On Two Different Notions of Type*

Richard Zippel

TR 90-1078
January 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, ONR Grant N00014-89-J-1946, and NSF Grant DMC-86-17355.

On Two Different Notions of Type

Richard Zippel
Cornell University
Ithaca, NY 14853

December 1, 1989

1 Introduction

Programming languages designed to deal with the structures of modern algebra require a substantially more complex type system than languages designed for many other tasks. This complexity is due partially to the need to create new classes of expressions on the fly (polynomials in x over the integers, matrices of polynomials in x, and so on) and partially to the need to directly manipulate and extract information from the domains from which these expressions come (the ring of polynomials in x over the integers, etc.). We have recently developed a package for manipulating algebraic expressions called *Weyl* [10], where one of the primary design goals is that the objects that naturally arise when describing algebraic algorithms should be "first class citizens." This leads naturally to the key difference between Weyl and other algebraic systems [6, 1, 3]: the division of the concept of "type" into the distinct concepts of *structure type* and *domain type* and the participation in of domains in the computations.

The simplest example of this distinction is the (rational) integers. 7 is an *integer* but it is an element of the *ring of rational integers*, $\mathbf{Z}$. It is important to notice that we said the *ring* of rational integers and not the *set* of rational integers. We say that 7's *structure type* (or *structure* for short) is `integer`,[1] but that its *domain type* (or *domain* for short) is $\mathbf{Z}$. In many uses of the integers it is not necessary to distinguish objects which are represented as `integer`'s and elements of $\mathbf{Z}$. All elements of $\mathbf{Z}$ are represented as `integer`'s and all objects whose representation is that of an integer are elements of *the integers*.

However, this is not the case if dimensions are introduced. One collection of objects whose structure is `integer` might be interpreted as inches, while another set of `integer`'s would be interpreted as meters. Denote the first set by A and the second by B. First, these sets form an abelian group, not a ring, so multiplying two elements of A will not yield an element of A. Only addition and subtraction are closed operations. Second, even though it is syntactically possible to add an element of A to an element of B, since both elements have (structure) type `integer`, the result would not be meaningful.

[1] We use this typewriter style font to distinguish a *structure type* from a mathematical object of the same name. Thus the `integers` is the set of all objects in a machine which are represented as integer data structures, while the *integers* indicates $\mathbf{Z}$.

In algebra we have found that the introduction of the domain concept accurately represents the way we think about algebraic algorithms. Those issues that deal with object representation or manipulation of the representation are part of the structure type. Those issues that relate to the mathematical characteristics of the set to which an object belongs are part of the domain. In Section 2 we give a few more detailed examples to illustrate this. We give a more formal discussion of our type system in Section 3. Section 4 discusses how Weyl's type system maps onto the Common Lisp Object System, and Section 5 illustrates how the domains participate in computations. Finally, we present some conclusions.

2 Examples

First we introduce some common mathematical notation. We denote the ring of rational integers by $\mathbf{Z}$ and the field of real numbers by $\mathbf{R}$. A *ring* is defined as a triple $(R, +, \times)$ where R is a set, and $+$ and $\times$ are binary maps from $R \times R$ to R that obey certain axioms. The ring of polynomials in $x_1, \ldots, x_n$ whose coefficients are elements of $\mathbf{Z}$ is denoted by $\mathbf{Z}[x_1, \ldots, x_n]$.

Consider the two dimensional surface of a sphere, which we denote by S^2. We would like to construct a cartesian coordinate system that covers the surface of the sphere. This is impossible with a single coordinate system but for a neighborhood of any point on the sphere we can can produce a coordinate system by using the stereographic projection. This give S^2 the structure of a 2 dimensional *locally Euclidean space* [8]. That is, around every point α in S^2 we have an open set $U_\alpha \subset S^2$ and a (C^k) map φ_α from U_α to $\mathbf{R}^2$. The pair $(U_\alpha, \varphi_\alpha)$ is called a *coordinate system*. We further require that coordinate systems be consistent where they overlap. That is, $\varphi_\alpha \circ \varphi_\beta^{-1}$ is also C^k where the function is defined.

