Types as Intervals

Robert Cartwright†

Computer Science Department
Rice University
Houston, TX 77251

Abstract

To accommodate polymorphic data types and operations, several computer scientists—most notably MacQueen, Plotkin, and Sethi—have proposed formalizing types as *ideals*. Although this approach is intuitively appealing, the resulting type system is both complex and restrictive because the type constructor that creates function types is not monotonic, and hence not computable. As a result, types cannot be treated as data values, precluding the formalization of type constructors and polymorphic program modules (where types are values) as higher order computable functions. Moreover, recursive definitions of new types do not necessarily have solutions.

This paper proposes a new formulation of types—called *intervals*—that subsumes the theory of types as ideals, yet avoids the pathologies caused by non-monotonic type constructors. In particular, the set of interval types contains the set of ideal types as a proper subset and all of the primitive type operations on intervals are extensions of the corresponding operations on ideals. Nevertheless, all of the primitive interval type constructors including the function type constructor and type quantifiers are computable operations. Consequently, types are higher order data values that can be freely manipulated within programs.

The key idea underlying the formalization of types as intervals is that *negative* information should be included in the description of a type. Negative information identifies the finite elements that do not belong to a type, just as conventional, positive information identifies the elements that do. Unless the negative information in a type description is the exact complement of the positive information, the description is partial in the sense that it approximates many different types—an interval of ideals between the positive information and the complement of the negative information. Although programmers typically deal with total (maximal) types, partial types appear to be an essential feature of a comprehensive polymorphic type system that accommodates types as data, just as partial functions are essential in any universal programming language.

1. Introduction

One of the major unresolved questions in programming language design is how to define the notion of *data type*. This paper focuses on type systems for *abstract* programming languages (*e.g.*, SETL, ML) which emphasize mathematical elegance and expressive power rather than execution efficiency. The justification for this focus is twofold. First, it is important to understand what type systems are mathematically possible, regardless of their impact on execution efficiency. Second, abstract programming languages are steadily growing in importance as tools for program specification, prototyping, and implementation. In many contexts—particularly program specification and prototyping—execution is much less important than simplicity and elegance.

The critical feature that distinguishes abstract programming languages from conventional ones is that data values are treated exclusively as *abstract* objects; their underlying representation within a computer is completely hidden from the programmer. In this limited context, it is much easier to identify and compare possible type systems because it avoids the difficult question of whether types refer to abstract data values or their representations. In fact, nearly all of the type systems proposed for abstract programming languages (*e.g.*, [Scot76], [ADJ77], [Gutt78], [Cart80], [MacQ82]) share the basic intuition that a type identifies a meaningful subset of the program data domain. The principal issue on which they differ is the question of *which* subsets of the data domain can be designated as types.

1.1. Partition vs. Predicate Types

There are two basic paradigms for subdividing a data domain into types: types as *partitions* (disjoint sets) and types as *predicates* (overlapping sets). In a partition type system, every data value belongs to a unique type. Most production programming languages (*e.g.*, Fortran, Pascal, C, Ada) embrace this point of view. In a predicate type system, on the other hand, a data value can belong to many different types; a type is simply a designated subset of the program data domain. Most interactive programming languages (*e.g.*, APL, LISP) subscribe to this approach.

Partition typing is justifiably popular because it is easy to understand, easy to implement, and supports "static" (translation-time) type-checking—an effective tool for finding program errors. Partition typing also facilitates the efficient implementation of data values and operations, because the representation for each type can be optimized independently of the representations for other types. The major weakness of this approach is the severe limitation it imposes on the variety of possible types. For this reason, the domains (sets of intended inputs) of most program operations *cannot* be captured by type declarations. In partition type systems, many run-time errors such as division by zero are not classified as type errors. Consequently, a "type-correct" program can still generate errors at run-time.

In contrast, predicate typing allows the domain of every program function to be declared as a type. "Type-correctness" in this discipline is a much stronger property than it is in the partition type discipline, because a predicate-typed program is type-correct iff it *cannot* generate a run-time error. The major disadvantage of this approach is that verifying the type-correctness of a program is an undecidable problem. Complete type-checking at translation time is impossible.

Nevertheless, there are valuable, less ambitious alternatives to complete type-checking that are feasible in predicate-typed languages. In fact, in a well-designed predicate type system (such as that in Typed LISP [Cart76a,76b]) it is straightforward to perform "coarse" type-checking that detects *exactly* the same errors as conventional "static" type-checking in the corresponding partition type system. In coarse type-checking [Cart76a,76b], every predicate type is associated with a coarse type that contains the predicate type. Each coarse type is the union of a finite collection

†This research was performed while the author was a visiting professor at Carnegie-Mellon University and has been partially supported by DARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-8-K-1539.

of disjoint atomic types.[1] Coarse type-checking ensures that the coarse type of every function argument list overlaps the declared coarse domain of the function. A program is "coarse-type-correct" if and only if this condition holds.

At first glance, coarse type-checking appears less stringent than conventional partition type-checking, because it does not preclude type errors during program execution. This conclusion, however, is erroneous, because the notions of type are different. It is easy to show that a predicate-typed program P is coarse-type-correct if and only if the semantically equivalent partition-typed program P' is type-correct. At every point in the program P, where the coarse type t of an argument α is not contained in the type u required by its context, the corresponding partition-typed program P' must apply an explicit type conversion function $\text{Convert}_{t\to u}$ to α to convert it from type t to type u (modifying the "tag" attached to the value). If the value is not convertible from type t to u, the conversion function Convert must generate a run-time error, even though the function application is "type-correct".

The primary advantage of predicate typing is that it enables programmers to document the intended behavior of program operations much more precisely than is possible within the rigid framework of partition type systems. This information can potentially be exploited by sophisticated heuristic type-checkers that detect far more program errors than conventional static type-checkers. In essence, heuristic type-checking is a restricted form of program verification in which all program assertions are type declarations. Much of technology developed for program verification systems such as fast simplification methods [Nels79] should be applicable to this problem.

1.2. The Impact of Polymorphism

If we expand our discussion to include the subject of polymorphic operations—functions that work for every member of a family of structurally similar types—the differences between partition typing and predicate typing become even more dramatic. In predicate-typed languages the primitive functions for manipulating composite objects such as sequences are naturally polymorphic. Program-defined functions that are constructed from these naturally polymorphic operations automatically inherit the polymorphic behavior. This property is one of the most attractive features of predicate type systems. In LISP, for example, the sequence operations car, cdr, cons, and null work for all sequences regardless of the element types involved. As a result, every LISP program constructed from these polymorphic operations is polymorphic as well; the library functions append, reverse, and last are simple examples of this phenomenon.

In contrast, partition-typed languages must include distinct operations for each member of a family of structurally similar types (such as sequences), precluding natural polymorphism. To support polymorphic operations, additional machinery is required. The standard solution is to explicitly pass types as parameters—a cumbersome convention for naturally polymorphic operations where no type information is necessary.

1.3. Research Objective

The critical design decision in formulating a coherent predicate type system is determining the class of definable predicates. If the class of definable predicates is too small, then the domains and ranges of many program operations will not be definable as types. On the other hand, if the class of definable predicates is too large or poorly constructed, then the collection of definable types will form an amorphous set—preventing types from being treated as data values and eliminating the possibility of heuristic type-checking.

[1]In a data domain where the universe of values is formalized as a free term algebra, it is natural to define an atomic type as the set of all terms with the same outermost constructor.

The primary objective of this paper is to develop a predicate type system suitable for any data domain D that accommodates a comprehensive set of predicate types, yet is computationally tractable. More specifically, the type system should satisfy the following requirements:

1. *Breadth*: the type system should be applicable to any data domain in the sense proposed by Scott [Scot83] (a countably-based, algebraic cpo) that is likely to arise in practice. In particular, the type system should accommodate higher order data values like functions and infinite trees (lazy data objects).
2. *Expressiveness*: the set of definable types should be rich enough that every program operation, including naturally polymorphic ones, can be precisely typed. Although a rigorous definition and investigation of this property is beyond the scope of this paper, the informal intent is that the type constraints required to guarantee the absence of run-time errors should be logically implied by appropriate type declarations for the operations defined in the program. The notion is analogous to the well-known *expressiveness* property for program assertion languages.
3. *Effectiveness*: the set of types should form a finitary domain on which all of the primitive type constructors are computable functions. This property guarantees that recursive definitions have computable least solutions and enables programs to manipulate types as data.

2. Previous Work

Among the predicate type disciplines discussed in the literature, the two that come closest to meeting this goal are types as retracts and types as ideals. Each system satisfies two of the three criteria enumerated above. The system of retracts is broad and effective, but not expressive; the system of ideals is broad and potentially expressive[2], but not effective. Both of these disciplines are rooted in Scott's *theory of domains* which formalizes data domains as countably-based, algebraic cpo's. Scott calls these structures *finitary domains*. The following overview of the two systems presumes some familiarity with domain theory, which is summarized at the beginning of Section 4.

2.1. Types as Retracts

In the theory of types as retracts [Scot76,81,83], every type t within a data domain D (a countably-based, algebraic cpo) is a subdomain of D: a subset of D that is generated by closing a set of finite elements of D under the least upper bound relation (with respect to D) on consistent subsets.[3] Each type t forms a finitary domain under the approximation ordering on D and conforms with the consistency, least upper bound, and finiteness relations on D. To accomodate functions and infinite trees as data values and to support interesting type definitions, the data domain D typically includes isomorphic images of its own function space $[D\to D]$, Cartesian product space $[D\times D]$, and coalesced sum space $[D+D]$. In most cases, these three subspaces are disjoint, but it is not technically necessary.

The theory of types as retracts has many important mathematical properties including the following:

1. The set of retracts over a finitary domain D forms a finitary domain $\mathbf{Ret}_D$. If D is effective, then so is $\mathbf{Ret}_D$.
2. The three basic operations $\{\to, \times, +\}$ for building composite types from simpler ones are computable functions on $\mathbf{Ret}_D$. In addition, all of the higher order operations used to define recursive types—in particular λ-notation (usually formalized as combinators) and the least fixed point operator μ—are computable.

