is a complete realization of φ under the precondition α, t verifies ϕ with signature $\mathcal{R} \subseteq Q$ under precondition β, and $\mathcal{A}$ is a signature that is a subset of $\mathcal{R}$, then $s/_{\mathcal{A}}t$ is a complete and terminating realization of $(\forall \mathcal{A})\varphi \Rightarrow \phi$, under the precondition $\alpha \wedge (\varphi \Rightarrow \beta)$.

Proof The semantic realization of $s/_{\mathcal{A}}t$ is discussed above. As to the precondition, $(\varphi \Rightarrow \beta)$ is sufficient, rather than β, because every tuple given to t is generated by s, and hence satisfies φ.

Literals and Argument Binding

So far in the discussion of the procedural calculus, the formal and actual arguments to predicates have been assumed identical. That is, there has been no description of how to realize **`p(f(x), y)`**, for example, given a realization of **`p(u, v)`**. Variable binding turns a procedure for a predicate into a procedure for a

www.manaraa.com

positive literal built on the predicate. This is formalized in the definition below.

Def. 20 *Variable binding.* Let l be a positive literal formed from the literal $p(x_0,...x_z)$, i.e., l is $p(t_0,...t_z)$, where $t_0,...t_z$ are terms. Let s be a procedure that realizes p, where:

$x_b,...x_d$ are the pure input variables (mode is *in*),

$x_e,...x_g$ are the consumed input variables (mode is *inx*),

$x_h,...x_j$ are the output variables (mode is *out*), and

$y_0,...y_m$ are the variables in the terms $t_b,...t_d$.

If the terms $t_e,...t_g$, $t_h,...t_j$ (which correspond to the consumed input variables and output variables of s) are simple variables, then define the procedure s^l with input signature $\mathcal{I}=\{y_0,...y_m,t_e,...t_g\}$, consumed input signature $\mathcal{D}=\{t_e,...t_g\}$, and output signature $\mathcal{O}=\{t_h,...t_j\}$, on an input tuple $i=\{w_0/y_0,...w_m/y_m, v_e/t_e,...v_g/t_g\}$, as follows. Let $v_b,...v_d$ be the values of terms $t_b,...t_d$ applied to the input tuple i. If the k-th result of s on i (as passed through the binding) is:

$$\{v_b/x_b,...v_d/x_d, v_h/x_h,...v_j/x_j\} = s_k(\{v_b/x_b,...v_d/x_d, v_e/x_e,...v_g/x_g\}),$$

then the k-th result of s^l is:

$$s^l_j(i) = \{w_0/y_0,...w_m/y_m, v_h/t_h,...v_j/t_j\}.$$

The important condition in the above definition is that the terms bound to the consumed input arguments and the output arguments must be simple variables. Obviously, the procedure s^l has the same semantic properties as s. This is formally stated below.

Lm. 21 The procedure s^l is sound, effective, complete, and terminates when s is sound, effective, complete, and terminates, respectively.

Chapter 7 deals with generating code for terms. Some terms give rise to preconditions, thus s^l may have preconditions that s does not. This is discussed in that chapter.

www.manaraa.com

Recursion

Only effective realization is supported for recursively defined predicates. (Obviously, if a recursively defined predicate can be proved a function, then its effective realization is also complete.) Let ***P*** be a program specification with postcondition predicate **p**, which has the recursive definition $\mathbf{p} \Leftrightarrow \Phi_{\mathbf{p}}$. Let E_s be a composition of sequence-valued functions that realizes $\Phi_{\mathbf{p}}$ such that (1) s is the procedure symbol assigned to **p** in E_s, (2) s is assumed to be effective for **p**, (3) given this assumption, E_s is an effective realization of $\Phi_{\mathbf{p}}$, and (4) s and E_s have the same input and result signatures. Given this, a least fixed-point can be defined for s so that it is indeed an effective realization of **p**.

Let s_0 be the unique procedure with same signature as s that has <> as its output for any input tuple, i.e., s_0 is the procedure that enters an endless loop for any input tuple. Recursively define s_{k+1} to be E_{s_k}, i.e., the procedure that results from the procedure composition E_s, with s_k in the place of s. Then the least fixed-point for s is the sequence limit as k increases without bound of s_k. This is formalized in the lemma below.

Lm. 22 For any input tuple i, the result tuple returned $s_k(i)$ is in the relation $\sigma_i([\Phi_{\mathbf{p}}]\uparrow k)$, that is, $\pi_I([\Phi_{\mathbf{p}}]\uparrow k)$ is the set of input tuples for which s_k returns a result tuple q, and $i \cdot q \in [\Phi_{\mathbf{p}}]\uparrow k$.

Proof By induction. This is trivially true for k=0. If it is true for any k, then it is true for k+1, because given the inductive assumption, $E_{s_{(k+1)}}$ is an effective realization of $[\Phi_{\mathbf{p}}]\uparrow(k+1)$.

The immediate corollary is that s_k is a monotonic sequence of partial functions, and that it therefore has a least fixed-point s, and that this is an effective realization of **p**.

Determining that recursively defined procedures terminate is a little bit trickier. It was fair to assume that s_0 was effective precisely because it was the most undefined procedure, the one that never terminates. If E_s is shown to be a terminating composition *without* assuming this of s, then s is in fact terminating. This does not help much, because the composition rules above do not support any

www.manaraa.com

such deduction! (All compositions produce procedures that may fail to terminate on some input tuples if any component fails to terminate.) Hence, for recursive procedures, proving termination is thrown back on the usual methods, such as showing that the recursion is grounded.

Termination is only at issue for input tuples for which there is no result tuple that satisfies the predicate, i.e., for input tuples that should cause the procedure to fail. If the recursive predicate realized by an effective procedure can be satisfied for any input tuple, then the procedure is terminating. Thus, one way to prove termination is to show that for any input tuple that satisfies a procedure's precondition, there exists a result tuple that satisfies the predicate. This is a more general method of proving termination than showing that a recursion is grounded.

Summary of Procedure Operations

The table below provides a summary of procedure operations, the logical constructs they realize, and their preconditions. In it, s and t are procedures that realize the formulae φ and ϕ under preconditions α and β, or s is a procedure that realizes the predicate in the literal l under precondition α and γ is the precondition for evaluating l's terms. O is the output signature of s.

Operation	*Realizes*	*Precondition*	*Purpose*
$s \otimes t$	$\varphi \vee \phi$	$\alpha \wedge \beta$	*Choice* effectively realizes disjunction.
$s \oplus t$	$\varphi \vee \phi$	$\alpha \wedge \beta$	*Concatenation* completely realizes disjunction.
$s^{\wedge}\mathcal{A}$			*Signature augmentation* merely passes through variables not used by s.
t(s)	$\varphi \wedge \phi$	$\alpha \wedge (\varphi \Rightarrow \beta)$	*Sequential composition* realizes conjunction.
p(s)	$(\exists \mathcal{A})\varphi$	α	p projects off the variables of $\mathcal{A}$.
!s	$!(\exists O)\varphi$	α	*Complement* realizes negation.
$s/_{\mathcal{A}}t$	$(\forall \mathcal{A})\varphi$	$\alpha \wedge (\varphi \Rightarrow \beta)$	*Division* realizes universal quantification.
s^{l}	l	$\alpha \wedge \gamma$	Variable binding for the literal l.

www.manaraa.com

Operation	*Realizes*	*Precondition*	*Purpose*
E_S	φ	α	Indicates least fixed-point realization of a recursively defined predicate.

Note that the precondition for sequential composition and division may be weaker than the second component's precondition and, regardless, is weaker than the conjunction of the two components' preconditions. This is the underlying mechanism that allows the programmer to build run-time checks into a predicate definition, making the predicate more complex, while weakening the precondition of the program that realizes the predicate.

5.3 From Program Specifications to Code

The algorithm described below returns a procedure composition that satisfies a given semantic description, providing that a prerequisite assertion generated by the algorithm holds. This algorithm relies on a stock of procedures built into the compiler that realize the primitive predicates and a library of previously constructed programs. The algorithm employs the following variable types:

`lgclForm`	Represents a logical formula.
`execProp`	Represents the desired execution properties and the desired input and output signatures.
`procComp`	Represents a procedure composition and its precondition.

The following functions are used:

`ConstructionOf(F)`	Returns an enumerated type that indicates the construction of the formula **F**, i.e., whether it is a conjunct, disjunct, etc.
`NeedComplete(P)`	Returns true if the execution properties **P** specify complete realization.
`Fn(F)`	Returns the n-th component of the formula **F**.
`En(F, E)`	**E** specifies the desired signatures and execution properties for **F**. **En** returns the needed signatures and execution properties for the n-th component of **F**.
`Concat(P1, P2)`	Returns the concatenation of the procedures P1 and P2 and the precondition for the composition.

www.manaraa.com

Choice(P1, P2)	Returns the choice composition of the procedures P1 and P2, and the precondition for the composition.
...(P1, P2)	For each procedure operator, there must be a function that returns the composition procedure and its precondition.
MakePrim(F, E)	Given a literal **F** built from a primitive predicate and desired execution properties **E**, this returns the procedure that realizes the literal.
MakeCall(L, F, E)	Given a literal **F** built from a defined predicate and desired execution properties **E**, this searches the library **L** for a program that realizes the predicate with the desired execution properties, and returns the procedure that calls this program to realize the literal.

The algorithm is presented as a recursive function, **Compile**. Recursion ends in the construction of literals using the functions **MakePrim** and **MakeCall**. The former encodes the compiler's knowledge of how to realize the primitive predicates. The latter makes use of a library of programs that have been previously compiled. **MakePrim** will fail if there is no way to realize the primitive predicate with the desired execution properties. (This is discussed in Chapter 7.) **MakeCall** will fail if the library does not contain a previously compiled program that realizes the desired predicate with the desired execution properties. Either will fail if the literal uses complex terms for consumed input arguments to or output arguments from the predicate. The pseudo-code for **Compile** is shown below.

```
procComp Compile ( // calculate composed procedure &
                   // precondition from:
   lgclForm Formula, // logical formula to realize
   execProp Prop // desired signatures & exec. properties
){
   procedure Proc1, Proc2;
   switch (ConstructionOf (Formula)) {
   case Disjunction:
      if (NeedComplete (Prop)) {
         Proc1 = Compile (F1(Formula), E1(Formula, Prop));
         Proc2 = Compile (F2(Formula), E2(Formula, Prop));
         return Concat (Proc1, Proc2);
      }
      else
         Proc1 = Compile (F1(Formula), E1(Formula, Prop));
```

www.manaraa.com

```
            Proc2 = Compile (F2(Formula), E2(Formula, Prop));
            return Choice (Proc1, Proc2);
         }
      case Conjunction :
            Proc1 = Compile (F1(Formula), E1(Formula, Prop));
            Proc2 = Compile (F2(Formula), E2(Formula, Prop));
         return SeqComp (Proc1, Proc2, Proc);
      case Negation:
         ...
      case PrimitiveLiteral:
         return MakePrim (Formula, Prop);
      case OtherLiteral:
         return MakeCall (Library, Formula, Prop);
      }
}
```

This algorithm assumes that when there is a choice of execution properties that **E1** and **E2** can assign the component formulae so that the composed formula has its desired properties — i.e., when more than one rule of the procedural calculus might apply — that **E1** and **E2** oracularly select a rule that makes the rest of compilation succeed, if such a rule exists. The algorithm is actually implemented as a backtracking search that tries one rule, and if that fails, then returns to the choice point and tries another. (The first example in Chapter 8 discusses this further.)

The algorithm above calculates a precondition, β, that is required by the composed procedure. The programmer specifies a desired precondition, α, in the program specification. In order for the calculated composition to satisfy the program specification, $\alpha \Rightarrow \beta$ must hold. This is the prerequisite assertion on which successful code generation depends.

As described in Section 3.3, the compilation process has several steps. These are repeated below, but with specific reference to the compilation algorithm above and the form of the prerequisite assertion.

www.manaraa.com

(1) The programmer defines a predicate and specifies a program that realizes it.

(2) **`Compile`** the program. If the program fails to compile, display the points of failure and permit the programmer to change the predicate definition or program specification.

(3) Verify the prerequisite assertion $\alpha \Rightarrow \beta$. The programmer does this using theorem proving tools.

(4) Generate C code from the procedure composition returned by **`Compile`**. Add the compiled program to the library. Add theorems about the program specification, generated during verification of the prerequisite assertion, to the theorem library.

The compilation algorithm given above infers the order in which component formulae are made true from their textual order in conjunctions and disjunctions. If compilation fails for this ordering of formulae, for example, because the first formula in a conjunct requires more input to be realized, the compilation algorithm does not try a different ordering. It is clearly desirable to relieve the programmer of this responsibility. On the other hand, it is impossible for the compilation algorithm to look at all logically equivalent forms of a postcondition. This gives rise to the notion that a small number of rewrite rules that preserve logical equivalence could significantly ease the programmer's task without too much burdening the compiler. (See Figure 23.)

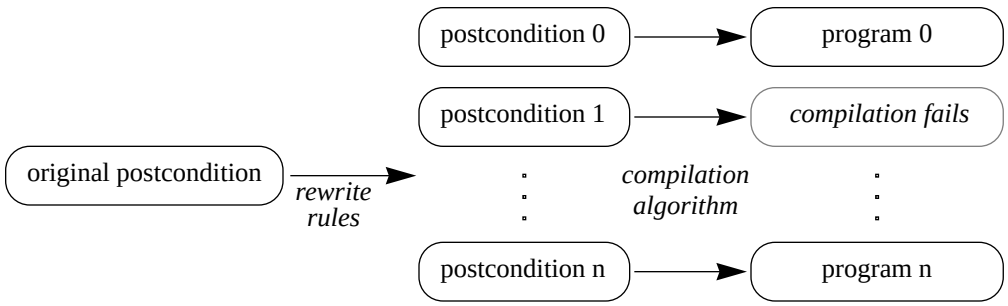


Figure 23: Combining logical rewrites with computation graph construction.

These rewrite rules would be applied to a formula during the compilation algorithm if the formula fails to compile. This would enlarge the number of formula for which the compiler successfully generates a procedure composition. Alternatively, the compiler, after showing the programmer where compilation fails, could allow the programmer to select from the set of rewrite

www.manaraa.com

rules to apply for a further try at compilation. In this scheme, the programmer interacts with the compiler in searching for an order of computation that realizes the desired formula.

5.4 The Contract between the Calling Program and a Galois Program

In practice, a Galois program is a callable unit of execution (such as a C function) invoked from an application written in a traditional language (such as C). A Galois program has two modes of invocation, known as *entry points*. The *return-first* entry point binds the input arguments and calculates the first result tuple. The *return-next* entry point calculates the next result tuple. An effective program only needs the return-first entry point, since it only returns one tuple.

A program has a return status. After it is invoked, either (1) the program *succeeds*, returning the status *success* and the next qualified output tuple, (2) the program *fails*, returning the status *failure*, indicating that there are no more qualified output tuples, or (3) the program loops forever. As described earlier, this behavior is determined by the program's output sequence s, where each invocation of the program via the *return-next* entry point advances the position k in the sequence s(i). The program succeeds if $(s(i))_k$ is a tuple, fails if $(s(i))_k$ is , and loops forever if k |s(i)|. Invocation via the*return-first* entry point sets k to 1 and causes the appropriate behavior for k=1.

Each successful invocation of a Galois program changes the current memory state. As described in Section 2.8, a memory state is equivalent to a state specific submodel of the intended model. Hence, each successful invocation of a program can be viewed as moving from one such submodel to a second. In the simplest case, this change to the state specific submodel can be described as follows. (1) Remove all data structure instances that are bound to *inx* arguments, and all their components, from the submodel. (2) Add to the submodel all data structure instances, and their components, that are in the output tuple. This is shown in Figure 24, below.

www.manaraa.com

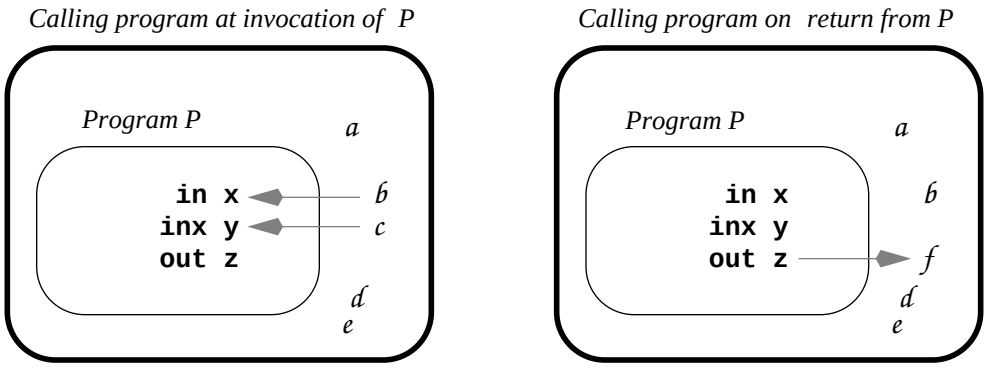


Figure 24: Invoking a Galois program.