Using just two coordinate systems we have the following spaces and maps:

$$\begin{array}{ccc} S^2 & \xrightarrow{\varphi_\alpha} & \mathbf{R}^2 \\ \big\downarrow{\scriptstyle \varphi_\beta} & & \\ \mathbf{R}^2 & & \end{array}$$

Let s be an element of S^2 and denote let $s_\alpha = \varphi_\alpha(s)$and $s_\beta = \varphi_\beta(s)$. At first blush the types of s_α and s_β would seem to be identical. They are both in $\mathbf{R}^2$ so the operations that work with one also work with the other. The elements of $\varphi_\alpha(U_\alpha)$ and $\varphi_\beta(U_\beta)$ could both be implemented as pairs of real numbers and all the operations that work in one case also work in the other. However, there is a subtle difference.

Consider a binary operation on $\mathbf{R}^2$; for instance, the Euclidean distance $\operatorname{dist}(p_1, p_2)$. This function is well defined for pairs of elements of $\mathbf{R}^2$. The definition that works when both arguments are elements of either $\varphi_\alpha(U_\alpha)$ or $\varphi_\beta(U_\beta)$ is syntactically correct for s_α and s_β, but the result is nonsensical. We claim that it should be an error to try to compute $\operatorname{dist}(\varphi_\alpha(s), \varphi_\beta(s))$, where $s \in U_\alpha \cap U_\beta$. Thus the "types" of $\varphi_\alpha(s) = s_\alpha$ and $\varphi_\beta(s) = s_\beta$ should be different.

Somehow we must distinguish the "type" of s_α, which is an element of the coordinate system $(U_\alpha, \varphi_\alpha)$, from that of s_β which is associated with (U_β, φ_β). A very natural way to interpret this is to say that the two instances of $\mathbf{R}^2$ shown in the diagram above are actually distinct sets, both of which are isomorphic to $\mathbf{R}^2$. We indicate this by using subscripts:

$$\begin{tikzcd} S^2 \arrow[r, "\varphi_\alpha"] \arrow[d, "\varphi_\beta"] & \mathbf{R}^2_\alpha \\ \mathbf{R}^2_\beta & \end{tikzcd}$$

There are two classes of properties of s_α that we care about: the details of its representation and *how* algorithms are to operate on s_α, which we call its *structure type*, and the algebraic structure of which it is an element, which we call s_α's *domain type*. Collectively, these two aspects are called the *type* of an object.

One might suspect that the separation of these two notions of type is superfluous. We have already shown examples (dimensioned objects, coordinate systems) where objects with the same structure type but different domain types. The following paragraph gives an example of objects that have different structure types but the same domain type. This suggests the orthogonality of the two notions. In both of these cases, we would like to code algorithms where we fix one of the type aspects, but leave the other aspect free. In later sections we give some additional examples.

Polynomials can in implemented in a number of different ways. They could be implemented as linked lists of non-zero coefficients, as vectors of all coefficients, recursively in the variables, etc. Furthermore, no one representation is optimal for all computations. Thus we may have two polynomials, p_D and p_S, where p_D is represented using a dense structure type and p_S is represented using a sparse structure type and yet both of the these polynomials are elements of the polynomial ring R. This domain is not just the intersection of the types of p_D and p_S, but is a particular mathematical object whose structure determines which operations can be applied to its elements and what their semantics is. For instance, whether it is an integral domain or not might indicate which algorithm is used to invert matrices over R. This is very similar to the approach used in Capsules [9].

3 Weyl's Type System

What we call the *structure type* of an object is almost exactly that which is called its *type* in other treatments. The concept of *domain type* includes other aspects of an object that are not covered by the usual concept of type. Intuitively, the structure type encapsulates information about how the object is put together, its syntax, while the domain type captures the intended semantics of the object. We argue that for the sake of naturalness the domain type and structure type of an object should be orthogonal concepts.

We define the *type* of an object to be the pair of a structure type and a domain type. For instance the rational integer 7 has structure type `integer` and domain type $\mathbf{Z}$. Thus it has type $\langle\!\langle \texttt{integer}, \mathbf{Z} \rangle\!\rangle$. For clarity we use Greek letters to denote structure types, and capital Roman letters to denote domain types. The only exception is when the domain has some other standard representation, such as $\mathbf{Z}$.