[2]Depending on the mechanisms available for defining types.

[3]Scott has proposed two different formulations of retracts. See Section 4.1.

3. For each type t, there is a corresponding continuous function ρ_t (called a *projection*) on the data domain $\mathbf{D}$ that *coerces* an arbitrary data value to the "nearest" value within t. The fixed-point set of ρ_t is precisely t. If $\mathbf{D}$ is effective, the projection ρ_t is computable iff the finite elements of t are recursively enumerable.

Although the system of retracts obviously satisfies the goals of *breadth* and *effectiveness* enumerated in Section 1.4, it fails to meet the *expressiveness* criterion. Formalizing types as retracts precludes the precise typing of naturally polymorphic functions. In the theory of types as retracts, types are *coercions* that adversely affect the behavior of potentially polymorphic functions. Every function f of type A→B (where A and B are retracts) must yield outputs in B for all inputs regardless of whether or not they are in A. More precisely, f must satisfy the equation

$$f = \rho_A \circ f \circ \rho_B$$

where ρ_A and ρ_B are the projections (coercions) corresponding to A and B respectively. In informal terms, a function f belongs to type A→B only if it maps both legal (A) illegal inputs ($\bar{A}$) into legal outputs (B). Consequently, a function f that maps elements of type $\alpha(t)$ into elements of type $\beta(t)$ for every type t does not generally belong to type $\alpha(t) \rightarrow \beta(t)$ for every type t.

To help clarify the situation, let us consider two simple examples. First, assume we are given the identity function $\lambda x.x$ for an arbitary data domain $\mathbf{D}$. Although this function $\mathbf{D}$ clearly works as an identity function for any type (retract) t within $\mathbf{D}$, it does not belong to the type t→t for any type t except $t = \mathbf{D}$. To obtain an identity function of type t→t for type $t \subset \mathbf{D}$, we must coerce $\lambda x.x$ to $\rho_t \circ (\lambda x.x) \circ \rho_t = \rho_t$. Consequently, it is impossible to write a polymorphic identity function that has type t→t for every retract t.

As a more realistic example, assume that we are given a data domain $\mathbf{D}$ that includes a type (retract) SeqAny consisting of the set of all finite sequences over $\mathbf{D}$. Let Seq:$\mathbf{Ret}_D \rightarrow \mathbf{Ret}_D$ be the computable function that maps each type t in $\mathbf{D}$ to the type consisting of all finite sequences over t, and let Cat be the operation mapping SeqAny×SeqAny into SeqAny that concatenates sequences:

$$\mathrm{Cat}(<x_1,\ldots,x_m>,<y_1,\ldots,y_n>) = <x_1,\ldots,x_m,y_1,\ldots,y_n> .$$

The type SeqAny obviously contains the type Seq(t) for all types t, yet Cat does not belong to the type Seq(t)×Seq(t)→Seq(t) for any t other than the entire domain $\mathbf{D}$, because Cat maps illegal inputs (within SeqAny) to illegal outputs. If t excludes any element $d \in \mathbf{D}$, then $\mathrm{Cat}(<d>,<d>) = <d,d>$ does not belong to Seq(t), implying that Cat does not belong to Seq(t)×Seq(t)→Seq(t).

This anomaly is inherent in the formulation of types as retracts. It cannot be fixed by changing the definition of the function type constructor →. The set of continuous functions that map one retract into another does not necessarily form a retract unless at least one of two retracts is downward closed under the approximation ordering on the domain.

The only approach to polymorphism that appears compatible with formulating types as retracts is to pass types explicitly as parameters. In this scheme, the naturally polymorphic behavior of operations like the identity function $\lambda x.x$ and the sequence concatenation function Cat is ignored; every definition of a polymorphic function must include an abstractions with respect to the type of each polymorphic argument as well an abstraction with respect to the argument itself. Similarly, every application of a polymorphic function must include type arguments as well as conventional data arguments. This approach is explored in detail in [Reyn74] and [McCr79].

2.2. Types as Ideals

In contrast to the theory of types as retracts, the theory of types as ideals [MacQ82,MacQ84a] is specifically designed to exploit naturally polymorphic operations. Although the theory of ideals is cast in same mathematical framework as the theory of retracts, it is based on a different intuitive notion of type. In the theory of types as ideals, types are viewed as *constraints* rather than coercions. This change in viewpoint produces a profoundly different theory of types.

To prevent data objects from having mulitiple interpretations, the theory of ideals assumes that the data domain $\mathbf{D}$ is defined by a domain equation of the form

$$\mathbf{D} = [\mathbf{D} \rightarrow \mathbf{D}] + [\mathbf{D} \times \mathbf{D}] + [\mathbf{D} + \mathbf{D}] + \mathbf{A}_1 + \ldots + \mathbf{A}_n$$

where the equality symbol denotes isomorphism; the domain constructors $\{\rightarrow, +, \times\}$ have their usual meanings; and $\mathbf{A}_1,\ldots,\mathbf{A}_n$ denote type expressions constructed from the symbol $\mathbf{D}$, constant symbols denoting primitive domains (*e.g.*, the flat domain of natural numbers), and function symbols denoting continuous operations on domains. Although this assumption appears restrictive, it does not adversely affect the applicability of the theory, because any data domain of practical interest can easily be cast in this form.

The most visible difference between the theory of ideals and the theory of retracts is the definition of the set of types. As its name suggests, the theory of ideals designates the set of *ideals* over $\mathbf{D}$ (denoted $\mathbf{Idl}_D$) as the types of $\mathbf{D}$. An *ideal* of $\mathbf{D}$ is simply a downward-closed, directed-closed subset of $\mathbf{D}$. In other words, a type must be closed under both approximations and limits. In contrast, a retract is closed under least upper bounds and limits.

To support type definitions, the theory includes operations corresponding to all the standard type operations in the theory of retracts. With the exception of the function type constructor, the definitions of these operations are consistent with the corresponding operations on retracts. On the other hand, the function type constructor ⊃,[4] is specifically designed to accommodate polymorphism. It is defined by the equation

$$A \supset B = \{ f \in [\mathbf{D} \rightarrow \mathbf{D}] \mid \forall x \in A\ f(x) \in B \}$$

where A and B are types over $\mathbf{D}$ and $[\mathbf{D} \rightarrow \mathbf{D}]$ denotes the domain of continuous functions mapping $\mathbf{D}$ into $\mathbf{D}$. This definition of the function type constructor is incompatible with formalizing types as retracts: if A and B are retracts, $A \supset B$ is not necessarily a retract. One of the principal reasons for formalizing types are ideals is the fact that are ideals are closed under the polymorphic function type constructor ⊃, but retracts are not.

In addition to adopting the "polymorphic" version of function type construction, the theory includes four extra primitive operations to support polymorphism: the intersection and union operations $\{\cap, \cup\}$ from naive set theory and the type quantifiers ∀ and ∃ (which map functions on ideals into ideals) defined by the equations:

$$\forall(f) = \sqcap_{t \in \mathbf{Idl}_D} f(t); \quad \exists(f) = \sqcup_{t \in \mathbf{Idl}_D} f(t)$$

where ⊔ and ⊓ denote the least upper bound and greatest lower bound operations. The expressions ∀(f) and ∃(f) are usually written ∀t f(t) and ∃t f(t).

Using the theory of types as ideals, MacQueen, Plotkin, and Sethi have generalized the elegant approach to polymorphic type definition and inference developed by Milner [Miln78] for the programming language of ML and by Hindley [Hind69] for typing expressions in the lambda calculus. In this approach to polymorphism, a polymorphic operation is assigned a type that is the intersection (greatest lower bound) of many simpler types. For each application of the polymorphic operation, the appropriate type in the intersection is inferred as the relevant typing for that application. For example, the function Cat described in Section 2.1 belongs to the ideal type ∀t Seq(t)×Seq(t)→Seq(t).

[4]Since the function type constructor ⊃ on ideals is inconsistent with the usual → constructor on retracts, it is denoted by a different symbol.

Like the system of retracts, the system of ideals has many important mathematical properties including the following:

1. The set of ideals over a finitary domain **D** form a finitary domain $\mathbf{Idl_D}$. If **D** is effective, then so is $\mathbf{Idl_D}$.
2. With the exception of the function type constructor $\supset$, all of the basic operations $\{\times, +, \cap, \cup\}$ for building composite types from simpler ones are computable functions. However, none of the higher order operations for defining recursive and polymorphic types $\{\mu, \forall, \exists\}$ are computable, because their input spaces (which include non-monotonic functions) are not finitary.
3. For each type t, there is a corresponding continuous function ξ_t (called a *constraint*) mapping **D** into the trivial domain $\{\bot, \mathbf{true}\}$ that identifies the elements of **D** that *do not belong* to t. In particular, t satisfies the formula

   $\forall x \in \mathbf{D}\ [\xi_t(x) = \mathbf{true} \Leftrightarrow x \notin t]$.

   Unfortunately, the computable elements of $\mathbf{Idl_D}$ do not generally correspond to computable constraints.

The primary disadvantage of the theory of types as ideals is the fact that the function type constructor $\supset$ is not monotonic, much less computable. Hence, the theory fails to meet the goal of *effectiveness* stated in Section 1.4. This fact has three significant consequences.

First, since $\supset$ is an indispensable primitive operation on types, types cannot be treated as data values because expressions involving $\supset$ are not computable. As a result, type constructors (such as Seq in Section 2.1) and polymorphic program modules that take types as arguments[5] cannot be formalized as higher order computable functions.

Second, there is no general mechanism—such as the familiar Kleene least fixed point construction—for solving recursive type equations. The function corresponding to a system of recursion equations is not necessarily monotonic. Although MacQueen, Plotkin, and Sethi [MacQ84a] have established that unique solutions exist for an important syntactic class of recursive type equations (called *formally contractive* equations), the theory is restrictive; it is not applicable either to non-contractive systems of type equations or to more general systems of recursive type equations that include the definition of type constructors. For this reason, it is *syntactically* illegal to apply the fixed-point operator to type expressions that are not formally contractive.