Unfortunately, this simplest case does not always hold. For example, in Figure 24, if *a* and *c* have common parts, then removing *c* would change the memory state of the calling program in an unexpected way. This is the problem of side effects that results from data aliasing. Aliasing can result in even worse problems. For example, if *b* and *c* have common parts, the Galois program may not behave correctly (may not meet its specification), because the value bound to **x** might change in unpredictable ways during the program's execution.

The problems that arise from aliasing are prevented by the application abiding by certain conventions. These conventions, and the subsequent guaranteed behavior of called Galois programs, can be viewed as a contract between the application and Galois programs. (See [41] for a discussion of this view of programming.) The compilation process guarantees that Galois programs abide by their end of the contract. The application programmer is responsible for seeing that the calling application does likewise. (The fact that the compilation process guarantees this for Galois programs argues that as much of the application's work as possible should be realized through Galois programs.)

The first requirement of the contract is that the application and the invoked Galois program use correctly a common memory allocation mechanism. There are two rules that determine the correct usage of the memory allocation mechanism.

(1) Only memory that is currently allocated from the memory allocation mechanism is included in the set of data structure instances that characterize the application's memory state, and only this memory may be modified.

www.manaraa.com

(2) Only memory that is currently allocated from the memory allocation mechanism can be deallocated.

The memory allocation mechanism can be viewed as dynamically extending or contracting the program's address space. The program's memory state is the state of allocated memory, which is viewed as a state specific submodel of the logic, i.e., a collection of data structure instances and their components.

The next few conditions are ones that the application guarantees for the invoked Galois program. If the application fails to uphold these conditions, the Galois program may execute incorrectly.

(3) The application passes to a Galois program, as its input tuple, a set of data structure instances that satisfy the Galois program's precondition.

(4) Each data structure instance passed as a consumed input argument is disjoint from any of the other input arguments (whether or not consumed).

(5) Each data structure instance passed as a consumed input argument resides in a collection of memory chunks, one for each **`struct`** and array component, that were allocated from the memory allocation mechanism with the appropriate size. (The appropriate size for a **`struct`** or array follows from its representation in C data structures, as described in Chapter 7.)

Given the assumptions made so far, a Galois program guarantees several things for the application. These are listed below.

(6) The Galois program satisfies its program specification.

(7) On a Galois program's return from a failed execution, the state of memory is unchanged from when the Galois program was invoked.

(8) On a Galois program's return from a successful execution, memory is changed only by (A) the deallocation of all memory in data structure instances that were bound to the consumed input arguments, followed by (B) the allocation and assignment of values to new memory in which data structure instances bound to the output values now reside.

From the point of view of the application, a destructive update is functionally equivalent to the deallocation of consumed input arguments, followed by the allocation of memory for the output arguments. That some output arguments reside partly in the same memory that had been used for consumed input

www.manaraa.com

arguments can be viewed as an accident of memory allocation. In fact, whenever possible, a Galois program often short-circuits the memory manager by directly updating the data structure instances that are passed as consumed inputs. This is transparent to the application, except for the gain in efficiency.

The rules above do not prevent all problems from aliasing. The application may still pass a consumed input argument that is aliased with or referenced from a data structure instance unknown to the Galois program. (Note that rule (8) is phrased in terms of what memory is deallocated, not in terms of what data structure instances are changed or removed.) When an application does this, it is most likely a mistake. There is no way that Galois programs can detect or prevent this. And as long as the application abides by the rules above, the Galois programs will continue to behave correctly, despite any destruction they are doing to data structure instances or pointers they leave dangling in the "outside" world. Of course, if the application mistakenly passes as a consumed input a data structure instance on which the application later relies, it is unlikely that the application will be able to keep up its end of the bargain for long. The next rule below sharpens rule (4), to prevent this last problem due to aliasing. Clearly, this rule is not required for Galois programs to behave correctly. The previous rules suffice. It is more an indication of risky programming that most times will cause the application to violate one of the previous rules.

(9) Each data structure instance passed as a consumed input argument is disjoint from any data structure instance in memory except its own components. All pointers (with the exception of the argument binding) that point into the data structure instance are also part of it.

Given that the above rule is followed, the guarantee in (8) can be sharpened. The way a Galois program changes memory state now satisfies the clean picture shown in Figure 24. The final rule below states this sharpened guarantee.

(10) On a Galois program's return from a successful execution, memory is changed only by (A) the removal of data structure instances that were bound to the consumed input arguments, and their pieces, and (B) the addition of data structure instances bound to the output values.

www.manaraa.com

The remainder of this work assumes that the application upholds its end of the above contract.

www.manaraa.com

6. Computation Graphs: An Intermediate Representation

The previous chapter described a procedural calculus and a compilation algorithm that generates a procedure from a program specification. The generated procedure is represented as an expression in the procedural calculus. A different, graphical form for representing procedures is presented below. This form serves as an intermediate step between compilation and code generation. There are several reasons for introducing an intermediate graphical form.

- **Visual display:** A graphical representation provides a better visual display of the generated procedure, for example, to show where the compiler fails when the logic is inadequately refined for compilation, to show the order of computation, and to show the origin and transmission of preconditions through the procedure.
- **Advanced compilation:** The compilation algorithm presented in the previous chapter fails when memory deallocation needs to be pushed into procedures that consume memory. This chapter presents an algorithm to deal with this problem. This algorithm is most easily presented in terms of graphs.
- **Support for optimization:** Compiler optimization techniques typically deal with programs expressed as graphs that explicitly represent control and data flow [18]. The introduction of a graphical intermediate representation paves the way for further program analysis and optimization.

The exposition below shows how a computation graph is derived from a formula in the logic, and it assigns a computation graph a semantic map function f as described in Section 4.5. It also describes how a computation graph can be augmented with *control information* that represents the choices made during compilation. There are two important properties of computation graphs, for their service as an intermediate representation:

(1) The computation graph and the formula from which it derives have the same semantic map function.

(2) Computation graphs, when augmented with control information, are equivalent to procedure compositions.

www.manaraa.com

These properties, and the different processes involving the representations, are shown in Figure 1, below.

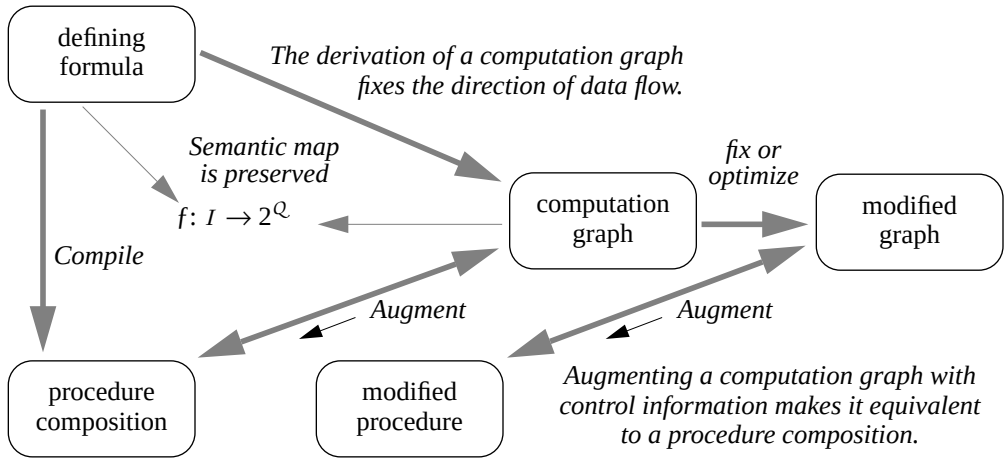


Figure 1: The relationships between the representations.

The derivation of a computation graph from a formula fixes the direction of data flow. (Rewriting the order of conjunctions and disjunctions, as described in Section 5.3, will change the computation graph.) The control information describes which procedure compositions are used when the specifying formula permits a choice.

6.1 Introduction to Computation Graphs

A computation graph is a directed acyclic graph[1] with annotated[2] nodes. A computation graph is constructed according to a set of composition rules

[1] As usual, a *directed graph* $\mathcal{G}$ is a set V of *nodes* (vertices) and a set E of *arcs* (edges). Each arc e∈ E is an ordered pair <m, n> in V×V. The arc e is an *outgoing arc* for m, which is the *tail* of e, and an *incoming arc* for n, which is the *head* of e. Arcs are said to leave, enter, or impinge on nodes, and nodes to carry arcs. If <m, n> is an arc, the node n is said to *succee*d the node m. A node is a *source* if it is the successor of no other node, and a *sink* if no other node succeeds it. Sources and sinks are *terminal nodes*; all others are *internal nodes*. Let R be the transitive closure of the successor relationship, i.e., <m, p> is in R if <m, p> is an arc, or if <m, n> is an arc and <n, p> is in R. R is called the *reachability relation* and a node p is said to be reachable from m precisely when <m, p> is in R. A directed graph is *acyclic* if no node is reachable from itself.

www.manaraa.com

described later. For now, only one implication of this construction is important: every computation graph has one source node and one success node. The nodes used in computation graphs are shown in Figure 2, below.

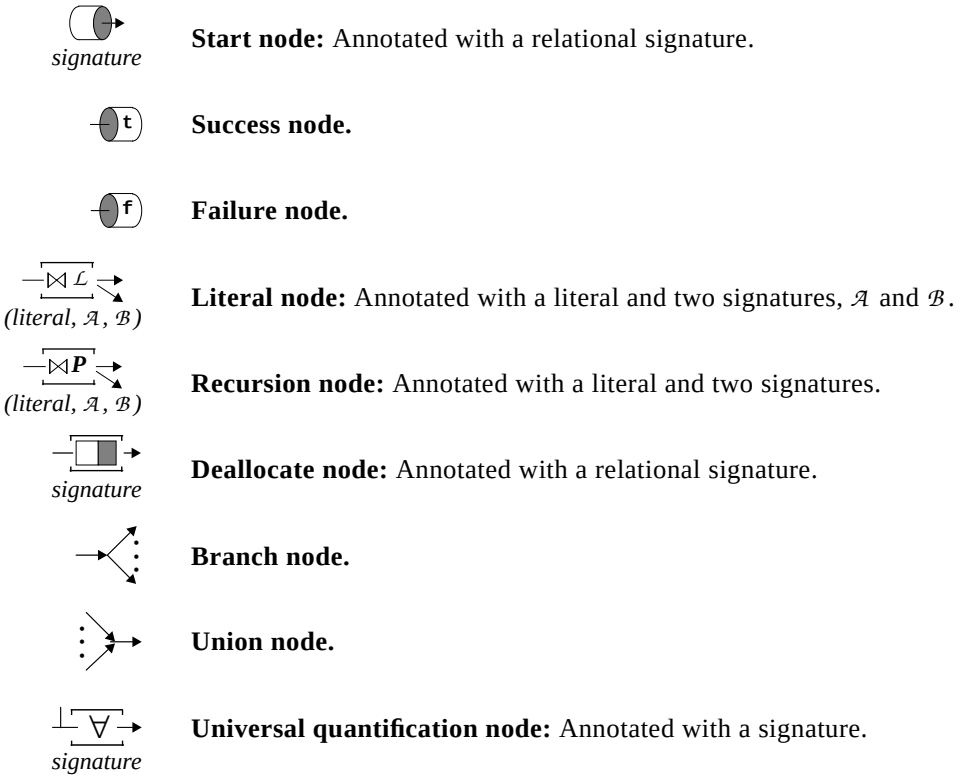


Figure 2: The basic nodes.

Each node type has a fixed number of incoming and outgoing arcs, as indicated by its icon, except for the branch node, which has one incoming arc and two or more outgoing arcs, and the union node, which has two or more incoming arcs, and one outgoing arc. The literal node and recursion node each have two outgoing arcs: a *success arc*, on top, and a *failure arc*, beneath. The universal quantification node has two incoming arcs, the *original arc* and the *test arc*.

Semantically, each internal node, except the union node and universal

---

[2] Formally, a *node annotation* is a function f from the set of nodes V into some set, f(V). Similarly, an *arc annotation* is a function g from the set of arcs E into some set, g(E). Annotations are usually viewed as labels, icons, or decorations attached to nodes and arcs.

www.manaraa.com

quantification node, denotes a function, f: $\mathcal{A} \rightarrow 2^{\mathcal{B}}$ that transforms each input tuple into a result relation, where $\mathcal{A}$ and $\mathcal{B}$ are the signatures of the input tuple and result relation, respectively. f can be viewed as a function that maps a relation over $\mathcal{A}$ into a relation over $\mathcal{B}$, by applying it to one tuple at a time and taking the union of the result. Formally, define the relation function $f' : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{B}}$ by:

$$t \in f'(R) \Leftrightarrow (\exists x)(x \in R \wedge t \in f(x))$$

One can view each input relation as being presented to the node along its incoming arc, and the node as presenting a result relation along each of its outgoing arcs. The sole role of the start node is to pass an *initial tuple* on to its outgoing arc. (This tuple has the signature I that is indicated by the start node.) By chaining the relational functions of the internal nodes, each arc e in the graph is assigned a function f_e: $I \rightarrow 2^{\mathcal{B}}$ that transforms the initial tuple into a relation. (See Figure 3.) If e is the incoming arc to the success node, then f_e is the denotation assigned to the computation graph as a whole, written $f_{\mathcal{G}}$.

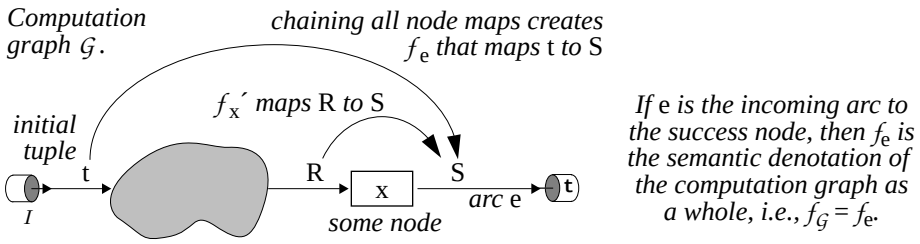


Figure 3: Denotational semantics for the computation graph.

The terminal nodes can be viewed as an interface for the computation graph as a whole, where the start node accepts an initial tuple, and the success node returns a result relation. The relation returned by the success node provides the graph's semantic denotation as a function that maps a tuple to a relation. (It will turn out that the failure node returns the initial tuple precisely when the success relation is empty.)

The function f_e defines a relation, and hence a signature, for each arc e. The signature of an arc can be viewed as indicating what variables are alive at that portion of the graph. This signature is called the *variable scope* at the arc. The deallocate node has as its sole function the removal of variables from the current

www.manaraa.com

scope. There is no corresponding allocate node. Instead, variables are introduced by literal or recursion nodes. In the next chapter, the notion of variable scope will be carried forward into a more detailed look at memory management.

To complete this account of denotation for computation graphs, it suffices to describe what function is assigned to each internal node. The relation S produced by an internal node is described below in terms of (1) the node's annotations, and (2) the node's input relation, R. For literal and recursion nodes, S_s is the relation presented to the success arc, and S_f is the relation presented to the failure arc. For the universal quantification node, R_o is the relation passed along its original arc, and R_t is the relation passed along its test arc.

Branch node: S = R. This node's output relation is a copy of its input relation. Its output relation is placed on all of its outgoing arcs.

Union node: $S = \cup_i R_i$. This node's output relation is the union of its input relations. It is the only node that has multiple input relations.

Universal quantification node: $(\forall x \in \mathcal{X})(\sigma_{\mathcal{X}=x}S = (\sigma_{\mathcal{X}=x}R_o)/(\sigma_{\mathcal{X}=x}R_t))$. Like the union node, this node performs a relational operation. First, divide the input relations into partitions each of which have a common $\mathcal{X}$ value, where $\mathcal{X}$ is the signature of the node. In each partition, the result is the original relation divided by the test relation. In other words, for each $\mathcal{X}$ value, S contains all of the tuples in R_o if R_t also contains these tuples, otherwise, S contains none of these tuples.