Identifier	::=	string of alphabetic characters
BasicType	::=	`integer` | `float` | ...
StructClass	::=	`record`([*Identifier*: *Type*]*) `using` ([*StructClass*]*)
StructType	::=	*Basic Type* | *StructClass*

Figure 1: Structure Types

Domain types and structure types are distinguishable and one cannot be used in the place of another. That is, there cannot exist two objects, A and B, such that the structure type of A is the domain type of B. The domain type of an object is often called a *domain*.

3.1 Structure Types

Structure types form a fairly conventional multiple inheritance data type system. There is a set of primitive types, *e.g.* `integer` and `float`. New types may be created by defining records consisting of existing types and by inheriting record structures from different types. This is indicated in Figure 1. For instance, the structure type of a complex number might be:

```
record (RealPart: float, ImagPart: float) using ()
```

Inheritance is indicated by including *StructClass*'s after the keyword `using`. The *slot identifiers*, in this case `RealPart` and `ImagPart`, are assumed to be distinct and it is an error for any of the slot identifiers of the included StructClass's to conflict. Notice that the we associate a full *type* with a slot not a *structure type*.

A instance of a structure type has a slot for each named slot in its record definition as well as the union of the slots of the *StructClass*'s from which it inherits. There are a variety of different ways to define the *subtype* relationship between two structure types A and B. At one end of the spectrum, we say that A is a subtype of B if an instance of structure type A includes all the slots of an instance of structure type B. At the other end, we say that A is a subtype of B if A inherits from B. This is a more restrictive definition of subtyping than the first. In between these two extremes lie numerous other definitions. Here we shall use only the most restrictive sense and we write $A \leq B$ to indicate that A is a subtype of B.

3.2 Domain Types

Domain types are regular objects, which themselves have a structure type and a domain type. For instance, the structure classes `Abelian-Group` and `Ring-with-Unit` might be defined as in Figure 2. Notice that just as in mathematics, an `abelian-group` consists of a set of elements and an operation, which is written additively. The class `ring-with-unit` is similar except that in inherits some structure for `abelian-group`. Domains have domain

```
Abelian-Group:
  record (Elements:  Set(Type),
          PlusFun:   Type → Type)
    using ()
Ring-with-Unit:
  record (Unit:   Type,
          TimesFun:   Type → Type)
    using (Abelian-Group)
```

Figure 2: Definition of the Structure Type of a Ring

types themselves. For now, we have decided to posit a superdomain, which we denote by $\mathcal{SD}$, that is the domain of all domains.[2] For mathematical domains, it seems that the domain of a domain would be a category. Thus the domain of $\mathbf{Q}$ might be the category of fields. However, this introduces numerous other problems.

Functions can be typed as well as data objects. The function $\mathtt{minus}_{\mathbf{Z}}$, which computes the additive inverse of an element of $\mathbf{Z}$, has type

$$\langle\langle \mathtt{integer}, \mathbf{Z} \rangle\rangle \rightarrow \langle\langle \mathtt{integer}, \mathbf{Z} \rangle\rangle.$$

This indicates that $\mathtt{minus}_{\mathbf{Z}}$ deals only with elements of the domain $\mathbf{Z}$ and only those elements of $\mathbf{Z}$ that are represented as `integer`'s.

Not all functions pay attention to the domain of their arguments. For example, the Lisp function `minus` has type

$$\forall A. \langle\langle \mathtt{integer}, A \rangle\rangle \rightarrow \langle\langle \mathtt{integer}, A \rangle\rangle.$$

A is universally quantified because `minus` is a "syntactic" operation and pays no attention to the semantic interpretation of its arguments. Notice that the value returned by `minus` might not be semantically meaningful. For instance, A might be the domain of integers modulo a prime number p, $\mathbf{Z}/(p)$, where elements are represented by non-negative integers between 0 and $p-1$. In this case, the value of `minus` would be not be a legitimate value. Nonetheless, this universal quantification technique allows us to embed the type systems of most programming languages within ours.

A slightly more complex situation arises when we want to quantify over the structure type to indicate (*ad hoc*) polymorphic operations. Without type system, we use the domain type to indicate the semantic intent of the operation while leaving the syntactic aspects of the operation free.