Third, reasoning about ideal types is a complex problem that lies outside the scope of established deductive systems for data domains (such as Edinburgh LCF [Gord77] or the first order theory of domains described in [Cart82]). All of these systems presume that every function is continuous.

3. Types as Intervals

The principal research contribution of this paper is the construction of a new theory of types—called *types as intervals*—that satisfies the three goals enunciated in section 1.4. The new theory of types is closely related to the theory of types as ideals, because it is based on exactly the same intuitive notion of type and approach to polymorphism. In fact, the set of interval types over a domain **D** forms a superset of the set of ideal types over **D** and the type operations on intervals are extensions of the corresponding operations on ideals.

The motivation for formalizing types as intervals instead of ideals comes from the following observation. In the theory of types as ideals, the description of a computable type A specifies how to enumerate the set of finite elements of **D** that belong to A, but it does not specify how to enumerate the set of finite elements that do not belong to A—even though this set is recursively enumerable in almost all cases of practical interest. The omitted information is important; if it were available, the function type construction $A \supset B$ would be computable, because all of the "one-step" functions[6] $a \mapsto b$ in $A \to B$ where $a \notin A$ (vacuously satisfying the membership test for $A \supset B$) would be recursively enumerable, as well as those where $a \in A$ and $b \in B$. Without it, the one-step functions that vacuously belong to $A \supset B$ cannot be enumerated.

The theory of types as intervals is specifically designed to overcome this problem. The essential difference between the theory of types as ideals and theory of types as intervals is that interval type descriptions contain *negative* information specifying the elements that do *not* belong to a type as well *positive* information specifying the elements that do belong. The interval type corresponding to an ideal type A includes both a description of A and a description of the complement of A.

The addition of negative information to type descriptions has three major consequences. First, it forces the inclusion of "partial" elements in the space of types. These elements do not have any analogs in the system of types as ideals. If the negative information in an interval type description is not the exact complement of the positive information, the description is partial in the sense that it describes an interval of ideals between the positive information and the complement of the negative information. Although the total (maximal) types are the types of immediate practical importance, the partial types are required to make the set of intervals form a finitary domain under the approximation ordering determined by inclusion of information.

Second, the approximation ordering on interval types does not agree with the approximation ordering on ideals. In the theory of types as ideals, type A approximates type B if and only A is a subset of B. In the theory of types as intervals, the interval corresponding to the ideal A is completely unrelated to the interval corresponding to B unless A and B are identical.

Third, all of the standard type operations on ideals have natural extensions to the space of intervals which are *computable*—even though the function type constructor and higher order type operations on ideals are *not computable*. The inclusion of additional information in type descriptions is responsible for this apparent paradox.

3.1. Definition of Interval Types

There are two different ways to construct the domain of interval types. The two constructions complement each other: one has a simple, intuitive explanation; the other reveals the computational structure of interval types. In the simple construction, an interval type over a finitary domain **D** is defined as the set $[a,A]$ of all ideals over **D** that lie between two designated ideals a and A (inclusive) where $a \subseteq A$. The approximation ordering on intervals is simply the superset relation on sets: $[a,A] \sqsubseteq [b,B] \Leftrightarrow [a,A] \supseteq [b,B]$. The total (maximal) elements in the set of intervals over a domain are intervals of the form $[A,A]$ that contain a single ideal A.

In the computationally concrete construction, an interval type is defined as a pair of sets $<a,\bar{A}>$ where a is an ideal over **D** and $\bar{A}$ is a co-ideal (complement of an ideal) over **D** that does not intersect a. The approximation ordering in this formulation of intervals is the conjunction of the subset relations on corresponding components: $<a,\bar{A}> \sqsubseteq <b,\bar{B}> \Leftrightarrow a \subseteq b \wedge \bar{A} \subseteq \bar{B}$. Similarly, the total elements are pairs of the form $<A,\bar{A}>$ where A is an ideal and $\bar{A}$ is is complement.

The major advantage of the second construction is that it makes the finite elements of the domain manifest. They are simply pairs of the form $<b,\bar{B}>$ where b is a finite element in the space of ideals and $\bar{B}$ is a finite element in the space of co-ideals over **D**. For readers that are familiar with the Scott topology, an interval type is simply a pair consisting of a Scott-closed and a Scott-open set that do not intersect. It is straightforward to prove that both the set of ideals (Scott-closed sets) over a finitary

[5] As defined by Burstall and Lampson [Burs84].

[6] The one-step function $a \mapsto b \in A \to B$ is the least function $f \in A \to B$ such that $b \sqsubseteq f(a)$.

domain $\mathbf{D}$ and the set of co-ideals (Scott-open) sets over a finitary domain form finitary domains under the subset ordering.

3.2. Standard Operations on Intervals

The theory of types as intervals includes operations corresponding to all of the operations in the theory of types as interals. All of basic (first order) type operations on intervals are defined so that their retrictions to total intervals are identical to the corresponding operations on ideals. They can be defined in terms of the corresponding operations on ideals as follows:

$[a, A] \cap [b, B] = [a \cap b, A \cap B]$ $\quad [a, A] \times [b, B] = [a \times b, A \times B]$
$[a, A] \cup [b, B] = [a \cup b, A \cup B]$ $\quad [a, A] + [b, B] = [a+b, A+B]$
$[a, A] \supset [b, B] = [A \supset b, a \supset B]$

Similarly, the higher order operations on intervals $\{\forall \exists, \mu\}$ are generalizations of the corresponding operations on ideals, assuming that we identify continuous totality-preserving functions on intervals (functions that map total intervals to total intervals) with the corresponding functions on ideals (which are not necessarily continuous). For this reason, the parameter in an interval type quantification ranges only over total intervals. In the theory of intervals, the type quantifiers are defined by the equations

$$\exists f = < \sqcup_{x \in \mathrm{Type_D}^\dagger} f(x)^+ , \sqcap_{x \in \mathrm{Type_D}^\dagger} f(x)^- >$$
$$\forall f = < \sqcap_{x \in \mathrm{Type_D}^\dagger} f(x)^+ , \sqcup_{x \in \mathrm{Type_D}^\dagger} f(x)^- >$$

where $\mathrm{Type_D}^\dagger$ denotes the set of *total* intervals over the data domain $\mathbf{D}$ and f^+ and f^- denote the component functions defined by the equation

$$f(x) = <f^+(x), f^-(x)> .$$

In contrast, the least fixed-point operator μ is simply the standard **Y** operator from domain theory.

3.3. Important Properties of Interval Types

The most important mathematical properties of interval types are summarized in the following list:

1. The set of intervals over the finitary domain $\mathbf{D}$ forms a finitary domain $\mathbf{Type_D}$ under the superset ordering relation. The total elements of $\mathbf{Type_D}$ are all intervals of the form $[A,A]$ where A is an arbitrary ideal over $\mathbf{D}$. Hence, there is a natural one-to-one correspondence between the maximal elements of $\mathbf{Type_D}$ and the ideals of $\mathbf{D}$.
2. All of the standard operations for building composite types from simpler ones including the type quantifiers $\forall$ and $\exists$ are computable functions. Moreover, if we identify the total intervals with the corresponding ideals and functions on intervals that preserve totality with function on ideals, all of the type operations on $\mathrm{Type_D}$—including the higher order operations $\{\forall, \exists, \mu\}$—are simply extensions of the corresponding type operations on ideals to a larger space of types (with a different approximation ordering). Consequently, every type definition and type inference in theory of ideals has an immediate analog in the space of total intervals.
3. For each interval type $\alpha = [a,A]$, there are two corresponding continuous functions $\rho_\alpha:\mathbf{D} \rightarrow \mathbf{D}$ and $\xi_\alpha:\mathbf{D} \rightarrow \{\bot, \mathbf{true}\}$ called the *projection* and the *constraint* for α, respectively. The projection function ρ_α coerces an arbitrary element of $\mathbf{D}$ to the nearest value that lies within every ideal in α. Hence, ρ_α projects elements onto the ideal a forming the lower bound of α. Similarly, the constraint function ξ_α identifies the elements of $\mathbf{D}$ that *do not belong* to any ideal within α. In particular, α satisfies the formula

   $$\forall x \in \mathbf{D} \ [\xi_\alpha(x) = \mathbf{true} \iff x \notin A] .$$

In contrast to the theory of ideals, a type t is a computable element of $\mathbf{Type_D}$ iff both the *projection* and the *constraint* corresponding to t are computable functions.

3.4. Implications of Formulating Types as Intervals

The most significant and surprising property in the preceding list is the fact that all of the standard type operations are computable functions, yet they are extensions of the corresponding operations on ideals. This result is particularly surprising for the higher order operations $\{\forall, \exists, \mu\}$, since they are not computable in the theory of ideals. The construction required to compute the quantifiers $\forall$ and $\exists$ is described in detail in Section 4.7.

The fact that all of the primitive type operations are computable operations has three important consequences that are not immediately obvious. First, it enables programmers to define interesting new computable type *constructors*. Since recursive type definitions are simply recursive definitions of constants (0-ary functions), they can be freely incorporated in arbitrary recursive programs over any finitary domain $\mathbf{D}$ that includes appropriate subspaces $\mathbf{D_{Type}}$, $\mathbf{D}_\times$, and $\mathbf{D}_\rightarrow$ isomorphic to the domains $\mathbf{Type_D}$, $[\mathbf{D} \times \mathbf{D}]$, and $[\mathbf{D} \rightarrow \mathbf{D}]$. Hence, it is possible to define type constructors (functions from $\mathbf{D}$ to $\mathbf{Type_D}$) using ordinary recursive definitions. For example, the following equation

Tuple(n,t) = **if** n **equal** 0 **then** Empty **else** t$\times$Tuple(n-1,t)

defines the computable type constructor Tuple: $\mathbf{N}^* \times$Type $\rightarrow$ Type where Empty is the total interval containing only the empty sequence and $\mathbf{N}^*$ is the natural numbers augmented by the infinite element ω (the length of an infinite sequence). Tuple(n,t) builds the total type consisting of all tuples of length n formed from type t. For each $n \in \mathbf{N}^*$, Tuple(n,t) is a subtype of the the standard sequence type Seq(t) defined by the equations

Seq(t) = Empty $\cup$ PropSeq(t)
PropSeq(t) = t$\times$Seq(t) .