Deallocate node: $S = R\backslash\mathcal{A}$. This node is annotated with a relational signature $\mathcal{A}$ that indicates what columns to remove from its input relation R.

Literal node: $S_s = \pi_{\mathcal{B}}(R \bowtie [l])$, $S_f = R - \pi_{\tau(R)}(S_s)$. The node is annotated with a literal l in the logic. The predicate symbol of l is either part of the base logic or its predicate is realized by a previously compiled program. The node is also annotated with two signatures, the *input arguments*, $\mathcal{A}$, and the *returned arguments*, $\mathcal{B}$. Their union must equal the variables of l. The denotation of this node is the function f_l: $\delta\mathcal{A} \rightarrow 2^{\mathcal{B}}$ where $b \in f_l(a) \Leftrightarrow a \bowtie \beta \in [l]$. ($[l]$ is the denotation of l. This construction is described more fully in Chapter 5.)

Recursion node: $S_s = R \bowtie LFP(\mathcal{G})$, $S_f = R - \pi_{\tau(R)}(S_s)$. The only difference between the literal node and the recursion node is that the predicate for the latter is **p**, the predicate being defined by the formula. In terms of computation graphs, this node can be viewed as a recursive instantiation of the computation graph $\mathcal{G}$. That this leads to a least fixed-point, and that it mirrors the semantics of the logic, will be proved at the end of Section 6.2, below.

www.manaraa.com

As mentioned earlier, most nodes can be understood in terms of what result tuples they produce from each input tuple on their incoming arc. The union and universal quantification nodes must be understood as operating directly on relations, the former because it has several incoming nodes, and the latter because it performs a relational divide, which cannot be decomposed into tuple operations.

6.2 Constructing Graphs from a Program Specification

This section describes the constructions that generate computation graphs from a program specification, ***P***. The relevant parts of ***P*** are the formula $\Phi_{\mathbf{p}}$ that defines the new predicate **p** and the input signature and result signatures, I and Q. Each construction rule applies to a formula of the logic. Each construction generates a computation graph from the nodes described in Section 6.1, possibly combined with computation graphs that were constructed for syntactic pieces of the formula. Each construction yields a directed acyclic graph with one start node, one success node, and one failure node. Most important, each construction preserves the following semantic invariants.

Semantic Invariants for Computation Graph Construction:

(1) **Meaning of failure node:** If a computation graph $\mathcal{G}$ denotes the function f, then the incoming arc e to failure node denotes the function g, where g(t)=t if f(t) is empty, and g(t) is empty if f(t) has tuples. (For an input relation R, this means that g'(R) contains precisely those tuples from R that generate no tuples in f'(R).)

(2) **Preservation of semantic denotation:** If a computation graph $\mathcal{G}$ is constructed from a program specification ***P***, whose defining formula is $\Phi_{\mathbf{p}}$, with input signature I and result signature Q, then its semantic $f_{\mathcal{G}}$ is the same as the semantic map $f_{\boldsymbol{P}}$ for the program specification. (See Section 4.5.)

The base construction, if the formula $\Phi_{\mathbf{p}}$ is a non-recursive literal l, generates the computation graph shown in Figure 4. I and Q are the input variables and result variables specified in ***P***. This construction trivially satisfies the semantic invariants.

www.manaraa.com

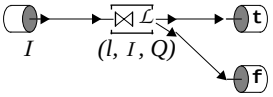


Figure 4: Construction for a literal.

The construction for positive, recursive literals is identical to the one for other literals, above, except that a recursion node is used instead of a literal node. It is a stickier issue to show that it satisfies the semantic invariant. This is addressed after the other constructions are explained.

Graphs are composed by removing terminal nodes and connecting their arcs elsewhere. For example, the upper part of Figure 5 shows unspecified computation graphs for formulae θ and ϕ, each drawn as a shaded body with arcs connected to terminal nodes. The lower part of Figure 5 shows these are combined to form a larger computation graph, for $\theta \wedge \phi$.

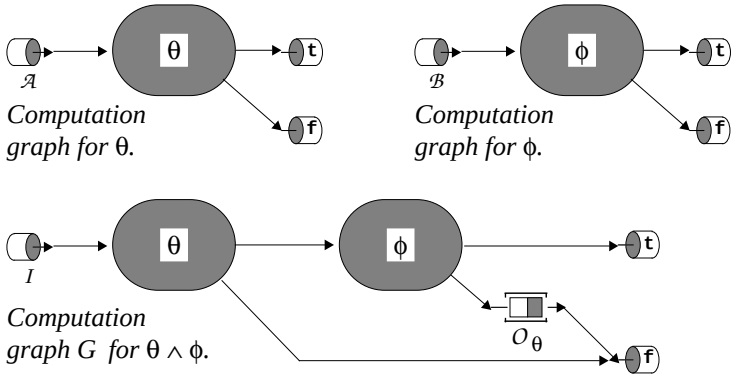


Figure 5: Computation graph for logical conjunction.

The computation graph shown in Figure 5 for conjunction trivially satisfies the first semantic invariant, but can fail the second if the input and result sets for the two component graphs do not synchronize correctly. For example, if $\mathcal{A}=\{x\}$ and $Q_\theta=\{y\}$ (meaning that x is absent from θ's output relation), and $\mathcal{B}=\{y\}$ and $Q_\phi=\{x\}$, then the graph is not even syntactically correct, because the incoming arcs to the union node have different signatures ({} for the upper arc, and {x} for the lower). Moreover, the second invariant cannot be satisfied, because y must be a column in either the domain or range of $f_{\theta\wedge\phi}$, but it is in neither for $f_{\mathcal{G}}$.

www.manaraa.com

The semantic correctness of this first construction, and of the remaining ones, depends on certain syntax rules being met. Some of the syntax rules pertain to the individual constructions. (Indeed, each graphical construction is a kind of syntax rule.) There are a few syntax rules that apply to all of the constructions. These are worth stating now. Notice that the descriptions of the nodes in Section 6.1 show how to calculate a signature for each node's outgoing arcs, given signatures for its incoming arcs. Because a computation graph is acyclic, it is trivial to run this calculation from the start node to the success and failure nodes. Each rule below constrains the nature of a node, sometimes in terms of the signatures on its incoming arcs, or the rule asserts some syntactic relationship between a graph and the program specification from which it is derived.

General Syntax Rules for Computation Graph Construction

(1) Recursive literals must be positive: Recursive literals may not be negated, nor may they appear in the conditional part of an if–then–else, nor in the conditional part of a universal quantification construct. (These constructs are described below.)

(2) Input signatures to the union node: Input relations to union nodes must have the same signature.

(3) Input signature to the deallocate node: The signature of the input relation to the deallocate node must contain the node's signature. (These first three rules were adumbrated above.)

(4) **Correct start, success, and failure node signatures:** The signature calculated for the arc leading to the success node must be Q_P. The annotation on the start node and the signature calculated for the arc leading to the failure node must be I_P.

The remaining rules apply to the literal and recursion nodes. Recall that these nodes are annotated with a literal, l, and two signatures, the *input arguments*, $\mathcal{A}$, and the *result arguments*, $\mathcal{B}$.

(5) **Consistency of annotations:** The union of $\mathcal{A}$ and $\mathcal{B}$ must equal V(l), the variables in the literal, l. Each variable in $\mathcal{B} - \mathcal{A}$ (the output arguments) and $\mathcal{A} - \mathcal{B}$ (the destructively consumed input arguments) must appear exactly once as an actual argument of l. (The input arguments, variables in $\mathcal{A} \cap \mathcal{B}$, can appear multiple times, and can appear embedded in terms that are arguments to l.)

(6) **Adequate input:** The signature of the incoming arc must contain $\mathcal{A}$.

www.manaraa.com

(7) **Write once:** The signature of the incoming arc must not intersect $\mathcal{B}-\mathcal{A}$. ($\mathcal{B}-\mathcal{A}$ is the output set for the node.)

These rules winnow the set of computation graphs that would otherwise be constructed from a program specification. In essence, they insist on a consistent order of computation, so that variables are not assigned multiple values and are assigned values before they are used as inputs. Given these rules, the construction shown in Figure 5 easily satisfies the semantic invariants, since θ generates values for some columns that are joined to the initial tuple t, and ϕ joins new values to the result. Both θ and ϕ may take away columns, but only from the original input tuple. Thus, the output is some projection of $t \bowtie [\theta] \bowtie [\phi]$, and so the computation graph preserves the denotation of the formula.

The construction for disjunction is very similar to that for conjunction. Instead of the two component graphs being chained along success arcs, leading to the success node, they are chained along failure arcs, leading to the failure node. This construction is shown in Figure 6, below. For similar reasons, it maintains the semantic invariants if the general syntax rules are met.

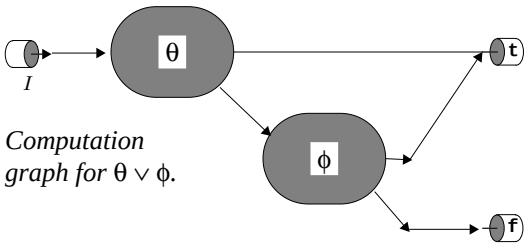


Figure 6: Construction for disjunction.

The construction for if–then–else, shown in Figure 7 below, combines elements of both previous constructions, as one would expect, since $\theta \Rightarrow \phi \nabla \varphi$ is equivalent to $(\theta \wedge \phi) \vee (\sim\theta \wedge \varphi)$. In fact, this construction subsumes both conjunction (by setting φ=**F**) and disjunction (by setting ϕ=**T**). For similar reasons, it satisfies the semantic invariants.

www.manaraa.com

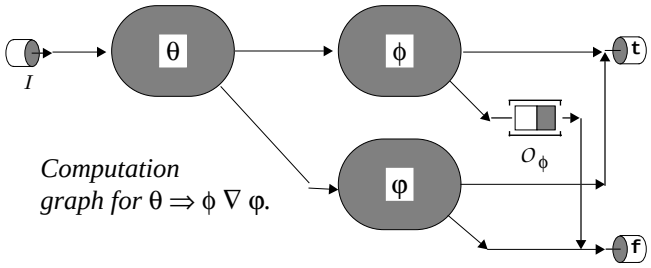


Figure 7: Construction for if–then–else.

An existentially quantified formula is implemented by a computation graph that produces a result tuple that satisfies the inner formula, and that then "forgets" the quantified variable. This is analogous to function skolemization. The computation graph is shown in Figure 8 below. By swapping the success and failure nodes, this construction realizes $\sim(\exists X)\theta$. This latter requires that X have as output variables precisely those in X.

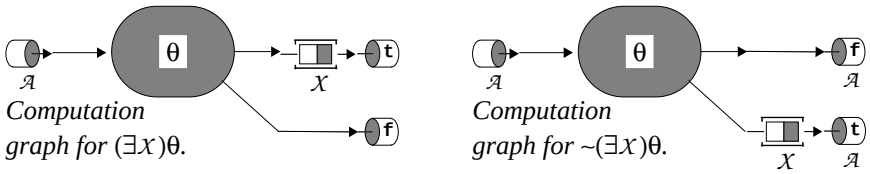


Figure 8: Construction for existential quantification.

Universal quantification is also treated constructively. Only an if–then formula can be universally quantified. The conditional component generates values for the quantified variables, and the consequent must hold for all of these values. The universal quantification node is used to "turn off" the results from the consequent if the consequent fails for any of the values generated by the conditional. The construction for universal quantification is shown in Figure 9 below.

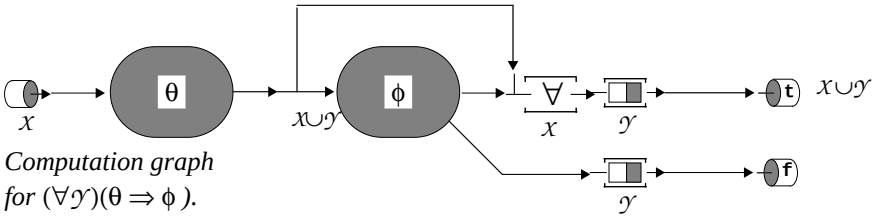


Figure 9: Construction for universal quantification.

www.manaraa.com

There are three sets of variables involved in the construction, identified by the signatures $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$. $\mathcal{X}$ is the signature of the initial tuple. The conditional component, θ, of the if–then–else generates values for $\mathcal{Y}$. The consequent, ϕ, generates $\mathcal{Z}$ tuples for each $\mathcal{X}\mathcal{Y}$ tuple. For each input tuple, the universal quantification node "turns off" the output to the success node except for $\mathcal{Z}$ tuples that are generated for all $\mathcal{Y}$ tuples. For any initial tuple, if there are no qualifying $\mathcal{Z}$ tuples, the initial tuple takes the failure arc out of the consequent. Thus, the meaning of the failure node and the preservation of logical semantics are preserved for the success path out of the conditional.

The alternative, φ, generates $\mathcal{Z}$ tuples when there is no $\mathcal{Y}$ tuple that satisfies the conditional for the initial $\mathcal{X}$ tuple. This is usually used to generate default values for the $\mathcal{Z}$ variables. It trivially satisfies the semantic invariants.

Recursion

The discussion of the above constructions, excluding recursion, can be summarized in a lemma.

Lm. 10 The syntax rules and the non-recursive constructions yield computation graphs that preserve the semantic invariants. In particular, any computation graph so constructed has the same semantic map as the program specification from which it is derived.

When the nodes were explained, the precise semantics of the recursion node was finessed, with reference to a later explanation. It is now time to give a more precise definition of the semantics of the recursion node, and to show that it preserves the semantic invariants. As was done for recursively defined predicates in Section 4.3, a least fixed-point semantic is defined below for computation graphs.

Given a recursion node r annotated with input variables $\mathcal{A}$, output variables $\mathcal{B}$, and literal l (whose predicate is **p**), define f_r^0: $\mathcal{A} \rightarrow 2^{\mathcal{B}}$ by $f_r^0(a)=\{\}$. Notice that in building up the least fixed-point for recursively defined predicates, $[\phi_{\mathbf{p}}]\uparrow 0$ is defined to be empty. Define f_r^{i+1} to be the denotation of any computation

www.manaraa.com

graph for ***P***, when recursive nodes in it are assigned the denotation f_r^i. (If the construction rules do not provide a computation graph for ***P***, then there is no point in assigning a denotation to its recursion node.) The important claim is that this series of denotations for the computation graph tracks the series whose limit is ***P***'s denotation.

Lm. 11 $b \in f_r^i(a) \Leftrightarrow a \cdot b \in [\phi_{\mathbf{p}}]\uparrow i$

Proof That this is true for i=0 provides the basis for an induction. Since the non-recursive nodes preserve semantic meaning and because $[\phi_{\mathbf{p}}]\uparrow(i+1)$ is defined as one non-recursive application of **p**, the inductive step holds.

Since f_r^i tracks $[\phi_{\mathbf{p}}]\uparrow i$, it is also a monotonic sequence, and has the same least fixed-point.[3] This least fixed-point is the denotation assigned to the recursion node. Because it is a fixed-point, it has the same denotation as any of ***P***'s computation graphs, and this is the denotation of ***P***. This proves the desired generalization of Lm. 10:

Thm. 12 If the computation graph $\mathcal{G}$ is constructed from ***P*** applying the construction and syntax rules, then $f_{\mathcal{G}} = f_{\mathbf{P}}$, i.e., the upper triangle in Figure 1 commutes.

6.3 Augmenting Computation Graphs with Control Information

The graph constructions described in the previous section follow from the logical construction of the specifying formula and, with two exceptions, provide exactly the same information as the corresponding procedure composition. These exceptions are non-recursive literal nodes and the construction for disjunction. The former construction lacks any information about what procedure is used to realize the literal, and what its semantic description is. The latter

[3] It is easy to prove that a computation graph has a least fixed-point denotation, without referring to ***P***'s least fixed-point, but their equality would then have to be proved separately.

www.manaraa.com

construction corresponds to *either* the choice *or* the concatenate procedure operations.

Note that each disjunction corresponds to a unique union node in the computation graph. An *augmented computation graph* adds additional annotations to union nodes and literal nodes. It adds to each union node (1) an indicator of whether the choice or concatenate operator is desired, and to each literal node (2) the name of the procedure chosen to realize the literal, and (3) the procedure's semantic description. For each literal node, the chosen procedure must have an input signature and result signature that matches those of the literal node.