For example, consider a function that uses repeated squaring to exponentiate a quantity. This function takes two arguments, first of which is an element of a set with a multiplication operation, and the second is an element of $\mathbf{Z}$, and returns an element of the set. It assumes

[2] This introduces a "Russell's Paradox," since the domain of $\mathcal{SD}$ cannot be $\mathcal{SD}$ itself. We ignore this problem for now.

that the set is contains a multiplication operation and that multiplication is closed and associative. In languages like ML [4], this function would be expressed as one with an additional argument of the multiplication function itself. Thus its type would be

$$\forall \alpha. (\alpha, \mathbf{Z}, \alpha \times \alpha \to \alpha) \to \alpha.$$

However, this misses a couple of issues. First, there is no indication that the functional argument is associative, only that it is closed. Second, we don't really want to include the third argument at all. We'd rather say something about α.

Using domains, we would say the type of exponentiation by repeated squaring is:

$$\forall \alpha. \forall G(G \text{ a semigroup}). (\langle\!\langle \alpha, G \rangle\!\rangle, \langle\!\langle \mathtt{integer}, \mathbf{Z} \rangle\!\rangle) \to \langle\!\langle \alpha, G \rangle\!\rangle.$$

We could indicate that G is a semigroup through the structure type of G or through the domain type of G. We choose to use the structure type. That is, G is a semigroup if its type is a subtype of $\langle\!\langle \mathtt{SemiGroup}, \mathfrak{A} \rangle\!\rangle$. Saying that G has structure type `SemiGroup` means that G consists of a set of elements and an operation `times`. In addition we assume that implicit in the structure structure type `SemiGroup` is the axioms that the operations must obey. This is why we use the stronger notion of subtypes in Section 3.1.

Taking the other approach, of indicated that G is a semigroup through the domain of G is attractive. This approach would say that the domain of G is the (mathematical) category of semigroups. However, this introduces other complications, that we would like to avoid at this point.

With these assumptions, the type of our repeated squaring exponentiation function is

$$\forall \alpha. \forall G(\mathtt{type\text{-}of}(G) \leq \langle\!\langle \mathtt{SemiGroup}, \mathfrak{A} \rangle\!\rangle). (\langle\!\langle \alpha, G \rangle\!\rangle, \langle\!\langle \mathtt{integer}, \mathbf{Z} \rangle\!\rangle) \to \langle\!\langle \alpha, G \rangle\!\rangle.$$

This approach to polymorphism embeds the notion of *protocols*, as used in Capsules [9], in the structure type of the domain of a type. We have not yet used protocols in the operation selection and type construction process as outlined in the Capsule work because of the complications of multiple argument method selection. Recently, Wadler and Blott [7] have introduced the related notion of *type classes*, which is a simplification of the protocol idea.

4 Implementation of Weyl's Typing System

Weyl is implemented as an extension of Common Lisp [5] so that all of Common Lisp's control structures and tools are available. Many of the standard Lisp functions are overloaded more than is already done in Common Lisp. The Common Lisp Object System [2] forms the basis of Weyl's type system. It is important to remember that Common Lisp is a weakly typed language where an object's type can be determined at run time. Furthermore, current implementations of the Common Lisp Object System (CLOS) delay deciding which method to run for a particular generic function until run time when they have full information about the generic functions arguments. This allows us to experiment with the semantics of Weyl a bit more freely than is possible in a strongly typed language.

The objects manipulated by Weyl are either *domains* (domain types) or *domain elements* (objects with both structure types and domain types). Examples of domains are the rational integers ($\mathbf{Z}$), polynomial rings ($\mathbf{Q}[x, y]$, $\mathbf{Z}_p[t]$) and vector spaces ($\mathbf{R}^3$, $P^3(\mathbf{Z})$). The elements of these domains are domain elements. Each domain element is the element of a single domain and this domain can be determined at run time from the element. Unlike some other languages, we do not use the structure types directly.

In the sphere example given in Section 2, the surface of the sphere, S^2 is a domain, while the points in S^2 are domain elements. $\mathbf{R}^2_\alpha$ and $\mathbf{R}^2_\beta$ are both domains and s_α and s_β are domain elements. The domains of s_α and s_β are different. Nonetheless, the structure types of s_α and s_β are identical—in both cases it is `Real` $\times$ `Real`. Thus the type of s_α is $\left\langle\!\left\langle \texttt{Real} \times \texttt{Real}, \mathbf{R}^2_\alpha \right\rangle\!\right\rangle$.