Second, since types are ordinary data values, it is possible to generalize the type quantifiers $\forall$ and $\exists$ for a domain $\mathbf{D}$ so that they quantify over the total elements $t^\dagger$ ($t \cap \mathbf{D}^\dagger$) of any total type (ideal) t that is *Lawson-compact*. An ideal t over $\mathbf{D}$ is *Lawson-compact* iff iff every infinite set of propositions of the form $b_i \sqsubseteq x$ or $b_i | \sqsubseteq x$ that is inconsistent with $t^\dagger$ has a finite inconsistent subset. If the entire domain $\mathbf{D}$ is Lawson-compact, then every total type $t \in \mathrm{Type_D}$ is Lawson-compact. As a result, for any Lawson-compact domain $\mathbf{D}$, we can define generalized quantifiers $\forall^*$ and $\exists^*$ that are parameterized by the domain of quantification (a total type).

For any Lawson-compact domain $\mathbf{D}$, the generalized quantifiers $\forall^*$ and $\exists^*$ are the continuous functions from $\mathbf{Type_D} \times [\mathbf{D} \rightarrow \mathbf{Type_D}]$ into $\mathbf{Type_D}$ defined by:

$$\exists^*([a,A],f) = < \sqcup_{x \in a^\dagger} f(x)^+ , \sqcap_{x \in A^\dagger} f(x)^- >$$
$$\forall^*([a,A],f) = < \sqcap_{x \in A^\dagger} f(x)^+ , \sqcup_{x \in a^\dagger} f(x)^- > .$$

For every total type $[a,A]$, $a^\dagger$ and $A^\dagger$ are obviously identical. For the sake of notational clarity, we abbreviate the generalized quantifier expressions $\exists^*(A, \lambda t.\alpha(t))$ and $\forall^*(A, \lambda t.\alpha(t))$ > by $\exists t \in A$ $\alpha(t)$ and $\forall t \in A$ $\alpha(t)$, respectively.

If the domain $\mathbf{D}$ includes (an isomorphic image of) $\mathbf{Type_D}$ as a downward-closed retract then the standard type quantifiers are simply instantiations of the generalized type quantifiers where the type parameter is bound to *Type* = $[\mathbf{Type_D}, \mathbf{Type_D}]$:

$$\forall = \lambda f. \forall^*(\mathit{Type},f); \ \exists = \lambda f. \exists^*(\mathit{Type},f) .$$

On any Lawson-compact domain $\mathbf{D}$, the parameterized quantifiers are not only continuous, they are computable in virtually all cases of practical interest. In particular, $\forall^*$ and $\exists^*$ are computable for any Lawson-compact domain $\mathbf{D}$ with a *totally effective* enumeration. A domain $\mathbf{D}$ has a totally effective enumeration iff it is decidable for every finite set of propositions of the form $b_i \sqsubseteq x$ or $b_i | \sqsubseteq x$ whether or not it is consistent with a total element of $\mathbf{D}$. This property obviously depends on the details of the enumeration of the basis $<b_i \mid i \in \mathbf{N}>$. In practice, data domains are almost always defined as the solutions of domain equations constructed using standard domain operations and finite primitive domains, a process that generates totally effective enumerations for the specified domains.

The principal limitation on the applicability of parameterized quantification is the restriction to Lawson-compact types. In practice, many data types are not Lawson-compact. The most important class of counterexamples is the set of infinite, flat data types such as the natural numbers augmented by $\bot$. Fortunately, it is possible to embed any finitely generated flat data type in a larger "lazy" type (see [Cart82] for a discussion of lazy data domains) that is Lawson-compact simply by making all the constructors for the type (*e.g.* the suc operation for the flat natural numbers) non-strict. It is easy to show that every domain **D** that is freely generated by non-strict constructors is Lawson-compact.

Two interesting illustrations of the utility of generalized quantification occur in the context of the Tuple example presented above. First, by using parameterized quantification, we can define the types $\exists n \in N^*$ Tuple(n,t) and $\exists n \in N$ Tuple(n+1,t) which are identical to Seq(t) and PropSeq(t), respectively; these facts are easily proved by fixed-point induction. Second, we can assign the following precise typings to the standard operations Head, Tail, and Cat (concatenation) on sequences:

Head: $\forall n \in N^*\ \forall t \in$ Type Tuple(n+1,t)$\supset$t
Tail: $\forall n \in N^*\ \forall t \in$ Type Tuple(n+1,t)$\supset$Tuple(n,t)
Cat: $\forall m,n \in N^*$
$\forall t \in$ Type Tuple(m,t)$\times$Tuple(n,t)$\supset$Tuple(m+n,t) .

These types are not only total; they are computable. They also imply the more familiar weaker typings:

Head: $\forall t \in$ Type PropSeq(t)$\supset$t
Tail: $\forall t \in$ Type PropSeq(t)$\supset$Seq(t)
Cat: $\forall t \in$ Type Seq(t)$\times$Seq(t)$\supset$Seq(t) .

The third consequence of the effectiveness of the type system is that it reduces the problem of type inference to the problem of reasoning about computable functions. It is straightforward to define both the domain of types (including all affiliated domains) and the standard operations on types within a conventional programming logic for finitary domains such as Edinburgh LCF [Gord77] or the first order theory of domains proposed in [Cart82]. In this context, it is possible to *derive* a set of specialized type inference rules analogous to those proposed by MacQueen, Plotkin, and Sethi for ideals. The only interesting issue involved in this exercise is determining how to generalize the notion of type membership to cope with the fact that an interval type is not a set of data values but a set of ideals (which are sets of data values). The simplest answer is to define two different forms of membership: *necessary* ($x \in \lfloor t \rfloor$) and *possible* ($x \in \lceil t \rceil$. A data value x *necessarily belongs* to type t iff x belongs to every ideal in t (hence, to the lower bound of t). Similarly, a data value x *possibly belongs* to type t iff x belongs to some ideal in t (hence to the upper bound of t). Both of these notions are definable in terms of the approximation relation $\sqsubseteq$ and computable functions on intervals.

For each rule in the MPS type inference system for ideals, the corresponding interval type inference system contains two rules: one for necessary membership and one for possible membership. The interval type system also contains a rule asserting that necessary membership implies possible membership. With the exception of the rules for $\supset$ introduction and elimination, the two interval rules corresponding to an ideal rule look identical to the ideal rule except that necessary and possible membership symbols, respectively, appear in place of conventional membership symbol. The most interesting rules are the rules of abstraction ($\supset$ introduction) and application ($\supset$ elimination) shown in Figure 1.

$$\frac{x \in \lceil \alpha \rceil \vdash M \in \lfloor \beta \rfloor}{\lambda x.M \in \lfloor \alpha \supset \beta \rfloor} \qquad \frac{x \in \lfloor \alpha \rfloor \vdash M \in \lceil \beta \rceil}{\lambda x.M \in \lceil \alpha \supset \beta \rceil}$$

$$\frac{\lambda x.M \in \lfloor \alpha \supset \beta \rfloor,\ t \in \lceil \alpha \rceil}{M_{x \leftarrow t} \in \lfloor \beta \rfloor} \qquad \frac{\lambda x.M \in \lceil \alpha \supset \beta \rceil,\ t \in \lfloor \alpha \rfloor}{M_{x \leftarrow t} \in \lceil \beta \rceil}$$

Abstraction Application

Figure 1: Rules for abstraction and application.

For total intervals, the two notions of membership are obviously equivalent. In practice, programmers deal almost exclusively with total types, eliminating the need to distinguish between the two forms of membership. For total types, the interval rules collapse to the corresponding rules for ideals. Consequently, all derivations of type assertions within the MacQueen-Plotkin-Sethi inference system for ideals can be duplicated verbatim in the corresponding inference system for intervals.

In addition to providing a simple foundation for a type inference system analogous to that proposed by MacQueen, Plotkin, and Sethi, the reduction of type inference to reasoning about computable functions enables us to perform more complex type inferences that require stronger proof rules such as fixed-point induction. The proof of the equivalence of the types $\exists n \in N^*$ Tuple(n,t) and Seq(t) defined in Section 3.4 by fixed-point induction is a good example of this capability.

4. A Mathematical Theory of Types

The remainder of the paper presents a rigorous formalization of interval types and justifies the informal statements made in the previous section. Several of the theorems—most notably the computability of the quantifiers $\forall$ and $\exists$ over Lawson-compact spaces—are quite general and may be applicable in other contexts.

With the possible exception of the naive powerdomain and the Scott and Lawson topologies, the fundamental definitions and lemmas of domain theory underlying the formulation of types as intervals should be familiar to computer scientists who are conversant with domain theory. All elementary definitions and routine proofs have been omitted to conserve space; the definitive reference on the mathematical foundations of domain theory is [Gier80].

Unfortunately, the terminology of domain theory has not been completely standardized. In addition, there are several different formulations of the theory with subtly different properties. This paper is based on Dana Scott's most recent formulation of domains as information systems [Scot81,Scot83]. The reader should be aware that the usage of the terms domain, universal domain, and subspace in this formulation of domain theory is not completely consistent with that found in some widely available references such as [Plot78]. The most significant difference between Scott's new formulation and earlier versions of domain theory is that subspaces ("retracts") are required to be images of algebraic projections not just images of finitary retractions.

The following set of definitions form the foundation for domain theory.

Definition Given a partial order $S = \langle S, \sqsubseteq \rangle$, a subset $R \subseteq S$ is *consistent* iff it has an upper bound in S. R is *directed* iff every finite subset $E \subseteq R$ has an upper bound in R. R is *filtered* iff every finite subset $E \subseteq R$ has a lower bound in R.

Definition A partial order S is *complete* iff every directed subset $R \subseteq S$ (including the empty set) has a least upper bound in S. The least upper bound in S of the empty set is denoted $\bot_S$. The phrase "complete partial order" is frequently abbreviated *cpo*.