There is an exact correspondence between procedure compositions and augmented computation graphs. Given an augmented computation graph, the procedure composition is obtained by deconstructing the graph, and applying the corresponding procedure composition for each graph construction. At disjunctions, where the only choice of procedure compositions obtains, the annotation of the union node tells which to make. At literal nodes, their annotation tells which named procedure belongs at that place in the procedure composition. The reverse process generates a computation graph from a procedure composition.

This suggests a compilation algorithm that is an alternative to the one presented in Section 5.3. It is sketched below, at a very high level:

```
1: construct ComputationGraph;
2: find ControlInformation to satisfy ProgramSpecif  ication;
3: perform Fixups & Optimizations;
4: convert ComputationGraph to ProcedureComposition.
```

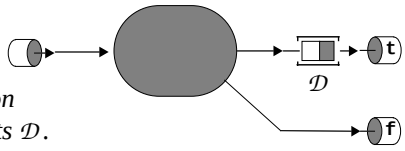
The next section describes an algorithm that modifies the computation graph for memory management. Its role is in step 3 of the above algorithm.

6.4 Memory Management

The compilation algorithm described in Section 5.3 will fail on programs that have consumed input arguments. It make no provision for

www.manaraa.com

deallocating these variables on the successful termination of the composed procedure. This is remedied below, through adjustments to the computation graph. To adjust a computation graph for the desired result signature, a deallocation node is inserted before the success node that discards the consumed input variables. This is shown in Figure 13, below.



Augmenting a computation graph for consumed inputs $\mathcal{D}$.

Figure 13: Construction for consumed input arguments.

This does not yet solve the problem. The deallocation node requires a procedure that deallocates the variable. (In terms of the procedural calculus, a deallocate node is equivalent to a projection procedure. See Section 5.2.) Several related problems arise.

- The compiler contains built in procedures to deallocate variables whose sort does not contain pointer references or sets. (For obvious reasons, the compiler cannot include procedures to deallocate non-contiguous data structures.)
- There may be a literal in the program's body that is only realized by a procedure that consumes an input variable. (This may be where the programmer intended the deallocation of a non-contiguous data structure to occur.) The deallocation node placed at the end of the composition graph in Figure 13 somehow must be percolated back to the literal that needs to deallocate its input variable.
- When there is a choice between a procedure that consumes an input invariable and one that does not, if the variable must be deallocated anyway, then it is more efficient to do the deallocation "deeper" in the program by choosing the procedure that consumes its input variable.

One literal that often may and sometimes must consume one of its input variables is data structure isomorphism. The compiler realizes data structure isomorphism (1) by copying a data structure, or (2) by destructively updating a data structure. The latter is clearly more efficient. The former is possible only when the data structure is contiguous or when the programmer has written a copy constructor for the data structure of concern.

www.manaraa.com

If a literal node appears immediately prior to a deallocate node, then deallocated variables that are pure inputs to the literal can be removed from the deallocate node and made consumed inputs on the literal node. This is shown in Figure 14, below. Deallocation is pushed into the procedure that realizes the literal. (In particular, this would turn data structure isomorphism from a copy into a destructive update.) Obviously, this should be done only if the procedure library has a procedure that consumes the concerned variable. Any deallocate node that is left with an empty signature can be moved from the graph.

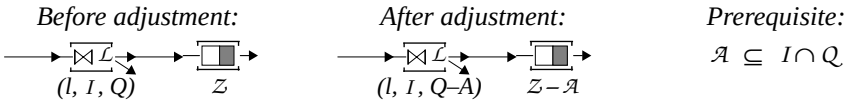


Figure 14: Creating literals with consumed input arguments.

To make effective use of procedures that consume their inputs, deallocate nodes, especially the one placed at the end of the computation graph as shown in Figure 13, should be pushed as far toward the front of the graph as possible. In doing this, certain constraints must be followed. First, a deallocation node cannot be pushed past a node that makes use of its variable. Failing to meet this constraint would commit the programming mistake of using a variable after it was deallocated! Once a deallocation node is pushed forward to a previous node that uses the same variable, it may be at the point where it can be eliminated by realizing the previous node through a procedure that consumes the variable, as shown in Figure 14. Second, if a deallocate node is pushed forward of a union node, it must be replicated along all incoming paths. Failure to meet this constraint would commit the programming mistake of deallocating the variable along some paths of execution, but not along others. Third, to be pushed past a node with multiple output arcs, the deallocation node must be present on all those output arcs. (This is the converse of the previous constraint.)

The algorithm below pushes variable deallocation as far forward as possible. This maximizes the amount of destructive update that is possible. The algorithm assumes that each deallocation node concerns exactly one variable. (If a deallocation node has several variables, it can be turned into a sequence of deallocation nodes, each with only one variable.) The algorithm marks

www.manaraa.com

deallocation nodes as they are processed. Each node's mark is either **Ready**, meaning that it can be pushed, **Wait**, meaning that it has been pushed as far as possible until other deallocation nodes are pushed, and **Done**, meaning that the node cannot be pushed any further.

```
OptimizeDeallocation () { // The graph is a global variable
   node DeallNode;
   MarkAll (Ready);
   while (ChooseReadyDeallocNode (DeallNode))
      Push (DeallNode);
}

// The function below pushes a single node forward

Push (DeallNode) {
   varName x = Signature (DeallNode);
   node CNode = PreviousNode (DeallNode);
   if (VarUsedInNode (CNode, x))
      Mark (DeallNode, Done);
   else if (NbrOutArcs (CNode) > 1) {
      if (all nodes on out arcs are same deallocate) {
         RemoveFromGraph (all nodes on out arcs);
         Insert (DeallNode, InArc (CNode));
      }
      else Mark (DeallNode, Wait);
   else if (NodeType(CNode) == Union) {
      RemoveFromGraph (DeallNode);
      for (each Arc leading to CNode) {
         new node NewNode;
         NewNode = Insert (DeallNode, Arc);
         Push (NewNode);
         delete (DeallNode);
      }
   else {
      RemoveFromGraph (DeallNode);
      Insert (DeallNode, InArc(CNode));
      CNode = PrevNode (CNode);
```

www.manaraa.com

```
    }
}
```

The first **`else if`** merges a set of deallocation nodes (for the same variable) that occupy *all* the outgoing arcs of a node. This latter node has multiple outgoing arcs and, because of this, it blocks the forward progress of any lone deallocation node that is pushed forward to one of its outgoing arcs. The deallocation nodes that are pushed forward to the blocking node, prior to the one that reaches the last outgoing arc, must wait for this merger. The second **`else if`** replicates a deallocate node on all the incoming arcs to a union node.

6.5 Reasoning on Computation Graphs

Reasoning about Galois programs as a whole is easy. If a program successfully executes, then the values bound to its arguments satisfy the formula that the program realizes, i.e., the postcondition in its specification. If a Galois program fails, then there are no result values that, together with the input values, satisfy the postcondition.

The problem is that it is sometimes necessary to reason about *part* of a Galois program. The computation graph intuitively displays the order of computation in a program. It incrementally makes true parts of the postcondition, until the postcondition is either satisfied as a whole, or the computation cannot proceed. The notion of what "is made true by the computation graph up to a certain point" is formalized below. This helps programmers to understand how prerequisite assertions are constructed, and how to modify predicate definitions to achieve desired computations. We assume, in the exposition below, that deallocate nodes for consumed input variables have not yet been propagated forward.

The Arc Invariant

Section 6.1 defined a relational map, f_e, for each arc e in a computation graph $\mathcal{G}$. For each input tuple i to the computation graph, f_e(i) is a relation assigned to the arc e. If e is incoming arc to the success node, then f_e is the

www.manaraa.com

semantic map assigned to the graph as a whole, $f_{\mathcal{G}}$. Thm. 12 states that $f_{\mathcal{G}}$ is the same as the semantic map for the program specification from which $\mathcal{G}$ derives.

No special meaning was assigned to f_e for arcs other than the one that leads to the success node; they were merely intermediate steps in the calculation. The task now is to define a logical formula Φ for each arc e, such that $f_{\Phi}=f_e$. Φ is a logical description of the arc's relational map, and hence, describes what can be assumed of the input tuples to its target node. In short, this formula describes "what has been made true" up to that point in the computation graph. It is called the arc's *invariant*.

As a starting point, if e is the arc leading from the start node, then define Φ_e=`True`. As with the definition of f_e, Φ_e is calculated in a depth-first fashion along the computation graph. At any node, let e be an outgoing arc and assume that Φ_d is available for each of the incoming arcs, d. The following rules determine Φ_e (the invariant for the outgoing arc).

Branch node: $\Phi_e = \Phi_d$. The invariant on each outgoing arc is identical to that on the incoming arc.

Union node: $\Phi_e = \vee_d \Phi_d$. The invariant on the outgoing arc is the disjunction of the invariants on the incoming arcs.

Universal quantification node: $\Phi_e = (\forall \mathcal{Y})\Phi_d$. The invariant on the outgoing arc is the universal quantification of the invariant on the incoming arc, over the quantified variables that annotate the node.

Deallocate node: $\Phi_e = (\exists \mathcal{Y})\Phi_d$. The invariant on the outgoing arc is the existential quantification of the invariant on the incoming arc, over the quantified variables that annotate the node.

Literal and recursion nodes: $\Phi_e = \Phi_d \wedge \mathcal{L}$ and $\Phi_{e'} = \Phi_d \wedge (\forall \mathcal{Z})(\sim\mathcal{L})$. The invariant on the success arc is the conjunction of the invariant on the incoming arc with the literal that annotates the node. The invariant on the failure arc is the conjunction of the invariant on the incoming arc with the negation of the literal that annotates the node, universally quantified over the literal's output variables.

The logical compositions above are precisely those for which the corresponding relational compositions in Section 6.2 provide a model. There is no need here to make special provisions for recursion, since the model of recursively

www.manaraa.com

defined formula is already well defined. Relying on Thm. 12, the following holds.

Lm. 15 If the computation graph $\mathcal{G}$ is constructed from a program specification ***P*** applying the construction and syntax rules given in the previous sections, then for each arc e, $f_\Phi = f_e$.

Note that the signature of e is precisely the set of free variables of Φ_e. An assertion is said to hold at a node with unique incoming arc e if the assertion can be derived from the logical axioms, the precondition in the program specification , and the arc invariant Φ_e. Any such assertion is satisfied by any of the tuples in $f_e(i)$, for any initial tuple i that satisfies the precondition from the program specification .

The Preconditions for a Literal Node

A procedure chosen to realize a literal, whether a previously compiled Galois program or primitive procedure built into the compiler, may require a certain precondition to hold on its input tuples. It suffices for the precondition to hold at the concerned literal node. (See Figure 16.)

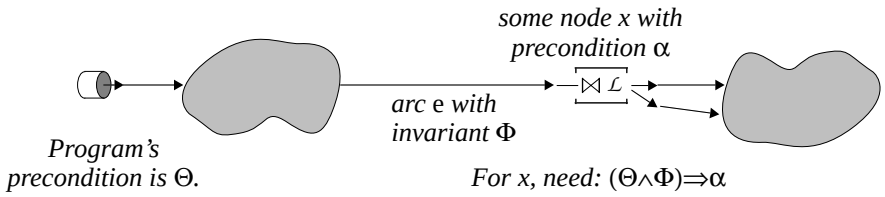


Figure 16: The effects of a literal node's precondition.

In short, to verify the concerned literal node with precondition α, the programmer must prove $(\Theta \wedge \Phi) \Rightarrow \alpha$, where Φ is what the computation graph has made true up until the concerned node, and Θ is the precondition from the program's specification. The conjunction of all assertions of this form for all nodes in the computation graph is precisely the *prerequisite assertion* calculated in Chapter 5. Being able to see these assertions individually, at the concerned locations in the graph, helps the programmer understand from whence they derive and how changes in the program specification or predicate's definition will affect the prerequisite assertion.

www.manaraa.com

7. Producing C Code

Procedural semantics can be discussed precisely and fully without reference to a particular implementation. The compilation algorithm in Chapter 5 generates a program in the form of a procedure composition in the procedural calculus. Chapter 6 describes an equivalent graphical representation, the computation graph. Both representations of a program fully describe its behavior.

Compilers are often divided into a front-end that turns input in the source language into an intermediate form, and a back-end that produces code in the target language from this intermediate form. This latter step is often the easier of the two. The Galois compiler puts more distance than usual between the front-end and back-end, because verifying the prerequisite assertion (and possibly rewriting the postcondition) are necessary steps between front-end compilation and back-end code generation.

The major parameters for a back-end are what language to target, what calling conventions to use, and how to represent data in the target language. To illustrate the conversion of logic into code, this chapter describes code production in C. The back-end described here is meant to serve as an example of code production and provides the context for the examples. The C code is not highly optimized.

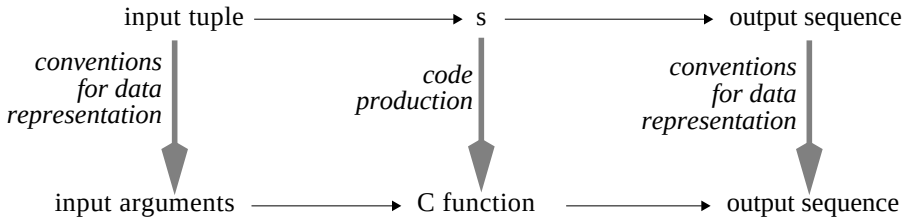


Figure 1: The role of code production.

The code produced by the back-end for any procedure s must have a concrete behavior that mirrors the abstract behavior defined for s. In other words, the back-end described in this chapter generates from each procedure s a C

www.manaraa.com

function that causes the diagram in Figure 1 to commute.

The target language is C as described by the second edition of the classic text by Kernighan and Ritchie [29]. It makes use of several of the "new-style" conventions. (Refer to Appendix C of [29].) In particular, the type of function arguments are declared directly in the argument list of the function header, and generic pointers are given the type **`void*`**. One exception is made: comments are written in the style of C++, in order to avoid clutter in the presentation.

7.1 Skeleton Code for Programs

The narrative below describes how code is produced for a logic program "from the outside-in." That is, given a procedure composition, in the form of an expression in the procedural calculus or of a computation graph, code production begins with the outermost composition, it generates skeletal code for this composition, remembering where unfinished "holes" corresponding to components of the procedure composition remain, and it then repeats this process on the components, filling in the "holes." The outermost shell is the skeleton for the C function that implements the logic program:

```
int ProgName ( argument list )
{
   definition of G_success;
   definition of G_stop;
   // .. generated code for ProgName
}
```

The skeleton above demonstrates several conventions that are followed in the following presentation. First, in presenting code production, we use the C++ convention for comments, i.e., everything from "`//`" to the end of the line is a comment. Second, a line that contains only a comment that begins with an ellipsis (`..`) denotes a "hole" to be replaced by further generated code. The comment can be viewed as a placeholder that indicates what code will replace it. For example, in the skeleton above, the comment "**`// .. generated code for ProgName`**" will be

www.manaraa.com

replaced by the code generated for the procedure composition that implements the specified program. Third, italics are used to indicate code that varies according to some rule that will be explained in the text. For example, the definition of **G_success** and **G_stop** are shown in more detail later, when two more specific versions of the function skeleton are explained. Fourth, variables that are artifacts of code production — as opposed to representing logical variables — are prefixed by **G_**. These variables are summarized at the end of Section 7.2.

Every program contains the variable **G_success**, which is initialized to true in C's fashion of dealing with booleans — i.e., to 1 — and the variable **G_stop**, which is initialized to false — i.e., to 0. The variable **G_success** can be interpreted roughly as answering the question: it still possible for the program to generate another result tuple that satisfied the program specification? The variable **G_stop** can be interpreted roughly as answering the question: should execution break out of the current generator? The use of these variables will be explained further as this chapter proceeds.

Data Representation

Values in the basic sorts are represented by C data types as described in the table below.

Galois sort	C type
integer & **int**	**int**
address & **addr**	**void***
character & **char**	**char**
floatingPt & **float**	**float**
***sort*[]**	***type*[]**
sort{}	objects of the corresponding type allocated in the heap
struct	same **struct** without set fields

www.manaraa.com

In C, there is no need to distinguish between the base sorts and their corresponding addressed sorts. Thus, values in both the **integer** sort and the **int** sort are represented in C by an **int**. The correspondence between the logic's sorts and C's types has been made straightforward.

Variables in the code are either *direct* or *indirect*. All destructively consumed input arguments, output arguments, and existentially quantified variables of an addressed sort are indirect, and are declared with one level of indirection beyond that in the above table. In other words, their declarations are **type*** rather than just **type** as shown. When an indirect variable is passed to a direct argument, it is dereferenced in the procedure invocation. Direct variables are never passed to indirect arguments.