Structure types are implemented as CLOS classes and the CLOS inheritance mechanisms are used for structure type inheritance. Thus both domains and domain elements are implemented as instances of CLOS classes. The class hierarchy used when constructing domains includes classes corresponding to groups, rings, fields and many other familiar algebraic structures.

This approach allows us to provide information about domains that is often difficult to indicate if we could only specify information about the type of the domain's elements. For instance, in addition to the class `Ring` we also provide classes like `integral-domain`.[3] Whether a domain is an integral domain or just a ring does not change how the domain's elements are represented or the operations that can be performed on them. What it may affect is the algorithms that are used to implement the operations. In a less mathematical context, one might consider the difference between binary trees and balanced binary trees.

In addition we provide information about the permissible operations involving elements of the domain. For instance, a `group` must have a `times` operation that can be applied to pairs of its elements. The result will be an element of the abelian group. This parallels the mathematical definition of a group where a group is a pair $(G, \times)$ consisting of a set of elements (G) and a binary operation ($\times$) that maps pairs of elements of G into G. Contrast this to the typical definition of a type as a set of objects that obeys a predicate.

We attach information about the permissible operations on elements of a domain to the domain itself. This allows us to ensure that when an instance of a domain is instantiated, the appropriate operations have been provided. This is especially valuable in an incremental development environment like those common for Lisp.

Domain elements are also implemented as instances of CLOS classes and their structure type is their CLOS class. In addition a slot is provided in each domain element for its domain. There is no single object that represents an object's complete type. The structure type of an object can be obtained using the function `structure-of` and the domain type by `domain-of`.

Figure 3 illustrates the relationship between domains, domain elements and CLOS classes. We have a polynomial $x + 1_{\mathbf{Z}[x]}$, which has structure type (is an instance of the CLOS class) `polynomial`. Its domain, $\mathbf{Z}[x]$, has structure type `polynomial-ring`, which

[3]An integral domain R is ring that does not have any zero divisors. That is, for all $a, b \in R$, if $ab = 0$ then either a or b is zero.

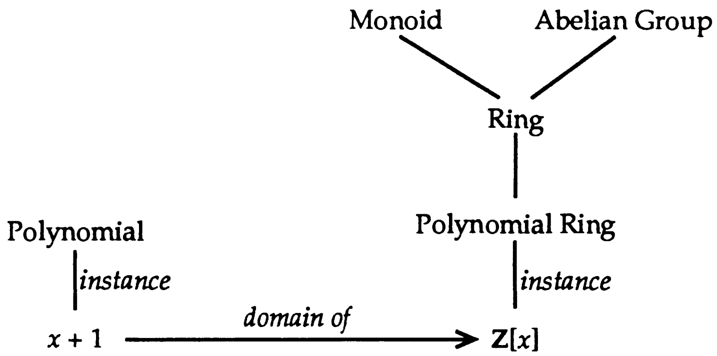

Figure 3: Domain Example

is a subtype of `Ring`, `monoid` and `abelian-group`. The polynomial $x + 1_{\mathbf{Z}[x]}$ is an element of the domain $\mathbf{Z}[x]$. Notice that there is a complete type hierarchy sitting above the class `polynomial-ring`.

We might choose to implement finite algebraic extensions using polynomials. This would be done as follows: We would create a new class of domains, `algebraic-extension`, that inherits from the `ring` class, but also includes a slot for the minimal polynomial of the extension. The elements of this ring would be instances of the `algebraic-object` class which inherits from `polynomial`. The `plus` operation on `algebraic-objects` would be the one inherited from `polynomial`, while we would define a new routine for multiplying algebraic numbers. This routine would first check its two arguments to make sure they are elements of the same domain, signaling an error if this is not the case. It would then multiply its arguments as if they were polynomials and finally compute the remainder modulo the minimal polynomial of the algebraic extension which it would get from the domain. This example illustrates that the structure type class hierarchy need not match that of the domain class hierarchy.

5 Usage of Domains

Unlike structure types, domains are actively used in algebraic computations. There are numerous operations that create new domains from old domains and that extract information from domains. Among the simplest is `make-polynomial-ring`, which takes a ring, R, and a set of variables $\{x_1, \ldots, x_n\}$, and returns a domain of polynomials, $R[x_1, \ldots, x_n]$. `Make-quotient-field` takes a domain and returns a domain that is the quotient field of its argument. Thus `(make-quotient-field Z)` returns the rational numbers $\mathbf{Q}$.