Definition An element s of a cpo S is *finite* iff for every directed subset $R \subseteq S$ has the property that $s \sqsubseteq \sqcup_S R$ implies that $\exists r \in S$ such that $s \sqsubseteq r$; it is *infinite* iff it is not finite. An element s is *total* if it is maximal under the approximation ordering $\sqsubseteq$: $\forall x \in S\ s \sqsubseteq x \supset s = x$.

Notation Let R be an arbitrary subset of a cpo S. The set of finite elements of R (within S) is denoted R^0. Similarly, the set of total elements of R is denoted $R^\dagger$.

Definition A subset R of a cpo S forms a *basis* for S iff it satisfies the following two properties:

(i) R is closed under the least upper bound operation on finite consistent subsets.

(ii) Every element $x \in S$ is the least upper bound of the subset of R that approximates it, *i.e.*

$$x \in S \;\; x = \sqcup_S \{y \in R \mid y \sqsubseteq x\}.$$

Definition A *domain* D is a pair $<D,\beta>$ consisting of a complete partial order D and an enumeration $\beta = \{b_i \mid i \in N\}$ of the finite elements D^0 of D.[7]

Definition A domain D is *finitary* iff D is *algebraic*: the set D^0 of finite elements forms a basis for D.

Definition The *finitary basis* of a finitary domain D is the set D^0 of finite elements of D.

Notation When no confusion is possible, we will frequently omit the subscripts (identifying a domain) on the symbols $\sqcup$ (*sup*), $\sqcap$ (*inf*), and $\perp$. In addition, we will often use the symbol D denoting a domain in place of the symbols D and D.

Definition An n-ary function $f:D^n \rightarrow D$ is *monotonic* iff

$$\forall [x_1,\ldots,x_n], [y_1,\ldots,y_n] \in D^n \;\; x_1 \sqsubseteq y_1 \wedge \ldots \wedge x_n \sqsubseteq y_n \supset f(x_1,\ldots,x_n) \sqsubseteq f(y_1,\ldots,y_n).$$

The function f *preserves directed sups* (*filtered infs*) iff for every n-tuple $S_1,\ldots,S_n$ of directed (filtered) subsets of D,

$$f(\sqcup S_1,\ldots,\sqcup S_n) = \sqcup (\sqcap) \{f(d_1,\ldots,d_n) \mid (d_1,\ldots,d_n) \in S_1 \times \ldots \times S_n)\}.$$

It is *strict* iff the image of every argument list containing $\perp$ is $\perp$:

$$\forall x_1,\ldots,x_n \in D \;\; x_1 = \perp \;\ldots\; x_n = \perp \supset f(x_1,\ldots,x_n) = \perp.$$

For reasons that we will explain in Section 4.2, functions that preserve directed sups are called *Scott-continuous* (or simply *continuous*) functions. Similarly, functions that preserve both directed sups and filtered infs are called *Lawson-continuous*.

4.1. Fundamental Domain Constructions

In specifying finitary domains, it is often convenient to construct composite domains from simpler ones. Although there are many useful domain constructors, most of those that occur in practice can be recursively defined in terms of three fundamental constructions: the Cartesian product construction (denoted $A \times B$), the coalesced sum construction (denoted $A+B$), and the (Scott) continuous function construction (denoted $A \rightarrow B$). For a precise definition of these constructions, see [Scot81,Scot83]. In this paper, we will also rely heavily on four other domain constructions that are all related to the familiar powerset construction from set theory: the retract power domain, the open and closed power domains, and the naive power domain. Each power domain construction takes a finitary domain D and generates a finitary domain containing a different class of subsets of D. The definition of the retract power domain appears below. We will define the remaining power domain constructions as soon as we introduce a sufficient set of supporting definitions.

Definition A domain A is a *retract* (or subspace) of the domain B iff[2]

(i) $A \subseteq B$; $\sqsubseteq_A = \{(x,y) \mid x,y \in A \wedge (x,y) \in \sqsubseteq_B\}$; and $\perp_A = \perp_B$.

(ii) $A^0 = A \cap B^0$.

(iii) For all directed subsets $R \subseteq A$, $\sqcup_A R = \sqcup_B R$.

The function π_A defined by

$$\pi_A(x) = \sqcup \{y \in A^0 \mid y \sqsubseteq x\}$$

is called the *algebraic projection* corresponding to A.

Definition A domain A is a *weak retract* of the domain B iff

(i) $A \subseteq B$; $\sqsubseteq_A = \{(x,y) \mid x,y \in A \wedge (x,y) \in \sqsubseteq_B\}$; and $\perp_A = \perp_B$.

(ii) For all $x,y \in A$, $\{x,y\}$ is consistent in A iff $\{x,y\}$ is consistent in B.

Any continuous function $f:B \rightarrow B$ such that $f \circ f = f$ and $f(B) = A$ is called a *retraction* for A.

Remark Every retract is obviously a weak retract. The converse, however, is false because a finite element of a weak retract is not necessarily a finite element of the parent domain. Similarly, the least upper bound relation within a weak retract may not be a restriction of the least upper bound relation on the parent space.

For the remainder of the section, let D be an arbitrary domain with enumeration $<b_i \mid i \in N>$.

Definition The domain of retracts $\mathbf{Ret_D}$ is defined as the pair $<Ret_D,\rho>$ where Ret_D is the partial order consisting of the set of retracts of D under the subset relation and ρ is the enumeration $<R_i \mid i \in N>$ consisting of all finite retracts (finite sets in Ret_D) sorted by rank

$$\sum_{\{i \mid b_i \in R \;\; \forall j < i \; b_j \neq b_i\}} 2^i.$$

It is easy to verify that the set $\{R_i \mid i \in N\} = (Ret_D)^0$, confirming that $\mathbf{Ret_D}$ is in fact a domain.

Lemma If D is finitary, then so is $\mathbf{Ret_D}$.

Definition The partially ordered set of weak retracts $WeakRet_D$ is defined as the pair $<WeakRet_D, \subseteq>$ where $WeakRet_D$ is the set of weak retracts of D and $\subseteq$ is the subset relation on $WeakRet_D$.

Remark The partial order $WeakRet_D$ is *not* complete, because pairs of consistent weak retracts do not necessarily have least upper bounds.

4.2. The Scott and Lawson Topologies

Definition A subset S of a partially ordered set D is *downward closed* iff $\forall x \in S \; \forall y \in D \; (y \sqsubseteq x \supset y \in S)$. S is *upward closed* iff $\forall x \in S \; \forall y \in D \; (y \sqsupseteq x \supset y \in S)$. The *upward closure of* S, denoted $S\uparrow$, is the set $\{x \in D \mid \exists y \in S \; y \sqsubseteq x\}$. The *downward closure of* S, denoted $S\downarrow$, is the set $\{x \in D \mid \exists y \in S \; x \sqsubseteq y\}$. We will abbreviate the upward and downward closure of a singleton set $\{x\}$ by the symbols $x\uparrow$ and $x\downarrow$, respectively.

Definition Let S be an arbitrary set. A *topology* σ on S is a family σ_S of subsets of S, called the *σ-open* sets of S, with the following three properties:

(i) $S \in \sigma_S$.

(ii) For every subset V of σ_S, $\bigcup_{s \in V} s \in S$.

(iii) For every finite subset F of σ_S, $\bigcap_{s \in F} s \in S$.

Remark Note that property (iii) implies that the empty set ϕ belongs to σ_S.

Definition Let σ be a topology on the universe S. A subset $\omega \subseteq \sigma$ is a *sub-basis* for σ iff σ is the closure of ω under *arbitrary* unions and finite intersections. σ is called the topology *generated* by the sub-basis ω.

Definition Let σ be a topology on the universes A and B. A function $f:A \rightarrow B$ is *σ-continuous* iff the inverse image under f of every σ_B-open set is σ_A-open: $\forall S \in \sigma_B \; f^{-1}(S) \in \sigma_A$.

Definition Let σ be a topology on the universe A. A subset S of σ *covers* a subset B of the universe A iff $B \subseteq \bigcup_{s \in S} s$. S is called a σ-covering of B. A subset B of A is *σ-compact* iff every σ-covering has a finite subset (called a *finite σ-subcovering*) that covers B.

---

[7]Since the elements in the enumeration are not necessarily distinct, D can be finite.

Definition Let σ_A be a topology on the universe A. A subset $S \subseteq A$ is *σ-closed* iff its complement A–S is σ-open, i.e. $A\text{-}S \in \sigma_A$.

Notation If the universe A is clear from context, we will denote the complement of a set S with respect to A by $\bar{S}$ (or alternately $\neg S$).

Definition (The Scott Topology) A subset S of the domain **D** is *Scott-open* (or simply *open*) iff S is upward closed and $\forall x \in S\, \exists y \in S\ [y \text{ is finite} \wedge y \sqsubseteq x]$.

Definition Let S be a downward closed subset of the domain **D**. The *boundary* of S (denoted ΔS) is the set $\{y \in \bar{S} \mid \forall x \in (\bar{S})^0\ x \not\sqsubseteq y\}$. The *Scott-closure* of S (denoted $|S|$) is the set $S \cup \Delta S$. The Scott-closure of an arbitrary subset $S \subseteq \mathbf{D}$ is the set $|S\!\downarrow|$.

Lemma S is Scott-closed iff $S = |S|$.

Definition For every domain **D**, the open (closed) powerset Op_D (Cl_D) is the cpo consisting of the universe Op_D (Cl_D) of open (closed) subsets of D under the subset relation.

Lemma A Scott-open (Scott-closed) set $O \in \mathbf{Op}_D$ ($C \in \mathbf{Cl}_D$) is finite in the cpo $< Op_D$ (Cl_D) iff there exists finite set F of finite elements of **D** such that $O = F\!\uparrow$ ($C = F\!\downarrow$).

Definition The domain $\mathbf{Op}_D$ ($\mathbf{Cl}_D$) is the pair $< Op_D, \sigma >$ ($< Cl_D, \sigma >$) where σ is the enumeration $< S_i \mid i \in N >$ consisting of all sets $\{S\!\uparrow (S\!\downarrow) \mid S \subseteq D^0 \text{ and S is finite}\}$ sorted by rank

$$\sum_{\{j \mid b_j \in S\ \forall k < j\ b_k \neq b_j\}} 2^i .$$

Theorem If **D** is finitary, then so is $\mathbf{Op}_D$ ($\mathbf{Cl}_D$).