Two Kinds of Programs

For the purpose of code production, programs are categorized as generators or non-generators. Generators are programs that potentially produce more than one output tuple for each input tuple. Hence, generators are complete programs, excluding functions. (Complete programs produce all output tuples that satisfy their logical formula. A function is complete, but is known to have at most one output tuple for each input tuple.) Non-generators produce at most one output tuple for each input tuple. These are effective programs and function programs.

The code produced for a program takes one of two forms, depending on whether or not the program is a generator. A generator has an initial, integer argument, **G_first**, that, if non-zero, indicates that the program should generate the first output tuple for the passed input tuple. (See Section 5.4 for a discussion of program execution.) If this argument is zero, the program generates the next output tuple in the sequence. This argument is passed by reference, and is always returned equal to zero. The skeleton of the C function that implements a generator program is shown below in detail:

```
int ProgName ( int* G_first, predicate arguments )
{
   static int G_success;
```

www.manaraa.com

```
   static int G_stop;
   if (!G_first) goto Inside;
   G_success=1;
   G_stop=0
   // .. first part of procedure composition for ProgName
      // Success continuation of ProgName:
      return G_success;
      // Output deallocation unnecessary (read below)
Inside:
   // .. remainder of procedure composition for ProgName
   return 0;
}
```

All programs are implemented as integer-valued C functions that return one when they generate an output tuple and that return zero when there is no qualified output tuple, or in the case of a generator, no further output tuples. As the above skeleton shows, a generator program retains state between invocations that determines its location in the output sequence. Note the difference in the declaration of internal variables between the skeleton above for a generator program and the skeleton below for a non-generator program.

```
int ProgName ( predicate arguments )
{
   int G_success=1;
   int G_stop=0;
   // .. body of ProgName
   return (G_success);
}
```

In a generator program, all variables (except the function's arguments) are declared **`static`**. Because of this, they are initialized through explicit initialization statements rather than in their definition. For simplicity, the remaining exposition of code production assumes that code is produced for a program that is *not* a generator. When similar fragments are produced for a program that is a generator, the only differences are (1) that variables are declared **`static`**, and (2) variables are initialized by an assignment statement immediately following their definition, rather than in the definition itself.

www.manaraa.com

The predicate arguments are the arguments in the logical definition of the predicate that the program realizes. These are declared with the types previously described. Consider a program whose specification is **`doIt (in x, inx y, out z)`**, that realizes some predicate whose arguments are **`x`**, an **`integer`**, **`y`**, a structure named **`foo`**, and **`z`**, a structure named **`bar`**. The code generated for a program that realizes this specification would have the following function header:

```
int doIt (int x, foo* y, bar* z)
```

Note that **`y`** and **`z`** are declared with one level of indirection, because **`y`** is a destructively consumed input argument, and **`z`** is an output argument.

7.2 Code for Procedure Compositions

Once the C function's skeleton is in place, code production proceeds for the procedure composition. Each procedure is turned into fragments of code that are fit into the framework of the code that has been previously generated. There are two general forms for the code that is produced for a procedure. If the procedure s is complete, it is implemented as two code fragments that surround a *success continuation*:

```
// .. first part of s
   // .. success continuation
   // .. deallocation of output variables of s
// .. remainder of s
```

The success continuation is a hole that is filled by appropriate other procedures. The code in the success continuation is executed repeatedly as the complete procedure iteratively produces qualified output tuples for the formula it realizes. The code in the success continuation can force an early break out of the iteration by setting **`G_stop`** to 1.

Each successful iteration through a successful procedure must rid itself of its output variables after the success continuation is executed. In the case where the success continuation is a return to a calling program, it is the calling program's responsibility to consume the output values. (This was noted in the code shown

www.manaraa.com

above for the skeleton of a generator program.) Note, also, that the variable allocation may have been pushed into the code in the success continuation, by the algorithm described in Section 6.4. Failing these two exigencies, the output variables of a complete procedure must be deallocated where shown above.

A procedure that is effective is implemented by one fragment of code:

```
// .. body of s
```

When execution exits this fragment of code, **G_success** is 1 if an output tuple has been produced, and 0 if there is no qualified output tuple. Both forms assume that **G_success** is set and **G_stop** is not set on entry.

Program Invocation

Atoms are realized either by invocation of C functions that were produced by compiling other programs, or by fragments of code[1] known to the compiler for implementing the primitive predicates. In either case, the code must fit into either the form defined for complete procedures or the form defined for effective procedures. The set of stock implementations of the primitive predicates is discussed later. A generator program (which is necessarily complete) is invoked as shown below.

```
// Complete use of a generator, ProgName
{
   int G_first = 1;
   int G_loop;
   // .. evaluate input arguments
   G_loop=ProgName (&G_first, actual arguments);
   while (G_loop) {
      // .. success continuation
      if (G_stop)
         G_loop=0;
      else
```

[1] In the prototype compiler, these fragments are embedded in macros, whose use in generated code looks very much like function invocation.

www.manaraa.com

```
            G_loop=ProgName (&G_first, actual arguments);
        }
    }
```

The comment "**.. success continuation**" marks the spot for code that is executed at the continuation of the procedure, as required by various compositions described below. The code above this comment replaces a comment of the form "**.. first part of Progname**". The code below comment replaces a comment of the form "**.. remainder of Progname**". Thus, this code fits the general form for complete procedures.

A fragment of code that evaluates input arguments is necessary if logical functions are employed in the terms of the atom that the compiler implements through code more complex than a C expression. (In this case, there may be more internal variables.) Input arguments need only be evaluated once, because the input tuple does not change as successive output tuples are generated.

The variable **G_first** is used to signal the first invocation of the program for the concerned input tuple. The variable **G_loop** is used to control the iteration through result tuples. In any one program, there may be many instances of these variables, which are distinguished by the block scope in which they reside.

The invocation of a non-generator program is much simpler:

```
// Effective use of a non-generator
// .. evaluate input arguments
G_success = ProgName ( actual arguments );
```

This fragment of code replaces a comment of the form "**.. body of ProgName**", and so it satisfies the general form for effective procedures. A recursive program is invoked where recursion occurs using the form above. Recall that recursive programs are never generators.

A program that realizes a function is both complete and effective. Because it is not a generator, it can be invoked as above for use as an effective procedure. To be used as a complete procedure, it is invoked as shown below.

```
// Complete use of a called function
```

www.manaraa.com

```
// .. evaluate input arguments
if (ProgName ( actual arguments )) {
   // .. success continuation
}
```

This fits the general form of a complete procedure, even though it is known ahead of time that the success continuation executes at most once.

Disjunction

The generated code for the effective realization of disjunction, s⊗t, is trivial. Both component procedures are effective.

```
// Disjunction (s⊗t)
// .. body of s
if (!G_success) {
   G_success=1;
   // .. body of t
}
```

In short: do s; if it fails, try t. In contrast, the complete implementation must generate all results for s and then all results for t. The code shown below combines **K** complete procedures into a complete disjunction.

```
// Disjunction, generator: s⊕t⊕...
{  int G_branch = 0;
   int G_loop = 1;
   while (G_loop) {
      switch (G_branch) {
      case 0:
         // .. first part of s
         goto Or_Success_N;
Or_Back_N_0:
         // .. remainder of s
         break;
      case 1:
         // .. first part of t
         goto Or_Success_N;
```

www.manaraa.com

```
Or_Back_N_1:
         // .. remainder of t
         break;
      ...
      case K:
         // .. first part of last disjunct
         goto Or_Success_N;
Or_Back_N_K:
         // .. remainder of last disjunct
         break;
      default: error ("Error in generated code\n");
      }
      if (G_branch++>=K) G_loop=0;
      continue; // while (G_loop)
Or_Success_N:
         // .. success continuation of s⊕t⊕...
      if (G_stop) break; // from while (G_loop)
      switch (G_branch) {
      case 0: goto Or_Back_N_0;
      ...
      case K: goto Or_Back_N_K;
      }
   } // while (G_loop)
}
```

The variable **G_branch** keeps track of which component procedure is currently generating result tuples. When all have run their course, the **while** loop terminates, and execution falls out of the code fragment. The label **Or_Success_N** serves as a single point where the success continuations for all the disjunctive components are combined into one.

N is a number that serves to keep distinct the labels used in the code produced for different disjunctions. Other compositions shown below also require a distinguishing number. A single global "ticket teller" is used during code production for a program to generate a sequence of unique numbers. This is a common method for generating unique names in the back-end of a compiler.

www.manaraa.com

Conjunction

The first case for conjunction combines two complete procedures to produce a complete result. It simply nests the code for the second procedure within the success continuation for the first. Recall from the table in Lm. 14 of Chapter 5 that the inner procedure must terminate.

```
// Conjunction, complete: t(s)
// .. first part of s
// Success continuation of s:
   // .. first part of t
      // .. success continuation of (t(s))
   // .. remainder of t
// .. remainder of s
```

In the second case for conjunction, a complete procedure and an effective procedure are combined to produce an effective procedure.

```
// Conjunction, effective from complete: t(s)
// .. first part of s
   // .. body of t
   if (G_success) G_stop=1;
// .. remainder of s
if (G_stop)
   G_stop=0;
else
   G_success=0;
```

The code fragment above is the first example where **G_stop** is set to break out of a complete procedure's iteration. Notice that **G_stop** is reset outside the scope of the complete procedure whose iteration is interrupted. The complete procedure can be produced by any of the code fragments described in this chapter for complete procedures, including the previous fragments for composing complete conjunctions and disjunctions. They all cause execution to fall all the way through when **G_stop** is set.

A special case deserves mention. The code fragment above can be used to turn a complete procedure into an effective procedure when the body of t is

www.manaraa.com

empty, i.e., when t realizes the formula **True**. Because **G_success** is set on entry, the **if** statement forces an exit from the iteration when the complete procedure returns its first result tuple. This special case is shown below:

```
// Conjunction, effective from complete: t(s)
{
   int G_first = 1;
   int G_loop;
   // .. evaluate input arguments
   G_loop=ProgName (&G_first, actual arguments);
   while (G_loop) {
      if (G_success) G_stop=1;
      if (G_stop)
         G_loop=0;
      else
         G_loop=ProgName (&G_first, actual arguments);
   }
}
if (G_stop)
   G_stop=0;
else
   G_success=0;
```

While the code above works, and allows a generator program to be used as an effective procedure, it can clearly be optimized. The final case for conjunction creates an effective procedure from an effective procedure for t and a total function s:

```
// Conjunction, effective from effective & total fn: t(s)
// .. body of t
if (G_success) {
   // .. body of s
}
```

Because s is a total function, it never fails. Assignments are the most common total functions.

www.manaraa.com

Existential Quantification & Negation

Existential quantification is implemented as a block that provides scope for the quantified variables, and definition of those variables at the beginning of the block. The general form is:

```
// Existential quantif ication: (∃)φ
{
   // .. declare quantif ied variables
   // .. body of effective procedure that realizes φ
   // .. deallocation of quantif ied variables
}
```

The variables are indirect, meaning that they are declared as pointers. The actual allocation of an existentially quantified variable occurs when it is given a value, either as an output variable from a procedure, or within a primitive predicate. The actual deallocation occurs where the algorithm of Section 6.4 decides. As discussed there, and below in Section 7.4, deallocation can occur through use as a destructively consumed input argument, through destructive update *via* data structure isomorphism, or simply by reaching the end of the scope.

Only an effective procedure that has no output variables can be negated. (Of course, a complete procedure can be turned into an effective one, as described above, and output variables can be shed through existential quantification.) Negation merely complements the boolean variable **`G_success`**:

```
// Negation: !s
// .. body of s (no output variables)
if (G_success)
   G_success=0;
else
   G_success=1;
```

Universal Quantification

Universal quantification of an implication — **`(∀i)(φ(i)⇒ϕ(i))`** — combines a complete procedure that generates all the values to be tested and an

www.manaraa.com

effective procedure that tests these values into an effective procedure that succeeds if all the generated values pass the test. The code for this is shown below.

```
// Universal quantif ication of implication:s/∀t
{
   // .. def inition of quantif ied variables,∀
   // .. first part of s
   // Success continuation of s:
      // .. body of t
      if (!G_success) G_stop = 1; // failed
   // .. remainder of s
   if (G_stop) {
      G_stop=0;
      G_success=0;
   }
}
```

Like existential quantification, universal quantification introduces a new block in which the quantified variables are active. Unlike existential quantification, the variables are direct, and are automatically deallocated by the C machine at the end of the block. As was seen with conjunction, **`G_stop`** is set to break out of a complete procedure's iteration, and is reset outside the complete procedure's scope.

Summary of Code Production

The exposition above describes code production for all procedure compositions discussed in Chapter 5 and Chapter 6. All that remains is to describe the implementation of primitive predicates. As promised, the variables that are artifacts of code production are summarized in the table below.

Variable	C scope	Purpose
`G_success`	function	Indicate success or failure of an effective procedure.
`G_stop`	function	Break out of complete procedure's iteration.

www.manaraa.com

Variable	C scope	Purpose
`G_loop`	block	Control complete procedure's iteration.
`G_first`	block	Indicate whether an invocation of a complete procedure is first or later for an input tuple.
`G_branch`	block	Indicate current component of complete disjunction.

7.3 Functions and Base Predicates

The programming environment must provide an initial procedure library, with procedures that realize the predicates of the base logic. Each predicate may have several realizations, with the corresponding procedures having different argument modes or semantic properties.

The back-end must also produce code for the functions in the logic. This is required when a literal is realized, since the input arguments to the literal can be arbitrary terms.

This section describes the implementation of the logic's functions and base predicates. The realization of base predicates for data structures, just like other procedure compositions, can require that assertions in the logic hold true in order for their semantic properties to obtain. Some of the realizations of the data structure predicates allocate memory, and others perform destructive updates.

Primitive Sorts

The back-end generates code for all the functions on the base sorts and the primitive data structure sorts. In other words, the compiler generates code that evaluates any term invoving only these sorts. These functions are the basic functions for arithmetic and character string manipulations. The concerned predicates for these sorts are equality and inequality comparison. These predicates

www.manaraa.com

are realized as described below.

= Realized as a function that always succeeds when one side is an input argument and the other an output argument, and as an effective procedure that always halts when both sides are input argument.

< Realized as a complete, non-looping procedure for integers when one side is an input argument and the other side is an output argument, and as an effective procedure that always halts when both sides are input arguments, for all the base sorts, except **address**, and the primitive data structure sorts, except **addr**. Note that the first realization generates sorted output that is ascending if the output argument is on the right and that is descending if it is on the left. For the second realization, the first argument is bounded above and the second is bounded below.

The corresponding implementations are also assumed for , >, , and , except that requires both sides to be input arguments.

Pointer Dereferencing

Pointer dereferencing is a special case, because it involves a predicate (=) on a base sort (**address**), but the "output" argument is an arbitrary data structure, to wit, the data structure that is referenced. Moreover, a pointer dereference, unlike the other predicates, does not make a formula true, or check that a formula is true. The form of a pointer dereference was described in Section 3.6. It is shown below.

(∃x∈y)(&x=r ∧ ...)

The pointer **r** must be an input variable to this formula. In order to dereference **r**, the formula must be true. The prerequisite assertion is:

(∃x∈y)(&x=r)

The reason for this is that a pointer carries no information. When a pointer is dereferenced, the *assumption* is that it points to a particular kind of data structure, in this case, an element of a known set. The act of dereferencing does not show the existence of **x**, but rather, makes the particular **x** available to other formulae within a scope, to the formulae masked by the elision (**...**) in the dereference example above. In the computation graph, **x** is formally an output

www.manaraa.com

variable to the literal **&x=r**, since it is available after the literal's execution. But in this case, **x** is identified, rather than produced, and at the end of the existential scope, it is not deallocated unless it is removed from the set.

Data Structure Sorts

Isomorphism, ::, is the important predicate for data structure sorts. With both sides as input, this predicate asks whether the two data structures are identical, modulo different mappings to memory, but with corresponding pointer values. With one input argument and one output argument, this predicate performs a *deep copy.* In this case, the input argument is often a term involving data structure functions.

A term (other than a variable) whose sort is a complex data structure sort is allowed only as an input argument to data structure isomorphism. Such terms are a variable, called the *underlying variable,* modified by data structure functions as described in Section 2.7. It would be easy to straightforwardly apply the modifying functions to the data structure instance, but in general, this is not possible, because the underlying variable to which the instance is bound is, like all variables in Galois, a logical variable, and therefore it cannot be modified. A value for the term is calculated in one of two ways.