Domains are also a convenient way of communicating information between modules. Consider a zero finding routine that obtains the exact zeroes of a polynomial. The polynomial it is passed, $p(x)$, would have to be an element of the domain $R[x]$, where R is some field. Ideally the zeroes of $p(x)$ would be elements of R, but in general they lie in an algebraic extension of R. For instance, if asked to find the zeroes of $p(x) = x^2 - 2$ it would find the domain of p to be $\mathbf{Q}[x]$. The zeroes of p however, are in $\mathbf{Q}[\sqrt{2}]$, an algebraic extension of $\mathbf{Q}$. In this case, the zero finder would return two zeroes: $\{\sqrt{2}, -\sqrt{2}\}$, which are elements of an extension domain of $\mathbf{Q}$, which was implicitly passed to the root finder as part of p.

Domains need not be static. This is an important feature that allows us to deal with algebraic closures of fields. Algebraic numbers are represented as a polynomial in a primitive element. The field $\mathbf{Q}[\sqrt{2}, \sqrt[3]{5}]$ is of degree 6 and its elements are represented as polynomials in a primitive element of degree 6. This approach doesn't directly work with algebraic closures, because algebraic closures are usually of infinite degree. However, in the course of a (finite) computation only a finite number of algebraic elements are introduced and their common primitive element will be of finite degree. Thus expressions in algebraic closures are represented as polynomials in a primitive element, but the degree of the primitive element is continually increased as new, algebraically independent, quantities are introduced.

6 Conclusions

We believe that the conventional notion of a type mixes two distinct concepts which we are have distinguished in Weyl by the concepts of *structure type* and *domain type*. These distinctions are most apparent when building systems that deal with algebraic objects like polynomials, rings and ideals. In this paper we have presented a number of examples of situations in which this distinction arises and it useful.

Many of the ideas in this system come from discussions with Barry Trager and Dick Jenks who were trying to solve similar problems for the Scratchpad system. This paper was originally motivated by an attempt to describe Weyl's organization to Brian Smith and Gregor Kiczales at Xerox Parc. We have has also greatly benefited from discussions with Georges Lauri, Prakash Panangadan and Doug Howe.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014–86–K–0591, the Army Mathematical Sciences Institute at Cornell under contract U03–8300, the National Science Foundation through contract DMC–86–17355 and the Office of Naval Research through contract N00014–86–K–0281.

References

[1] S. K. Abdali, G. W. Cherry, and Niel Soiffer. An object oriented approach to algebra system design. In Bruce Char, editor, *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*. ACM, 1986.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Kenneth Kahn, Sonya E. Keene, Gregor Kiczales, Larry Masinter, David A. Moon, Mark Stefik, and Daniel L. Weinreb. Common Lisp object system specification. Technical Report 88–002, X3J13, ANSI Common Lisp Standardization Committee, July 1988.

[3] Barbara Liskov, Alan Snyder, Russ Atkinson, and Craig Shaffert. Abstraction mechanisms in clu. *Communications of the ACM*, 20(8):564–576, August 1977.

[4] Robin Milner. A proposal for Standard ML. In *Proceedings of the Symposium on LISP and Functional Programming*, pages 184–197, New York, NY, 1984. ACM. Austin, TX, Aug. 6–8.

[5] Guy Lewis Steele Jr. *Common Lisp, The Language.* Digital Press, Burlington, MA, 1984.

[6] R. S. Sutor and Richard D. Jenks. The type inference and coercion facilities in the Scratchpad II interpreter. In *SIGPLAN '87: Symposium on Interpreters and Interpretive Techniques*, pages 56–63, New York, NY, 1987. ACM.

[7] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc.* In *Proceedings of 16th Conference on the Principles of Programming Languages*, pages 60–76. ACM Press, January 1989.

[8] F. W. Warner. *Foundations of Differentiable Manifolds and Lie Groups.* Springer-Verlag, New York, NY, 1983.

[9] Richard E. Zippel. Capsules. *SIGPLAN Bulletin*, 18:166–169, October 1983.

[10] Richard E. Zippel. The Weyl computer algebra substrate. Technical report, Department of Computer Science, Cornell University, Ithaca, NY, 1989.