Remark In the literature on types, the Scott-closed sets over a domain **D** are usually called the *ideals* of **D**.

Theorem A function f mapping a finitary domain **A** into a finitary domain **B** is Scott-continuous iff it preserves directed sups.

Definition (The Lawson Topology) A subset S of a finitary domain **D** is *Lawson-open* iff it is a member of the family of sets λ(**D**) generated by the *sub-basis* $\{x\!\uparrow \mid x \in D^0\} \cup \{D\text{-}x\!\uparrow \mid x \in D^0\}$.

Theorem A function f mapping a finitary domain **A** into a finitary domain **B** is Lawson-continuous iff it preserves both directed sups and filtered infs.

4.3. The Naive Powerdomain

For any domain **D**, there is a corresponding domain of subsets $\mathbf{2}^D$, called the naive powerdomain, that includes both $\mathbf{Op}_D$ and $\mathbf{Cl}_D$ and respects the same approximation and consistency relations. As before, let **D** be an arbitrary domain with basis enumeration $< b_i \mid i \in N >$.

Definition A subset $S \subseteq D$ is *directed-closed* iff $\forall R \subseteq S$ R directed $\supset \sqcup R \in S$. The *directed-closure* of S (denoted $|S|$) is the set $\{x \mid \exists R \subseteq S \text{ R directed } \sqcup R = x\}$.

Definition The naive powerset $\mathcal{2}^D$ over **D** is the cpo consisting the universe $\{S \subseteq D \mid S = |S^0|\}$ under the subset relation.

Lemma The finite elements of $\mathcal{2}^D$ are precisely the finite sets in 2^D.

Definition The naive powerdomain $\mathbf{2}^D$ over **D** is the pair $< \mathcal{2}^D, \sigma >$ where σ is the enumeration $< S_i \mid i \in N >$ consisting of all sets $\{|S| \mid S \subseteq D^0 \text{ and S is finite}\}$ sorted by rank

$$\sum_{\{j \mid b_j \in S\ \forall k < j\ b_k \neq b_j\}} 2^i .$$

Lemma $\mathbf{2}^D$ is a finitary domain.

We define analogs in $\mathbf{2}^D$ to the standard operations on subsets of **D** as follows:

Definition The *union* and *intersection* functions , $\mathbf{2}^D \to \mathbf{2}^D$ are defined by:

A B = $|A^0 \cup B^0|$; A B = $|A^0 \cap B^0|$.

The *complement* function $\tilde{}: \mathbf{2}^D \to \mathbf{2}^D$ is defined by: $\tilde{}(S) = |D^0\text{-}S^0|$.

Lemma The functions and are continuous but the function ~ is is antimonotonic and hence is not continuous.

The set functions ,, and ~ *do not* necessarily yield the same answers as the analogous set operations ∩, ∪, and ¬ on arbitrary sets. The following lemma identifies sufficient conditions for ensuring that they agree.

Lemma Let **D** be a finitary domain.

(i) For all sets $A,B \in \mathbf{Op}_D \cup \mathbf{Cl}_D$, AB = $A \cap B$.

(ii) For arbitrary sets $A,B \in \mathbf{2}^D$, AB = $A \cup B$.

(iii) **Lemma** For every set $A \in \mathbf{Op}_D \cup \mathbf{Cl}_D$, $\tilde{}A = \bar{A}$.

Definition Let **D** be a finitary domain and let $\mathbf{2}^D$ be the naive powerdomain over **D**. An n-ary function $f:(\mathbf{2}^D)^n \to \mathbf{2}^D$ is *tidy* iff

(i) f is Lawson-continuous (preserves both directed sups and filtered infs),

(ii) f *preserves closed sets*: if $C_1,\ldots,C_n \in \mathbf{Cl}_D{}^n$, then $f(C_1,\ldots,C_n) \in \mathbf{Cl}_D{}^n$, and

(iii) f *preserves open sets*: if $O_1,\ldots,O_n \in \mathbf{Op}_D{}^n$, then $f(O_1,\ldots,O_n) \in \mathbf{Op}_D{}^n$.

All of the naive set operations that we discuss in the remainder of the paper will be tidy. We will subsequently show that every tidy set operation induces a continuous operation on interval types that preserves total types.

4.4. Computability

In order to formalize the idea of computable functions on a domain, we must identify a concrete representation for the elements of the domain.

Definition A domain **D** is *effective* iff it is finitary and the following two relations are recursive:

(i) The binary relation **CON** defined by

$$\mathbf{CON}(i,j) \iff \exists k\ b_i \sqsubseteq b_k \wedge b_j \sqsubseteq b_k .$$

(ii) The ternary relation **LUB** defined by

$$\mathbf{LUB}(i,j,k) \iff b_k = \sqcup\{b_i, b_j\} .$$

Theorem The constructed domains $|\mathbf{D} \to \mathbf{E}|$, $|\mathbf{D} \times \mathbf{E}|$, $\mathbf{D}+\mathbf{E}$, $\mathbf{Ret}_D$, $\mathbf{Op}_D$, $\mathbf{Clp}_D$, and $\mathbf{2}^D$ are effective if the component domains **D** and **E** are.

Definition A subspace **A** (with enumeration $\alpha = < a_i \mid i \in N >$) of a finitary domain **B** (with enumeration $\beta = < b_i \mid i \in N >$) is *effective* iff the function $\text{rep}N \to N$ defined by

$$\text{rep}(i) = \min \{j \mid b_j = a_i\}$$

is recursive.

Definition An element d of an effective domain **D** with enumeration δ is *computable* iff the index set $\{i \mid \delta_i \sqsubseteq d\}$ is recursively enumerable.

Definition Let **A** and **B** be effective domains with enumerations $\alpha = \{a_i \mid i \in N\}$ and $\beta = \{b_i \mid i \in N\}$. A continuous function $f:\mathbf{A} \to \mathbf{B}$ is *computable* iff f is a computable element.

Theorem f is computable iff the relation F defined by $\{(i,j) \mid b_j \sqsubseteq f(a_i)\}$ is recursively enumerable.

Definition For any finitary domain $\mathbf{D}$, the *least fixed-point operator* $\mathbf{Y}$: $[\mathbf{D}\to\mathbf{D}]\to\mathbf{D}$ is defined by the equation

$$\mathbf{Y}\, f = \sqcup_{i\in N} f^{\,i}(\bot)$$

where $f^{\,i}$ denotes i compositions of the function f ($f^{\,0} = \lambda x.\bot$).

Theorem $\mathbf{Y}$ has the property that $\mathbf{Y}f$ is the least fixed-point of f, *i.e.* the least element d such that f(d)=d.

Theorem If $\mathbf{D}$ is effective, then $\mathbf{Y}$ is computable.

Definition A *universal domain* $\mathbf{U}$ is an effective domain in which every data domain $\mathbf{D}$ is isomorphic to a subspace $S_{\mathbf{D}}$ of $\mathbf{U}$. In addition, if $\mathbf{D}$ is effective, $S_{\mathbf{D}}$ must be an effective subspace of $\mathbf{U}$.

Theorem There exists a universal domain $\mathbf{U}$.

Proof See [Scot 81, Scot83]. □

Since every domain $\mathbf{D}$ has an isomorphic image $S_{\mathbf{D}}$ within the universal domain, the problem of defining an arbitrary domain can be reduced to defining an arbitrary subspace of a particular universal domain. The following theorem (in conjunction with Kleene's recursion theorem) implies that we can recursively define effective subspaces of a universal domain using the domain constructors $\times$, $+$, and $\to$.

Notation Let $\mathbf{U}_{\mathrm{Ret}}$ denote an effective subspace of $\mathbf{U}$ that is isomorphic to $\mathbf{Ret}_{\,\mathbf{U}}$.

Theorem Let $\mathbf{U}$ be a universal domain and let $a,b\in\mathbf{Ret}_{\,\mathbf{U}}$ be elements of $\mathbf{U}$ representing the subspaces $A,\in\mathbf{Ret}_{\,\mathbf{U}}$. For the three basic domain constructions $\{\times, +, \to\}$, there are corresponding computable functions mkProd, mkSum, and mkFun:$\mathbf{U}_{\mathrm{Ret}}\times\mathbf{U}_{\mathrm{Ret}}\to\mathbf{U}_{\mathrm{Ret}}$ such that mkProd(a,b) represents the subspace isomorphic to $\mathbf{A}\times\mathbf{B}$, mkSum(a,b) represents the subspace isomorphic to $\mathbf{A}+\mathbf{B}$, and mkFun(a,b) represents the subspace isomorphic to $\mathbf{A}\to\mathbf{B}$. Similarly, for the four power domain constructions {**Ret**, **Op**, **Cl**, **Pow**}, there are computable functions mkRet, mkOp, mkCl, mkPow: $\mathbf{U}_{\mathrm{Ret}} \to \mathbf{U}_{\mathrm{Ret}}$ such that the applications mkRet(a), mkOp(a), mkCl(a), mkPow(a) yield elements representing the subspaces of $\mathbf{U}$ isomorphic to $\mathbf{Ret}_{\mathbf{A}}$, $\mathbf{Op}_{\mathbf{A}}$, $\mathbf{Cl}_{\mathbf{A}}$, and $\mathbf{2}^{\mathbf{A}}$, respectively.

4.5. Definition of Interval Types

We have finally laid sufficient groundwork to define the set of interval types over a finitary domain $\mathbf{D}$ and show that it forms a finitary domain.

Definition An *interval type* (or simply *interval*) [a,A] on the finitary domain $\mathbf{D}$ is the set of ideals $\{I\in \mathrm{Cl}_{\mathbf{D}} \mid a\subseteq I\subseteq A\}$ on D where $a\subseteq A$ and a is non-empty. The set of interval types over a domain $\mathbf{D}$ is denoted $\mathbf{Type}_{\mathbf{D}}$.