The variable can be modified! In some (hopefully, most) cases, the data structure instance can be directly modified because the underlying variable will see no further use in the program, and it is scheduled for deallocation, either because it is a destructively consumed input argument, or because it is an existentially quantified variable that will not survive beyond its scope. The next section will describe how this is detected.

The variable is copied. The back-end can generate code to copy a contiguous data structure instance — i.e., one that has no sets — if it can determine the length of the arrays involved. (This is discussed below.) More generally, for complex data structures, the programmer will have to write a program that implements data structure isomorphism. (The programmer writes a predicate that

www.manaraa.com

performs operations on parts of the data structure, these being simpler operations than isomorphism of the whole, and then verifies that the defined predicate is logically equivalent to data structure isomorphism. This programming methodology is discussed in Chapter 1 and Chapter 3.)

The back-end must generate code to allocate contiguous data structure instances both for evaluating terms, as discussed above, and also for existentially quantified data structure variables. If the data structure instances do not involve arrays, or if the arrays have a fixed length, then the calculation of the instance's size is easy. If there are arrays whose size is not known, then realization of the concerned literal generates a prerequisite assertion as shown below.

len(*array_exp*) = *integer_term*

Here, the programmer must not only verify an assertion, but must also supply part of it, to wit, the right-hand side (indicated by ***integer_term***). This is the one case where this is required.

A similar prerequisite assertion is generated when an array is indexed or when a subarray expression is evaluated. Then, the concerned assertion, for each subscripting expression, is:

0 subscript_expr<*len(array_expr)*

Here, as usual, the assertion is fully generated by the front-end, and the programmer only has to verify it.

Set Sorts

As described in Chapter 3, the set valued functions have only one purpose: the control of memory management. (Memory management is discussed in more detail in the next section.) A data structure instance can be added to a set only if (1) it is bound to a destructively consumed input variable, or (2) it is bound to an existentially quantified variable. In either case, the point in the computation graph where the data structure instance is added to the set must follow every other use of it. (Determining this is discussed in the next section.) Adding a data

www.manaraa.com

structure instance to a set acts to deallocate the variable to which it is bound. Existentially quantified variables are normally deallocated because they have no purpose outside their scope, and destructively consumed input variables are normally deallocated because this is the specified semantic for the program.

The function that removes a data structure instance from a set can be applied only in a data structure term that is the input argument to a data structure isomorphism, as discussed above. The variable bound to the removed set element is destructively consumed by the literal, and the data structure instance bound to the variable is deallocated. Consider the formula below.

(∃x∈z.y)(&x=r ∧ w::z{ ...; y»x; ...})

This formula can be realized only if both **x** and **y** are destructively consumed by the literal **w::z{...}**. Obviously, this kind of fragment is intended to implement a destructive update of **y**, where certain parts of **y** are removed.

7.4 Variable Allocation & Deallocation

The code to allocate and deallocate contiguous data structures is trivial *if* the amount of memory is known. A deallocate node has a precondition identical to that described for allocation of a data structure term in the previous section, i.e., the length of an array must be known.

Deallocation of more complex data structure instances requires a program that realizes data structure isomorphism, i.e., that performs a deep copy. A program that realizes a deep copy can be automatically tweaked to deallocate its input data structure . When it is compiled, the following changes are made. First, every memory allocation for the output variable is pushed onto a stack. Second, before the procedure's successful return, it iterates through the stack and deallocates all the concerned memory. This tweak will convert a procedure **p(x, y)**, with destructively consumed **x** and output **y**, that implements **x::y**, into a procedure **p'(x)** that deallocates **x**.

www.manaraa.com

8. Compilation Examples

Two examples of compilation are presented below. The first is a simple example that permits the workings of the procedural calculus and code production to be followed in detail. The second example is the program **doAscPtsInsert** from Section 3.5, which inserts a point into an ordered linked list. This example uses dynamic memory, and illustrates the **OptimizeDeallocation** algorithm from Section 6.4.

8.1 A Simple Example

The example program is an array search. The predicate below (**InArray**) is true of an array, upper bound, key value, and index when (1) the index is non-negative and less than the upper bound, and (2) the indexed element in the array equals the key value. The specified program (**FindInArray**) realizes the predicate as a search for an index value.

```
// Define predicate InArray
InArray (key: int, a: int[], uprBd: int, idx: int) ⇔
(∃j: integer) (
   0 j<uprBd∧ a[j]=key ∧ idx=j
)

// Specify program FindInArray
{uprBd len(a)}
FindInArray (in key, in a, in uprBd, out idx)
{InArray} effective, terminates.
```

The program specification says that the program **FindInArray** realizes the predicate with all arguments as input, except for **idx**, which is an output argument. It further qualifies the program with a precondition on the predicate's input arguments: the program will be executed only when (is guaranteed to work only when) **uprBd len(a)**. The program is *effective*, meaning that it produces only

www.manaraa.com

one output value that satisfies the predicate. The program returns a status of success if there is such an output value; otherwise, it returns a status of failure.

The compilation of this program is described below. In this example, we focus on the procedural calculus, and ignore the equivalent route through computation graphs.

Following the recursive form of the **Compile** procedure from Section 5.3, begin by letting $\mathcal{C}_0$ represent the desired procedure composition. The first logical construct is **(∃j)**φ. The procedural calculus only provides one rule for existential quantification. So let j be the projection procedure that sets the scope for an existential variable. Then:

$\mathcal{C}_0$ = j($\mathcal{C}_1$), and

$\mathcal{C}_1$ is the procedure composition that realizes φ.

φ is the conjunction: **(0 j<uprBd**∧ θ**)**. There are two rules for effectively realizing conjunction. (See the first two rows of table in Lm. 14 of Chapter 5.) The library contains procedures that realize the literal **0 j<uprBd** either effectively or completely, so at this point, there is no way to choose between the two rules. Choosing the second rule would cause further compilation to fail, because the second component of the conjunct, θ, cannot be realized as a total function. If this rule were tried first, the **Compile** algorithm would backtrack to where this rule was chosen, and then try the remaining rule. We will proceed with the one rule that results in a successful compilation. Thus:

u is a complete realization of **0 j<uprBd**with output **j**,

$\mathcal{C}_0$ = j(u ($\mathcal{C}_2$)), and

$\mathcal{C}_2$ is the procedure composition that effectively realizes θ.

θ is the conjunction **a[j]=key** ∧ **idx=j**. This brings us to the tail of the recursive descent, since both components of the conjunction are literals. Only the second rule for conjunction applies to θ. The first literal is realized as a characteristic function that is not total, and the second literal is realized as a total

www.manaraa.com

function whose output is **idx**. Thus:

t is an effective realization of **a[j]=key**,

s is a total function that realizes **idx=j**, and

$C_0 = j(u(t(s)))$.

The Compile algorithm calculates the precondition for a procedure composition from the inside out. The procedure s has no precondition (or to think of it another way, it has the precondition **True**). The procedure t has the precondition **0 j<len(a)**, which is unchanged by its composition with s. The procedure u also lacks a precondition, but in sequential composition, it weakens the precondition of t(s), so that the precondition of u(t(s)) is **0 j<uprBd** ⇒ **0 j<len(a)**. The projection procedure has no precondition.

Before producing code, the prerequisite assertion must be verified. Recall from Section 5.3 that the prerequisite assertion is $\alpha \Rightarrow \beta$, where is the precondition specified for the program, in this case, **0 bd ln(a)**, and b is the precondition for the procedure composition, which was just calculated: **0 j<uprBd** ⇒ **0 j<len(a)**. Thus, the prerequisite assertion is:

0 bd ln(a) ⇒ **(0 j<uprBd** ⇒ **0 j<len(a))**

In this case, the prerequisite assertion is a trivial matter of arithmetic that is easily verified by an automatic theorem prover, so the programmer would not have to intervene. Notice that if the program specification had lacked a precondition, the theorem prover, equally easily, would have shown that **(0 j<uprBd** ⇒ **0 j<len(a))** is *not* a theorem. This would force the programmer to modify the predicate definition or to put the needed precondition into the program specification. Whenever **FindInArray** is used in other Galois programs, the compilation algorithm will insist that it is invoked with a search bound that is within the array limits, by virtue of this being its precondition, just as the compilation algorithm insists that array subscripting, **a[j]**, satisfies its precondition for its use in **FindInArray**.

C code is trivially generated for this procedure composition, following

www.manaraa.com

the scripts in Chapter 7. First, the skeleton for an effective function is laid. Note the one level of indirection used with the argument **idx**, because it is an output argument.

```
int FindInArray (int key, int a[], int uprBd, int* idx) {
{
   int G_success=1;
   int G_stop=0;
   // .. body of j(u(t(s)))
   return G_success;
}
```

Next, a scope for existential quantification is put in. Because **j** is neither an array nor a **struct**, it is represented directly as a C automatic variable and no explicit deallocation is required:

```
int FindInArray (int key, int a[], int uprBd, int* idx) {
{
   int G_success=1;
   int G_stop=0;
   // Begin existential scope
   {
      int j;
      // .. body of u(t(s))
   }
   // End existential scope
   return G_success;
}
```

The procedure u is completely realized. It has a first part and a remainder part, surrounding its success continuation. According to the composition rule, the success continuation contains a check to exit the loop:

```
int FindInArray (int key, int a[], int uprBd, int* idx) {
{
   int G_success=1;
   int G_stop=0;
   // Begin existential scope
   {
```

www.manaraa.com

```
      int j;
      // Conjunction, effective from complete
      j=-1;
      while (!G_stop && 0<=++j && j<uprBd) {
         // .. body of t(s)
         if (G_success) G_stop=1;
      }
      if (G_stop) G_stop=0;
      else G_success=0;
   }
   // End existential scope
   return G_success;
}
```

The inner two procedures are effectively realized, and so they generate very simple code. The complete C code for **`FindInArray`** is shown below.

```
int FindInArray (int key, int a[], int uprBd, int* idx) {
{
   int G_success=1;
   int G_stop=0;
   // Begin existential scope
   {
      int j;
      // Conjunction, effective from complete
      j=-1;
      while (!G_stop && 0<=++j && j<uprBd) {
         if (a[j]==key) G_success = 1; // Code for t
         else G_success = 0; // Code for t
         if (G_success) *idx=j; // Code for s
         if (G_success) G_stop=1;
      }
      if (G_stop) G_stop=0;
      else G_success=0;
   }
   // End existential scope
   return G_success;
}
```

www.manaraa.com

The scripts from Section 7.2 produce code that exhibits some awkward passages. But except for the excessive checking of `G_success`, and the use of the extra variable `G_stop` to break out of loops, the code above is a straight-forward implementation of the specified program. (A good C compiler would optimize the tests on common expressions, producing an executable that runs with respectable efficiency.)

8.2 An Example with Dynamic Memory

Given that a program does not use dynamic memory, it can be compiled without converting its procedural composition into a computation graph. (But as Chapter 6 notes, there are human interface reasons to use the computation graph.) The next example manipulates dynamic memory. The explanation of this example will focus on this aspect of compilation. The example is from Section 3.5. It is the specification for a program that allocates a point and inserts it into an ordered linked list. The computation graph that results from applying the constructions in Chapter 6 is shown in Figure 1.

www.manaraa.com

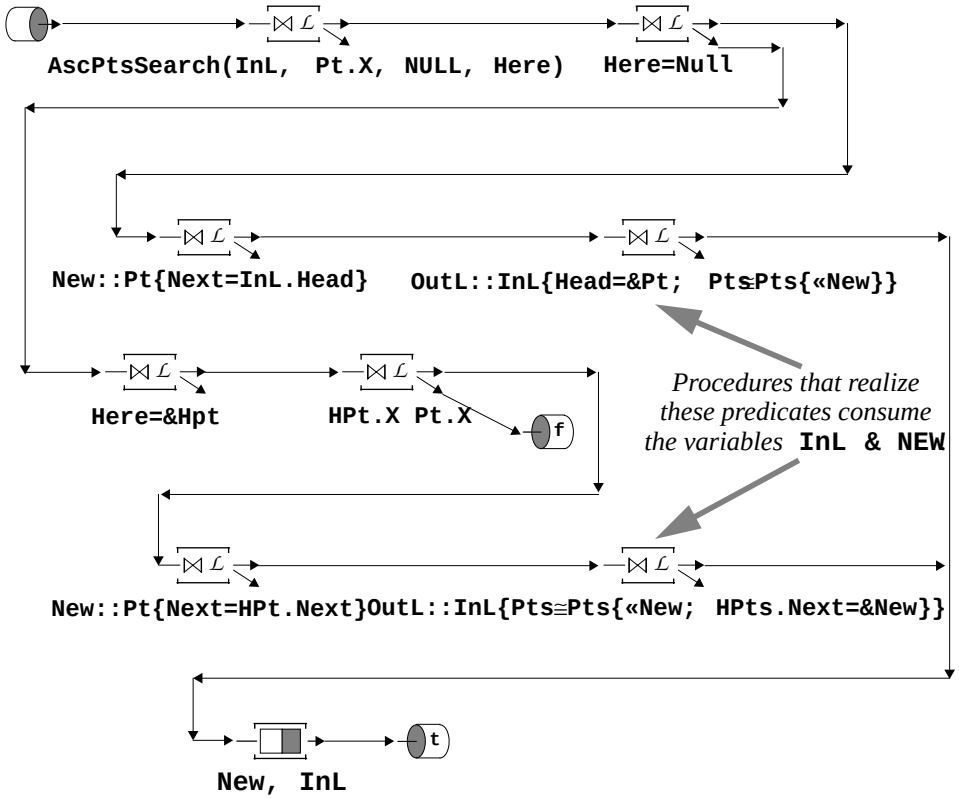


Figure 1: Computation graph for **doAscPtsInsert** prior to adjustments for variable deallocation.

Code cannot be produced for this graph, because the variables **New** and **InL** are not deallocated until just before the success node, implying that they are available until then. Conflicting with this requirement, the only procedures known to the compiler that realize the literals **OutL::InL{..Pts@Pts{«New ..}}** consume both of these variables. (The compiler can automatically realize these literals only as a destructive update to **InL** that also consumes **New**.) This problem is alleviated by the **OptimizeDeallocate** algorithm from Section 6.4. Its execution pushes the deallocation nodes forward in the graph, producing the computation graph shown in Figure 2.

www.manaraa.com

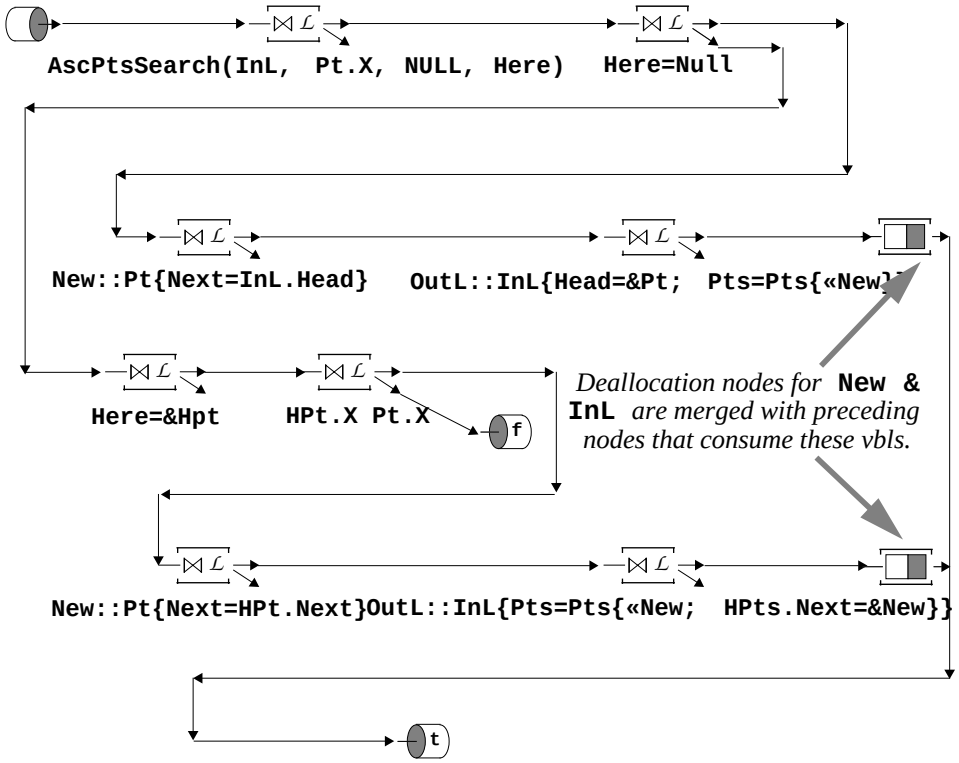


Figure 2: Computation graph for **doAscPtsInsert** after adjustments for variable deallocation.

In this computation graph, the deallocation nodes for the variables **New & InL** appear only after the literals that must consume these variables. Following the rule prescribed in Section 6.4, the deallocation nodes are removed. The computation graph then meets the syntactic rules on the signatures for nodes and arcs. In other words, variable allocation and deallocation will be handled correctly. The resulting code is shown below. It resides in the scope of the necessary **struct** definitions. Comments beginning with "**// ****" indicate how the code is changed because of the adjustment for dynamic memory.

```
int doAscPtsInsert (void* InL, Point Pt, void** OutL) {
{
   int G_success=1;
   int G_stop=0;
   // Begin existential scope
   {
      void* Here;
```

www.manaraa.com

```
Point* New; // Indirect because it is a struct
// Effective invocation of doAscPtsSearch
G_success = doAscPtsSearch
   (*((PointList*)InL), Pt.X, NULL, &Here);
if (G_success) {
   // Effective realization of if-then-else
   if (Here==NULL) {
      // Realize New::Pt{Next=InL.Head}
      New = (Point*) malloc (sizeof(Point));
      New->X=Pt.X;
      New->Y=Pt.Y;
      New->Next = ((PointList*)InL)->Head;
      // Realize OutL::InL{Head=&New; Pts≅Pts{«New}}
      *OutL=InL; // Consume InL
      ((PointList*)(*OutL))->Head=New;
   }
   else
      // Begin existential scope
      {
         void* HPt;
         HPt=Here; // Dereference Here
         // Effective conj. from eff & total fn.
         if (((Point*)HPt)->X!=Pt.X) G_success=1;
         else G_success=0;
         if (G_success) {
            // Realize New::Pt{Next=HPt.Next}
            New = (Point*)malloc (sizeof(Point));
            New->X=Pt.X;
            New->Y=Pt.Y;
            New->Next=((Point*)HPt)->Next;
            // OutL::InL{Pts≅Pts{«New; HPt.Next...}}
            *OutL=InL; // Consume InL
            ((Point*)HPt)->Next=New;
         }
      }
   }
}
```

www.manaraa.com

```
            // ** At this point, New would have been deallocated
            // ** if G_success were set.
        }
        // End existential scope
        // ** At this point, InL would have been deallocated
        // ** if G_success were set.
        return G_success;
    }
```

Again, the code above is awkward in places, but the implementation is reasonable. There are several important points. First, the literals where **`New`** is added to a set consume it by virtue of returning it as part of the output data structure. There is no code generated for this. By definition, **`New`** becomes an element in the set of the output data structure **`OutL`**. It is the programmers responsibility to create data structures where all parts are reachable. (Using theorem proving tools, he can prove this as a theorem in the logic.)

The procedures that produce **`New`** as output, i.e., the procedures that realize the two literals of the form **`New::Pt{..}`**, must allocate **`New`**. As described in Chapter 5, this is part of their role in producing an output data structure. In the code above, these procedures are realized as inline code, but the reader should consider that this code could have been encapsulated in a C function. **`New`** is not explicitly deallocated anywhere in the above code, because the program cannot fail between its allocation and its return to the application as part of **`OutL`**.

Finally, note that the data structure isomorphism that defines **`OutL`** is realized through pointer assignment and destructive update, as discussed throughout much of this work.

www.manaraa.com

9. Conclusion

In closing, we review what this research has accomplished, describe the status of current development, and discuss directions for future research.

9.1 Accomplishments

In [39], Lowry describes how software engineering is moving toward deeper descriptions of software systems, i.e., descriptions that include knowledge about software from which programming environments can automatically or semi-automatically deduce revisions, extensions, and compositions of existing software systems to meet new and revised specifications. This trend, which the author believes both desirable and inevitable, requires the representation of knowledge — in rules or through logic — about the covered programming domains. Automatic programming systems have encapsulated the knowledge for specific application domains such as oil-well logging and VLSI design, and for general high-level algorithms. Graphical user interface (GUI) development environments embody rules about composing the various widgets and windows that are presented to an interactive user and about coordinating the interaction between these and the underlying application.[1] Traditional logic programming and production systems support direct rule-based programming against high-level data types.

The research in these areas has, until now, either ignored the problem of general data structure programming or has assumed that component libraries for this purpose would somehow be provided. *If* programming in the future does not require many new data structures *and* imperative languages support sufficiently flexible software reuse, then this assumption is practical. Otherwise, we need a representation for data structure programming that supports reasoning from and

[1] GUI development environments are not traditionally viewed as a kind of automatic programming, but because of the noted characteristics, they fit within this category, at least for the purpose of the point being discussed.

www.manaraa.com

building of a knowledge base that in turn supports constructing data structure programs and proving their properties.

This research provides a representation and a compilation technique for data structure programs that:

- generates executable code from logical specifications,
- makes available the efficiencies of pointer reference and destructive update,
- facilitates proofs of the properties of these programs, including termination and the maintenance of data structure invariants, and
- supports the construction of libraries of the proven theorems and generated programs for reuse, so that more complex data structures can be built on and verified from the programs and knowledge about simpler data structures.

This research has led to subsequent work by others and the author to develop a prototype compiler and programming environment, and to explore the ramifications of this technology.

9.2 Current Development

Under a TARP grant [11] stemming from this research, several colleagues and I have built a prototype Galois compiler. This compiler implements the code generation described in this dissertation, producing C code. It supports a simple procedure library. Computation graphs are created and displayed, and these are used to show the programmer the prerequisite assertions required for code generation.

This current prototype suffers a few lacunae. Two of these will be addressed by work planned under the above-mentioned grant. First, the programming environment does not — yet — include an integrated theorem prover and, because of this, there is only little support for a theorem library. We plan to build a bridge to the Boyer-Moore theorem prover. As a first step, the axioms of the logic of Galois have been translated into rules in this theorem-prover. Coming steps will integrate the theorem-prover into the programming environment and

www.manaraa.com

will increase support for a theorem library.

Second, we need more experience with this technology to learn where it works well and where it has unexpected problems. This is also part of ongoing work.

9.3 Future Research

The first two directions offered for future research — object-oriented programming and parallel programming — extend Galois in natural directions. The third direction discussed pertains to Galois, but has much larger scope.

Support for Object-Oriented Programming

Galois supports a simple kind of specialization hierarchy. Data structure classes are defined by logical invariants (logical predicates), and one data structure class is a specialization of a second if its invariant is logically stronger. For example, an ordered array is a heap, which is an array. These data structure classes are defined through invariants — **`ordered(a)`**, **`heap(a)`**, and **`array(a)`** — and the specialization hierarchy is reflected in theorems that show the logical strength of these invariants: **`ordered(a)`**⇒**`heap(a)`** and **`heap(a)`**⇒**`array(a)`**. Given these strength theorems, general theorems that apply to the more general data structures also apply to the more specialized data structures, and because of this, programs that function with more general data structures also function with the more specialized data structures.

In Galois, this kind of specialization operates on one underlying sort. Indeed, a theorem such as **`A(x)`**⇒**`B(x)`** is not syntactically well-formed unless the predicates **`A`** and **`B`** take an argument of the same sort. This is clearly a limitation on Galois's ability to deal with a specialization hierarchy.

In the area of object-oriented programming, we know well the benefits of polymorphism, where a procedure p(x) can take as an argument any object x that belongs to some class or any specialized subclass. Polymorphic logics and

www.manaraa.com

logics for reasoning about classes have been explored [30, 31], but not realized as logic programming languages.

The integration of logic programming and object-oriented programming holds the potential of solving one of the major problems in object-oriented programming. Programmers in C++, Eiffel, and other object-oriented programming languages often find that existing classes cannot be reused through derivation without modification. This is because (1) the imperative code that implements a class makes assumptions about how derived classes work, and (2) the designer of a class cannot anticipate the demands that will be placed on the class by future derivation. If methods were implemented through logical specification rather than through procedural code, (1) would be much less of a problem. The logical specification of each public method would be part of the class's interface. A derived class could augment this logical specification in creating its version of the same method. Despite the fact that this might yield very different procedural code, the specification of the base method is reused, which is to say, the work that went into creating the specification is put to greater good. In short, if methods were specified in logic, it would be possible in creating a derived class to inherit and modify base methods at a level that supports more reuse than by invoking encapsulated imperative code.

Support for Parallelism

To deal with pointers and non-contiguous data structures in the logic, it was necessary to make all parts of a data structure explicit, using the set construct, and to impose assumptions that restrict aliasing among output arguments and destructively consumed input arguments. These same provisions make the computation graph an expression of data and control dependency. Wherever the computation graph forks and joins, indicating the realization of disjunction or if-then-else, the two paths can be executed in parallel. If there is a test at the branch, and the whole is effectively realized — i.e., the program realizes an if-then-else — then such parallel execution would be speculative, since only the results of one branch are needed.

www.manaraa.com

In the case of sequential composition, parallel execution is possible when there is no data dependency, i.e., when none of the input variables to the second procedure of the composition are outputs from the first procedure. These two rules allow the computation graph to be interpreted as an expression of parallelism much as the graphs in parallel programming environments, such as CODE [12, 49], and reengineering tools such as E/SP [61].

There are more subtle kinds of parallelism that are not so easily deduced from the computation graphs. In particular, it is often useful to unroll recursion and execute the different depths of a recursive procedure in parallel. Furthermore, the division of a program into predicates and procedures for conceptual and functional purposes does not always reflect the best division for parallelism. Interprocedural parallelism requires analysis of dependencies between pieces of a procedure and pieces of its calling procedures.

These kinds of analyses have been explored [18] for traditional programming languages. Galois brings the potential that the same logic that expresses programs can be used to reason about control and data dependencies. Predicates such as **`Disjoint`** directly express information about data dependency.

Integration of Knowledge-Based Programming Systems

This chapter began by emphasizing the need for an automatic programming system that targets the problem of data structure programming. Of course, automatic programming has already proved useful for particular application domains, for transaction based modeling, for general high-level algorithms, and — under the guise of interface builders — for graphical user interfaces. The development of most real programs usually involves more than one of these aspects, e.g., a new program for modeling oil field depletion might use a new graphical user interface, extend some existing high-level algorithms for oil field modeling, and use some new data structures to efficiently realize these algorithms. Even when automatic programming systems are used to build some parts of a program, traditional programming languages are relied upon as the

www.manaraa.com

common ground of last resort to glue the pieces together.

As long as this situation holds, software engineers will continue to view traditional programming languages as their major tool of the trade, with each engineer learning the one or two automatic programming systems that are most applicable to their problems. This raises the question of whether it is possible to create an automatic programming system that provides this glue in a way that supports greater reasoning about and reuse of software components than do traditional languages. If such a system were to rely on GUI development environments for user interaction, on various automatic programming systems for high-level and domain specific algorithms, and on Galois for data structure programming, then it would not have to include facilities to deal directly with any of these aspects of a program. It would only provide the glue. But if this glue is not made of user-interaction, algorithms, or data structures, of what does it consist? And what kind of framework can deal sensibly with such diverse systems for creating software components? To these questions, the author has no ready answers.

www.manaraa.com

Glossary

Address value. See *data structure instance*.

Application. Galois programs are designed to be components invoked from a larger application program. This larger application program is written in a conventional programming language, and must abide by certain conventions to correctly use Galois programs. (Galois programs that invoke other Galois programs are guaranteed by the compiler to always abide by these conventions.)

Atom. A positive literal. (See also *literal*.)

Base logic. The base logic is that described in Chapter 2. Programming extends the logic with new predicates that are recursively defined.

Base sorts. The base sorts are **`integer`**, **`floatingPt`**, **`character`**, and **`address`**. These sorts are *not* data structure sorts, and the values in the domains of these sorts are atomic.

Calculus of procedure composition. A set of operations that compose procedures (q.v.) and a set of rules that calculate the semantic definition (q.v.) of the resulting procedure from the semantic definitions of the component procedures.

Complete. See *execution properties*.

Computation graph. A directed graph that is generated from a program specification that shows the flow of tuples and the operations on these tuples at each node in the graph.

Consumed input (signature, tuple, variable). The destructively consumed input variables are those whose bindings are broken and whose values are deallocated by the successful execution of a procedure. These variables constitute, as a set, the procedure's destructively consumed input signature, and the values bound to them on procedure execution constitute, as a set, the destructively consumed input tuple.

www.manaraa.com

Data equality. Two data structure instances are data equal if their data values are equal, i.e., the instances $<a, u>$ and $<b, v>$ are data equal iff $u \cong v$. (See also *strict equality* and *data structure isomorphism*.)

Data structure class. All data structure instances that have a common sort and that satisfy a common invariant, i.e., if **`q(x)`** is a predicate whose argument is a data structure sort, then all data structure instances that satisfy **`q(x)`** compose a data structure class.

Data structure instance. A data structure instance is a value in the domain of a data structure sort. In the intended model, a data structure instance is an ordered pair $<a, v>$, where a is the instance's *address value*, and v is the instance's *data value*.

Data structure isomorphism. Two data structure instances are isomorphic if (1) their data values — excluding embedded pointers — are equal, and (2) analogous embedded pointers reference analogous parts of the data structure instances. Data structure isomorphism formalizes the notion of two data structure instances being "the same." (See also *data equality* and *strict equality*.)

Data value. See data structure instance.

Domain. A sort's domain is the set of values that provide denotations to terms in the sort. The domain of a constant, variable, or term of a logic is the domain of the constant's, variable's, or term's sort. The domain of a signature is the cross-product of the domains of its variables. The symbol is used to denote domains. In the relational algebra, variables are directly assigned to domains.

Effective. See *execution properties*.

Equality. See *strict equality*, *data equality*, and *data structure isomorphism*.

Execution properties. The result sequence of a procedure can display several properties relative to a postcondition. The procedure is *sound* if every tuple in the result sequence, together with the input tuple, satisfies the postcondition. It is *effective* if it is sound, and whenever there exists a qualified result tuple, the result sequence is just one such qualified result tuple, followed by . It is *complete* if the result sequence includes all qualified result tuples, up to data structure isomorphism. The procedure *terminates* if the result sequence ends in . (See also*semantic definition*.)

www.manaraa.com

Function. (1) A procedure that is both effective and complete, i.e., a procedure whose postcondition qualifies at most one result tuple (up to data structure isomorphism) and that produces this tuple if it exists. A function is *total* if a qualified result tuple exists for each input tuple. (2) The unit of encapsulation of C code.

Generator. A program that is complete, but that is not a function. In other words, a generator is a program that may produce many result tuples for each input tuple. Because of this, it has a different calling convention than a program that is not a generator.

Input (signature, tuple, variable). The union of the pure input (q.v.) and destructively consumed input (q.v.) signature, tuple, or variables.

Interpretation. A denotational semantic for a logic. In this work, the logic is interpreted in a relational algebra. The denotation of a formula is a relation with the same signature. A formula is *false* if it denotes the empty relation with empty signature. A formula is *true* if it denotes the relation with empty signature that contains just the null tuple.

Invariant. A predicate, usually with one argument, that qualifies a class of data structure instances, e.g., `Btree(x)`. See also *data structure class*.

Literal. A predicate whose arguments are replaced by syntactically correct terms (q.v.) of the appropriate sorts, optionally preceded by a negation symbol. If preceded by a negation symbol, the formula is a *negative literal*, otherwise it is a *positive literal*.

Logic. A logic comprises three things: (1) a formally defined *language*, (2) a set of sentences in the language, called the logic's *axioms*, and (3) *rules of inference* that allow one to form proofs by deriving new theorems from existing theorems. A logic's sentences are called *formulae*. A logic is *recursive* if its formulae and axioms are recursive sets. Traditional logics have languages that follow a particular syntactic pattern: atoms are predicate symbols applied to terms (q.v.), and formulae are atoms or other formulae that are combined through the boolean participles and quantifiers. A traditional logic is *first-order* if the domain of quantified variables does not include functions, predicates and formulae.

www.manaraa.com

Mode. The use (direction of data flow) of an argument to a program or a free variable of a procedure. There are three modes. *In:* a pure input variable provides a value from the execution environment that remains after execution ends. *Inx:* a destructively consumed input variable provides a value from the execution environment that is no longer available after execution is successful. *Out:* an output variable provides a value to the programming environment after execution is successful, but is not bound prior to execution. (See also *semantic definition.*)

Model. A model is an interpretation in which every theorem of the logic is true.