Theorem For every finitary domain, the set $\mathbf{Type}_{\mathbf{D}}$ of interval types over $\mathbf{D}$ forms a finitary domain under the superset ordering $\supseteq$ on intervals (as sets). The total elements of $\mathbf{Type}_{\mathbf{D}}$ are intervals of the form [A,A] containing a single ideal A.

Definition A total element of the domain $\mathbf{Type}_{\mathbf{D}}$ is called an *ideal image* over $\mathbf{D}$. The set of ideal images over $\mathbf{D}$ is denoted $\mathrm{Idl}_{\mathbf{D}}$.

The easiest way to prove the preceding theorem is to show that $\mathbf{Type}_{\mathbf{D}}$ is isomorphic to the domain of brackets, which are concrete representations for intervals that expose their computational properties. The following collection of definitions and lemmas define the domain of brackets and formalize the relationship relationship between brackets and intervals.

Definition A *bracket* $<a,\bar{A}>$ on the finitary domain $\mathbf{D}$ is a pair of subsets a, $\bar{A} \supseteq$ D such that: (i) a is non-empty and closed; (ii) $\bar{A}$ is open; and (iii) a and A are disjoint. The three sets a, $\bar{A}$, and $\neg(a\cup\bar{A})$ are called the *positive*, *negative*, and *neutral regions*, respectively, of the bracket A.

Remark The bracket $<a,\bar{A}>$ represents the interval [a,A].

Theorem The set of brackets on a finitary domain $\mathbf{D}$ (denoted $\mathbf{Bkt}_{\mathbf{D}}$) forms a finitary domain under the approximation ordering $\subseteq$ defined by $<a,\bar{A}> \subseteq <b,\bar{B}>$ iff $a\subseteq b \wedge \bar{A}\subseteq\bar{B}$. If $\mathbf{D}$ is effective, then $\mathbf{Bkt}_{\mathbf{D}}$ is effective. A bracket $<a,\bar{A}>$ in the domain $\mathbf{Bkt}_{\mathbf{D}}$ is *finite* iff a is finite in $\mathrm{Cl}_{\mathbf{D}}$ and A is finite in $\mathrm{Op}_{\mathbf{D}}$. $<a,\bar{A}>$ is total iff $a\cup\bar{A} = \mathbf{D}$. Given a universal domain $\mathbf{U}$, there is a computable function mkBkt:$\mathbf{Ret}_{\,\mathbf{U}}\to\mathbf{Ret}_{\,\mathbf{U}}$ that maps each subspace $\mathbf{D}$ of the universal domain $\mathbf{U}$ into a subspace isomorphic to $\mathbf{Bkt}_{\mathbf{D}}$.

Since the domain of brackets and the domain of intervals over a domain $\mathbf{D}$ are isomorphic, we can identify the two domains without any loss of precision. Henceforth, we will frequently use a bracket expression $<\alpha,\beta>$ to denote the corresponding interval type $[\alpha,\bar{\beta}]$.

Notation Let τ be an arbitrary interval [t,T] in $\mathbf{Type}_{\mathbf{D}}$. The positive and negative regions of (the bracket corresponding to) τ are denoted τ^+ and τ^-, respectively. Obviously, $\tau^+ = t$ and $\tau^- = \bar{T}$.

4.6. Type Operations

For the remainder of the paper we will adopt the following notational conventions.

Notation Let $\mathbf{U}$ denote a universal domain and let $\mathbf{U}_\to$, $\mathbf{U}_\times$, and $\mathbf{U}_+$ denote computable subspaces of $\mathbf{U}$ that are isomorphic to $[\mathbf{U}\to\mathbf{U}]$, $[\mathbf{U}\times\mathbf{U}]$, and $\mathbf{U}+\mathbf{U}$, respectively. Let $\to_{\,\mathbf{U}}$, $\times_{\,\mathbf{U}}$, and $+_{\,\mathbf{U}}$ denote the computable functions from $\mathbf{Ret}_{\,\mathbf{U}}{}^2$ into $\mathbf{Ret}_{\,\mathbf{U}}$ that map arbitrary subspaces A and B into the isomorphic images of A→B, A×B, and A+B within $\mathbf{U}_\to$, $\mathbf{U}_\times$, and $\mathbf{U}_+$, respectively. Let $(a,b)_{\,\mathbf{U}}$ and $f_{\,\mathbf{U}}$ denote the isomorphic images (in $\mathbf{U}$) of the elements $(a,b)\in A\times B$ and $f\in[A\to B]$, respectively. Similarly, let $\mathrm{in}_L(a)$ and $\mathrm{in}_R(b)$ denote the isomorphic images of the elements $(0,a)\in A+B$ and $(1,b)\in A+B$, respectively. In contexts where no confusion is possible, we will omit the subscript $_{\mathbf{U}}$ from the functions $\to_{\,\mathbf{U}}$, $\times_{\,\mathbf{U}}$, $+_{\,\mathbf{U}}$, and $(\,,\,)_{\,\mathbf{U}}$.

In a programming language, the data domain $\mathbf{D}$ typically consists of a disjoint collection of subspaces such as truth values, integers, tuples, and functions. Consequently, we will restrict our attention to program data domains that satisfy the following condition.

Definition Let $\mathbf{A}_1,\ldots,\mathbf{A}_n$ be flat subspaces of $\mathbf{U}$. The *standard domain* $\mathbf{D}$ with atomic types $\mathbf{A}_1,\ldots,\mathbf{A}_n$ is the subspace of $\mathbf{U}$ defined by the domain equation

$$\mathbf{D} = \mathbf{D}\to\mathbf{D} + \mathbf{D}\times\mathbf{D} + (\mathbf{D}+\mathbf{D}) + \mathbf{A}_1 + \ldots + \mathbf{A}_n\,.$$

We will denote the unary injection functions mapping each of the component spaces $\mathbf{A}_1,\ldots,\mathbf{A}_n$, $\mathbf{D}\to_{\,\mathbf{U}}\mathbf{D}$, $\mathbf{D}\times_{\,\mathbf{U}}\mathbf{D}$, and $\mathbf{D}+_{\,\mathbf{U}}\mathbf{D}$ into $\mathbf{D}$ by the function symbols $\mathbf{In}_1$, ..., $\mathbf{In}_n$, $\mathbf{In}_\to$, $\mathbf{In}_\times$, and $\mathbf{In}_+$, respectively. Similarly, we will denote the subspaces of $\mathbf{D}$ that are the injections of each of the same component spaces by $\mathbf{D}_1$, ..., $\mathbf{D}_n$, $\mathbf{D}_\to$, $\mathbf{D}_\times$, and $\mathbf{D}_+$, respectively.

With the exception of the polymorphic function type constructor $\supset$, all of the basic type constructors $\{\times, +, \cup, \cap\}$ on a standard domain $\mathbf{D}$ are defined in a uniform way from tidy operations on the corresponding naive powerdomain $\mathbf{2}^{\mathbf{D}}$. For the sake of concreteness, we will describe the constructions in terms of bracket notation.

Definition Let $\mathbf{D}$ be a finitary domain and let $t:(\mathbf{2}^{\mathbf{D}})^n\to\mathbf{2}^{\mathbf{D}}$ be an n-ary tidy operation on $\mathbf{2}^{\mathbf{D}}$. The *type operation* τ on $\mathbf{Type}_{\mathbf{D}}$ *induced by* t is the function τ defined by:

$$\tau([a_1,A_1], \ldots, [a_n,A_n]) = [t(a_1, \ldots, a_n), t(A_1,\ldots,A_n)]$$

where $|a_1,A_1|,\ldots,|a_n,A_n|$ denotes an n-tuple of intervals.

Remark It is easy to demonstrate that $\tau(|a_1,A_1|,\ldots,|a_n,A_n|)$ must be an interval because t is tidy.

Lemma τ is a continuous function from $\mathbf{Type_D}^n \to \mathbf{Type_D}$ that preserves totality.

Proof It is obvious from the definition of the induced operation τ that it preserves totality. The easiest way to prove that τ is continuous to express τ in terms of brackets. The function τ clearly decomposes into two separate functions $\tau^+:\mathbf{Cl_D}^n \to \mathbf{Cl_D}$ and $\tau^-:\mathbf{Op_D}^n \to \mathbf{Op_D}$ defined by:

$$\tau(|a_1,A_1|,\ldots,|a_n,A_n|) = |\tau^+(a_1,\ldots,a_n),\ \neg\ \tau^-(\bar{A}_1,\ldots,\bar{A}_n)|$$
$$\tau^+(a_1,\ldots,a_n) = t(a_1,\ldots,a_n)$$
$$\tau^-(\bar{A}_1,\ldots,\bar{A}_n) = \neg\ t(A_1,\ldots,A_n)$$

that yield the positive and negative regions of the output of τ. If we re-express the same decomposition in terms of brackets, it takes the following form:

$$\tau(<a_1,\bar{A}_1>,\ldots,<a_n,\bar{A}_n>) = <\tau^+(a_1,\ldots,a_n),\ \tau^-(\bar{A}_1,\ldots,\bar{A}_n)>$$
$$\tau^+(a_1,\ldots,a_n) = t(a_1,\ldots,a_n)$$
$$\tau^-(\bar{A}_1,\ldots,\bar{A}_n) = \neg\ t(\neg\bar{A}_1,\ldots,\neg\bar{A}_n) = {}^{\sim}t({}^{\sim}\bar{A}_1,\ldots,{}^{\sim}\bar{A}_n)\ .$$

Since the finite elements of $\mathbf{Type_D}$ are pairs of disjoint finite elements in $\mathbf{Cl_D}\times\mathbf{Op_D}$, the continuity of τ reduces to the continuity of the component functions τ^+ and τ^-. Moreover, since t is continuous and preserves closed sets, the function τ^+ must be continuous, reducing the continuity of τ to the continuity of the negative component function τ^-.