Output (signature, tuple, variable). When a procedure successfully returns, new values are bound to its output variables. The set of values constitutes the output tuple. (See also *consumed input* and *pure input*.)

Postcondition. A logical formula whose free variables are contained in the union of the input and result signatures that qualifies the set of valid result tuples for each input tuple. For a program, the postcondition is a predicate whose arguments are the same as the specified program's. (See also *semantic definition.*)

Precondition. A logical formula whose free variables are contained in the input signature that qualifies the set of input tuples on which a procedure will correctly function. (See also *semantic definition*.)

Prerequisite assertions. The prerequisite assertions are a set of closed formulae in the logic that the front-end generates at the same time that it generates a computation graph. The procedure represented by the computation graph is guaranteed to meet its semantic properties only if these assertions are true. (These assertions can also be viewed as open formulae attached to particular nodes on the computation graph.)

Procedural Calculus. See c*alculus of procedure composition*.

Procedure. Formally, a procedure is a function that maps an input tuple into a *sequence* of output tuples. A procedure is implemented as a fragment of code in the target language of the compiler. The calculus of procedure composition gives rules for composing procedures. These compositions correspond to different ways of gluing together code fragments.

www.manaraa.com

Program. A program is a procedure (q.v.) that satisfies a program definition, that is bound to the program name designated in the program specification (q.v.), and that is saved in the program library for reuse in creating larger programs. The example back-end implements a program as a C function.

Program specification. A program specification is a semantic definition (q.v.) for the program and a name for the program, where the postcondition for the program is a named predicate whose arguments are the union of the input signature and result signature.

Pure input (signature, tuple, variable). The pure input variables are those whose bindings remain across execution of a procedure. These variables constitute, as a set, the procedure's pure input signature, and the values bound to them on procedure execution constitute, as a set, the pure input tuple.

Recursive predicate definition. A formula of the form $\texttt{p(x,...z)} \Leftrightarrow \Phi_{\texttt{p}}$ recursively defines the new predicate symbol **p**. The *body* of the definition, $\Phi_{\texttt{p}}$, is any formula, perhaps involving **p**, whose free variables are **x,...z**. Given restrictions on the use of **p** in $\Phi_{\texttt{p}}$, in particular, that the **p** occurs at only an even negation depth within $\Phi_{\texttt{p}}$, **p** may be added to the logic and an interpretation of **p** may be added to the logic's model.

Relation. A signature and a set of tuples with that signature.

Result (signature, tuple, variable). When a procedure successfully returns, new values are bound to its output variables (q.v.) and original values remain bound to its pure input variables (q.v.). These variables together are the result signature, and the binding is a result tuple.

Semantic definition. The semantic definition of a *procedure* comprises (1) its *input* signature I and *result* signature Q, (2) a *precondition* whose free variables are contained in the input signature, (3) a *postcondition* whose free variables are contained in the union of the input signature and result signature, and (4) a set of *execution properties*. A procedure meets its semantic definition if for any tuple in the precondition, its output sequence satisfies the execution properties relative to the postcondition. (The italicized phrases also appear in this glossary. See also *program specification*.)

www.manaraa.com

Semantic map. For a formula φ where V(θ)=$I \cup Q$, the *semantic map* is the function f: $I \rightarrow 2^Q$ that maps each input tuple in I into the set of result tuples in Q where each result tuple in the set, together with the input tuple, satisfy φ.

Semantics. The assignment of *meaning* to a language. *Procedural semantics* give meaning to a programming language by describing the computation performed by each syntactically correct program. Any kind of semantics that is not procedural is said to be *declarative*. *Formal semantics* assign meaning through a formal theory. *Denotational semantics* assign meaning through a mathematical function that maps each syntactic part of the language to a mathematical object. This object is called the text's *denotation*, and the text is said to *express* or *denote* this object. *Examples:* The description of C in the ANSI standard is an informal, procedural semantics. The Hoare calculus provides a formal, declarative semantics. The usual model theory of logic is a denotational semantics.

Signature. A *relational signature* is a set of variables. Every variable has a sort, so a relational signature characterizes the sort of a predicate or formula, or the type of a relation. A *term signature* is a sequence of sorts, the first of which is the sort of the function or term, and the remainder of which are the sorts of the term's free variables or the function's arguments. Given a term `t` and a variable `x` that has the same sort as `t` and that is not free in `t`, a relational signature for the predicate `x=t` is equivalent to the term signature of `t`.

Sort. The syntactic categories of terms in a logic are its sorts. In a single sorted logic, such as first-order arithmetic, there is only one sort, which is the integer sort in the case of arithmetic. A logic can express different kinds of objects by having multiple sorts, for example, boolean and integer. (See [19] for an exposition on multi-sorted logic.) Sorts are roughly analogous to types in programming languages. See also, *domain*.

Strict equality. Strict equality (≅) is the usual notion of equality that permits in all contexts the substitution of "equals for equals," and that denotes semantic identity. This notion of equality is different from the informal notion of two data structure instances being "the same," since the latter is often applied to data structure instances that don't have the same address value. (See also *data equality* and *data structure isomorphism*.)

www.manaraa.com

Term. Traditional logics include functions. Terms are the syntactically correct, recursive application of functions to constants and variables. Thus, a constant **c** or variable **v** of sort **s** is a term of sort **s**. If **f** is a a function with signature ($s_0, \ldots s_n$) and $t_1, \ldots t_n$ are terms of sort $s_1, \ldots s_n$, respectively, then $f(t_1, \ldots t_n)$ is a term of sort s_0.

Terminates. See *execution properties*.

Theorem. A logic has a deductive apparatus consisting of *axioms* and *rules of inference*. Every formula that can be derived from the axioms by applying the rules of inference is a theorem.

Tuple. An assignment of values to a set of variables. The value assigned to a variable must be in the domain of the variable's sort. The *null tuple* is just the empty set.

www.manaraa.com

Bibliography

1. Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press*, 1991

2. Apt, Krzysztof R., *Introduction to Logic Programming*, Technical Report, Centre for Mathematics and Computer Science, PO Box 4079, 1009 AB Amsterdam, The Netherlands

3. Ambler, A. L., Good, D. I., et al, *GYPSY: A Language for the Specification and Implementation of Verifiable Programs*, Proc. ACM Conf. on Language Design for Reliable Software, 1977, pp. 1-10

4. Backus, John, *Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, CACM, v21, n8, August 1978

5. Baldwin, Doug, *Consul: A Parallel Constraint Language*, IEEE Software, 0740-7459/89/0700/0062

6. Barstow, D. R., *Domain-Specific Automatic Programming*, IEEE Trans. on Software Engineering, v SE-11, n11 (Nov. 1985), pp.1321-1336

7. Beeri, Catriel, et al, *Embedding Ψ-terms in a Horn-clause Logic Language*, MCC Technical Report, ACA-ST-050-88

8. Berg, H. K., et al, *Formal Methods of Program Verification and Specification*, Prentice-Hall, 1982

9. Boolos, George S. and Jeffrey, Richard C., *Computability and Logic, Third edition*, Cambridge University Press, 1989

10. Bowen, K. A., *Programming with Full First-Order Logic*, in *Artificial Intelligence 10*, 1982, 421-440

11. Browne, J. C., and Turpin, R., *Software Component Reuse: Formal Specification of Update Operations on Complex Data Structures*, UT Computer Sciences, TARP Proposal, 1991

www.manaraa.com

12. Browne, J. C., Lee, T. J., and Werth, J., *Experimental Evaluation fo a Reusability Oriented Parallel Progamming Environment*, IEEE Trans. Software Engineering, v16, n2, 1990

13. Cardelli, Luca, and Mitchell, John C., *Operations on Records*, Technical Report, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301

14. Clocksin, W. F., and Mellish, C. S., *Programming in Prolog*, Springer-Verlag 1981

15. Chang, Chin-Liang, and Lee, Richard Char-tung, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973

16. Corcoran, John, et al, *String Theory*, Journal of Symbolic Logic, v 39, n 4, Dec 1974

17. Engels, G., et al, *Graph-grammar Engineering: A Software Specification Method*, in *Graph Grammars and Their Application to Computer Science*, ed. Nagl & Rozenberg, Springer-Verlag, 1986

18. Ferrante, J., et al, *The Program Dependence Graph and its Use in Optimization*, ACM TOPLAS, v9, n3, July 1987

19. Gallier, Jean H., *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, 1986

20. Gelder, A. Van, Ross, K. A., and Schlipf, J. S., *The Well-Founded Semantics for General Logic Programs*, JACM, v38 n3, July 1991

21. Goguen, Joseph A., and Meseguer, Jose, *EQLOG: Equality, Types, and Generic Modules for Logic Programming*, in *Logic Programming: Functions, Relations, and Equations*, ed. DeGroot, Doug, and Lindstrom, Gary

22. Gordon, Michael J. C., *The Denotational Description of Programming Languages, An Introduction*, Springer-Verlag, 1979

23. Guttag, J. V., and Horning, J. J., *The Algebraic Specification of Abstract Data Types*

24. Harper, Robert W. and Pierce, Benjamin C., *Extensible Records Without Subsumption*, Technical Report, School of Computer Science, Carnegie Mellon, CMU-CS-90-102

www.manaraa.com

25. Hoare, C. A. R., *An axiomatic basis for computer programming*, CACM, October 1969

26. Hoare, C. A. R., *Recursive Data Structures*, Intl Jnl of Computer and Information Sciences, v4 n2, pp. 105-131

27. Hunter, Geoffrey, *Metalogic: An Introduction to the Metatheory of First Order Logic*, University of California Press

28. Iscoe, Neil, *Domain Specific Programming: an Object-Oriented and Knowledge-Based Approach to Specification and Generation*, Ph.D. Dissertation, University of Texas at Austin

29. Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language, Second Edition*, Prentice Hall, 1988

30. Kifer, M., and Lausen, G., *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme*, ACM SIGMOD Conf. on Mgt of Data, May 1989. pp. 134-146

31. Kifer, Michael and Wu, James, *A Logic for Programming with Complex Objects*, Technical Report, Dept of Computer Science, SUNY at Stony Brook

32. Knuth, Donald E., *The Art of Computer Programming, Volume 1, Second Edition*, Addison-Wesley, 1973

33. Kowalski, R., *Algorithm=Logic+Control*, CACM, 22 (1979), pp. 424-431

34. Krishnamurthy, Ravi, and Naqvi, Shamim, *Towards a Real Horn Clause Language*, MCC Technical Report, ACA-ST-077-88

35. Kumar, Mohan, *A Compiler for Galois*, Master's Thesis, University of Texas at Austin, forthcoming

36. Leler, William, *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, 1988

37. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1984

38. Lowry, Michael R., and McCartney, Robert D., ed., *Automatic Software Design*, AAAI Press, 1991

www.manaraa.com

39. Lowry, Michael R., *Software Engineering in the Twenty-First Century*, in [38]

40. Maier, D., *A Logic for Objects*, in *Proc of the Workshop on the Foundations of Deductive Databases and Logic Programming*, Washington DC, August 1986, pp. 6-26

41. Meyer, Bertrand, *Applying "Design by Contract,"* IEEE Computer, v25, n10, October 1992

42. Miwelski, Jaroslaw, *Functional Data Structures as Updatable Objects*, IEEE Transactions on Software Engineering, v16 n12, pp. 1427-1432

43. Montanari, Ugo, and Rossi, Francesca, *An Efficient Algorithm for the Solution of Hierarchical Networks of Constraints*

44. Mostow, J., *What is AI? And What Does It Have to Do with Software Engineering?*, forward to special issue on artificial intelligence and software engineering, IEEE Trans. on Software Engineering, v SE-11, n11 (Nov. 1985), pp. 1253-1256

45. Nagl, M, *Graph Technology Applied to a Software Project*, in *The Book of L*, ed. Rozenberg and Salomaa, Springer-Verlag 1985, pp. 303-322

46. Nam, Y., and Henschen, L. J., *Compiling Linear Recursive Programs with List Structure in Prolog into Procedural Languages*, Proc. COMPSAC90, Chicago, Ill., Nov. 1990

47. Naqvi, Shamim, and Tsur, Shalom, *A Logical Language for Data and Knowledge Bases*, Computer Science Press, 1989

48. Naqvi, Shamim, *Stratification as a Design Principle in Logic Programming*, MCC Technical Report ACA-ST-247-88

49. Newton, Peter, and Browne, James C., *The CODE 2.0 Graphical Parallel Programming Language*, Proc. ACM International Conf. on Supercomputing, July, 1992

50. Nicholson, Tim, and Foo, Norman, *A Denotational Semantics for Prolog*, ACM TOPLAS, v11 n4, pp. 650-665

51. Nikhil, Arvind and Rishiyur S., and Pingali, Keshav K., *I-Structures: Data Structures for Parallel Computing*, ACM TOPLAS, v11, n4, October, 1989

www.manaraa.com

52. O'Keefe, Richard A., *The Craft of Prolog*, MIT Press, 1990

53. Pagan, Frank G., *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, 1981

54. Pratt, Terrence W., *Formal Specification of Software using H-graph Semantics*, in *Graph Grammars and Their Application to Computer Science*, ed. Goos & Hartmanis, Springer-Verlag, 1982

55. Pratt, Terrence W., *H-Graph Semantics*, DAMACS Reports 81-15 and 81-16, University of Virginia, Charlottesville, VA 22901

56. Przymusinski, Teodor C., *Non-monotonic Reasoning vs Logic Programming: A New Perspective*, in *The Foundations of Artificial Intelligence*, ed. Partridge and Wilks, Cambridge University Press, 1990

57. Przymusinski, Teodor C., *On the Declarative Semantics of Deductive Databases and Logic Programs*, in *The Foundations of Deductive Databases and Logic Programming*, ed. J. Minker, Morgan Kaufmann, 1988

58. Ray, Patrick, *Program in Galois*, Master's Thesis, University of Texas at Austin, forthcoming

59. Setliff, Dorothy, *On the Automatic Selection of Data Structures and Algorithms*, in [38]

60. Smith, Douglas R., *KIDS — A Knowledge-Based Software Development System*, in [38]

61. Sridharan, K., Browne, J. C., and Newton, P., *An Environment for Parallel Structuring of Fortran Programs*, International Conf. on Parallel Processing, 1989

62. Stark, W. Richard, *LISP, Lore, and Logic: An Algebraic View of LISP Programming, Foundations, and Applications*, Springer-Verlag, 1990

63. Subrahmanyam, P. A., and Jia-Huai, You, *FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming*, in *Logic Programming: Functions, Relations, and Equations*, ed. DeGroot, Doug, and Lindstrom, Gary

www.manaraa.com

64. Van Hentenryck, Pascal, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989

65. Van Roy, P. and Despain, A. M., *High-Performance Logic Programming with the Aquarius Prolog Compiler*, IEEE Computer, January, 1992

66. Warren, D. H. D., *Implementing Prolog — Compiling Predicate Logic Programs*, Dept. of AI, Univ. of Edinburgh, Scotland, Res. Reps. 39 & 40

67. Waters, R. C. , *The Programmer's Apprentice: A Session with KBEmacs*, IEEE Trans. on Software Engineering, v SE-11, n11 (Nov. 1985), pp. 1296-1320

www.manaraa.com

VITA

William Russell Turpin was born in Wiesbaden, Germany, on May 7, 1958, and was soon thereafter adopted by Dr. and Mrs. William R. Turpin. He was four when his family moved to Austin, where he was naturalized an American citizen. He graduated from the (now defunct) high school department of Schreiner College, entered the University of Texas at Austin, in September, 1974, and graduated with a Bachelor of Arts degree from the liberal arts honors program (Plan II) in August, 1977. For the next year and a half, he studied mathematics at the graduate level at the University of Texas and at the University of Michigan in Ann Arbor.

From 1976, he worked as a computer programmer, and on returning to Austin in 1981, he joined Scientific and Engineering Software (then, Information Research Associates), where he has worked since, most recently as software architect, manager, and research scientist. In 1982, he began the formal study of computer science, again as a graduate student at the University of Texas. He was awarded a Master of Science degree in computer science in 1986. In 1991, the Texas Advanced Research Program funded a grant stemming from his research work. He has published refereed papers in distributed concensus algorithms and hardware-software co-design.

Addresses:	4032 S. Lamar #500-138, Austin, Texas, 78704	(U.S. mail)
	turpin@ses.com turpin@cs.utexas.edu	(Internet)

This dissertation was prepared by the author, using FrameMaker™ on Macintosh computers.

www.manaraa.com