To prove that τ^- is continous, let R be a directed set of n-tuples in $\mathbf{Type_D}^n$. We must show that

$$\tau^-(\sqcup R) = \sqcup_{r\in R}\ \tau^-(r)\ .$$

By the definition of the functions τ^- and ~, we can simplify both sides of preceding equation as follows:

(1) $\tau^-(\sqcup R) = {}^{\sim}t({}^{\sim}\sqcup R) = {}^{\sim}t(\sqcap\ {}^{\sim}R)$

(2) $\sqcup_{r\in R}\ \tau^-(r) = \sqcup_{r\in R}\ {}^{\sim}t({}^{\sim}r) = {}^{\sim}(\sqcap_{s\in {}^{\sim}R}\ t(s))$,

reducing it to

(3) ${}^{\sim}t(\sqcap\ {}^{\sim}R) = {}^{\sim}(\sqcap_{s\in {}^{\sim}R}\ t(s))$.

Since R is a directed set of n-tuples of open sets, the set ~R must be a filtered set of n-tuples of closed sets. Hence, (3) is an immediate consequence of the Lawson-continuity of t, which forces t to preserve the inf of ~R. □

Definition The *basic type constructors* $\{\times,\ +,\ \cup,\ \cap\}$ on a standard domain **D** are the type operations induced by the tidy set functions $\{\ \times_D,\ +_D,\ ,\ \}$ on $\mathbf{2^D}$ where $\times_D$ and $+_D$ are defined by

$$R \times_D S = \{\ in_\times((r,s)_U) \mid r\in R,\ s\in S\ \}$$
$$R +_D S = \{\ in_+(in_L(r)) \mid r\in R\ \} \cup \{\ in_+(in_R(s)) \mid s\in R\ \}\ .$$

Remark The subscript D refers to the fact that the functions $\times_D$ and $+_D$ are derived from the standard set theoretic functions $\times$ and + by injecting their outputs first into **U** and then into **D**.

Although tidy set operations always induce continuous type constructors, the induced constructors are not necessarily computable—even when the inducing operations are computable. The reason for this anomaly is that the complement operation ~ appearing in the definition of the negative component function τ^- is not computable (it is not even monotonic). Fortunately, all of the basic type constructors happen to be computable, because in each case the non-computable expression denoting the negative component can be transformed into an equivalent expression composed from computable operations. The appropriate transformation, however, depends on the particular operation.

Lemma Let **D** be a standard domain. The type constructors $\{\times,\ +,\ \cup,\ \cap\}$ on $\mathbf{Type_D}$ induced by the corresponding tidy operations $\{\times_D,\ +_D\ ,\ ,\ \}$ on $\mathbf{2^D}$ are computable.

Proof The positive components in the definition of all the type constructors are computable because the inducing operations on $\mathbf{2^D}$ are obviously computable. Hence, the proof of the lemma reduces to showing that for each type constructor τ, the negative component $\tau(<a,\bar{A}>,<b,\bar{B}>)^-$ of an arbitrary application is computable. Since ${}^{\sim}(D\times_D D)$ and ${}^{\sim}(D+_D D)$ are both computable elements of $\mathbf{2^D}$, the following identities—which hold for arbitrary sets $A,B\in\mathbf{Op_D}\cup\mathbf{Cl_D}$—reduce the negative components of $\times$ and + to computable form:

$${}^{\sim}({}^{\sim}A\times_D{}^{\sim}B) = {}^{\sim}(D\times_D D) \cup (A\times_D D) \cup (D\times_D B)$$
$${}^{\sim}({}^{\sim}A+_D{}^{\sim}B) = {}^{\sim}(D+_D D) \cup in_+(in_L(A)) \cup in_+in_R(B))\ .$$

Similarly, DeMorgan's Laws for sets $A,B\in\mathbf{Op_D}\cup\mathbf{Cl_D}$ reduce the negative components of $\cup$ and $\cap$ to computable form

$${}^{\sim}({}^{\sim}A{}^{\sim}B) = AB\ ; \quad {}^{\sim}({}^{\sim}A{}^{\sim}B) = (AB)\ .$$

Although many interesting type operations are induced by tidy functions on the naive powerdomain, the polymorphic function type constructor $\supset$ is not among them because it does not maintain the strict separation of positive and negative information that characterizes induced operations. It must be defined as a special case.

Definition Let **D** be a standard domain. The *polymorphic function set constructor* $\supset_D$ on $\mathbf{2^D}$ on **D** is defined by

$$R\supset_D S = \{in_\to(f_U) \mid f\in[\mathbf{D}\to\mathbf{D}] \wedge \forall x\in R\ f(x)\in S\}\ .$$

The function type constructor $\supset$ on $\mathbf{Type_D}$ determined by $\supset_D$ is defined by the rule

$$|a,A|\supset|b,B| = |A\supset_D b,\ a\supset_D B|\ .$$

In bracket notation,

$$<a,\bar{A}>\supset<b,\bar{B}> = <{}^{\sim}\bar{A}\supset_D b,\ {}^{\sim}(a\supset_D{}^{\sim}\bar{B})>\ .$$

Remark To confirm that $\supset_D$ maps $\mathbf{2^D}\times\mathbf{2^D}$ into $\mathbf{2^D}$, we must show that given arbitrary elements R and S in $\mathbf{2^D}$ $R\supset_D S$ is an element of $\mathbf{2^D}$. Let the notation f_{in} abbreviate the expression $in_\to(f)$. Let f_{in} be an arbitary element in the set $R\supset_D S$. We must show that $f_{in} = \sqcup\ \{g_{in}\in R\supset_D S \mid g_{in}$ is finite $\wedge\ g_{in}\subseteq f_{in}\}$ or equivalently that $f = \sqcup\ \{g\in D\to D \mid \forall x\in R\ g(x)\in S \wedge g$ is finite in $\mathbf{D}\to\mathbf{D} \wedge g\subseteq f\}$. But every function $g\subseteq f$ has the property that $\forall x\in R\ g(x)\subseteq f(x)\in S$. Hence, for every finite element g in $\mathbf{D}\to\mathbf{D}$ approximating f, g_{in} is a finite element in $R\supset_D S$. Since the finite elements of $\mathbf{D}\to\mathbf{D}$ form a basis, f_{in} is the least upper bound of the finite elements of $R\to S$ that approximate it. □

Theorem Let **D** be a standard domain. The function type constructor $\supset$ on $\mathbf{Type_D}$ is computable and preserves totality.

Proof Let $<a,\bar{A}>$ and $<b,\bar{B}>$ be arbitrary effective elements of $\mathbf{Type_D}$. To prove that $\supset$ is computable, we must show that $<a,\bar{A}>\supset<b,\bar{B}>$ is effective, i.e. the set of finite elements of $\mathbf{Type_D}$ that approximate $<a,\bar{A}>\supset<b,\bar{B}>$ is recursively enumerable. Since the set of finite elements in a standard domain **D** belonging to the complement of the function subspace (*i.e.*, the set $(\bar{D})^0$) is recursively enumerable, the effectiveness of $<a,\bar{A}>\supset<b,\bar{B}>$ reduces to the recursive enumerability of the following two sets: the one-step-functions[5] $u \mapsto v$ ($u,v \in D^0$) that are members of $A \supset_D b$ and the one-step-functions that are members of $a \supset_D \bar{B}$. In the former case, a finite element $u \mapsto v \in A \supset_D b$ iff either $u \in \bar{A}$ or $v \in b$ which are both recursively enumerable by hypothesis (the effectiveness of the inputs). In the latter case, $u \mapsto v \in a \supset_D \bar{B}$ iff $u \in a$ and $v \in \bar{B}$, which are also both recursively enumerable. Hence, $\supset$ is computable.

To show that $\supset$ preserves totality, we simply observe that if $[a,A]$ and $[b,B]$ are total intervals (ideal images) then $a = A$ and $b = B$, implying that $[a,A] \supset [b,B] = [a \supset_D b, a \supset_D b]$. □

4.7. Quantification over Lawson-Compact Sets

Definition Let $\mathbf{D}$, $\mathbf{A}_1$, ..., $\mathbf{A}_n$ be finitary domains. The quantifier operations $\exists^n$ and $\forall^n$ for the function domain $\mathbf{A}_1 \times ... \times \mathbf{A}_n \rightarrow \mathbf{TypeR_D}$, **are defined by**

$$\exists^1 f = < \sqcup_{x \in A_1 \dagger} f(x), \sqcap_{x \in A_1 \dagger} f(x)^- >$$

$$\exists^n f = \lambda\, y_2{:}A_2, ..., y_n{:}A_n\, .$$
$$< \sqcup_{x \in A_1 \dagger} f(x,y_2,...,y_n)^+ , \sqcap_{x \in A_1 \dagger} f(x,y_2,...,y_n)^- >$$

$$\forall^1 f = < \sqcap_{x \in A_1 \dagger} f(x), \sqcup_{x \in A_1 \dagger} f(x)^- >$$

$$\forall^n f = \lambda\, y_2{:}A_2, ..., y_n{:}A_n\, .$$
$$< \sqcap_{x \in A_1 \dagger} f(x,y_2,...,y_n)^+ , \sqcup_{x \in A_1 \dagger} f(x,y_2,...,y_n)^- > .$$

Theorem The quantifier operations $\exists^n$ and $\forall^n$ preserve totality: for any $f \in \mathbf{A}_1 \times ... \times \mathbf{A}_n \rightarrow \mathbf{Type_D}$ and $[y_2,...,y_n] \in \mathbf{A}_2 \times ... \times \mathbf{A}_n$ such that $f(x,y_2,...,y_n)$ is total for all $x \in A^\dagger$, the types $(\exists^n f)(y_2,...,y_n)$ and $(\forall^n f)(y_2,...,y_n)$ are total.

Proof Immediate from the definition of $\exists^n$ and $\forall^n$. □

In contrast to their strong totality properties, the quantifiers $\exists^n$ and $\forall^n$ are not necessarily continuous for some domain $\mathbf{A}_1$. The critical property of the domain that ensures continuity is Lawson-compactness. Fortunately, the domain of types $\mathbf{Type_D}$ over any finitary domain $\mathbf{D}$ is Lawson-compact.

Definition A subset S of a finitary domain $\mathbf{D}$ is *Lawson-compact* iff S is compact in the Lawson topology for $\mathbf{D}$. A finitary domain $\mathbf{A}$ is *Lawson-compact* iff the set $A^\dagger$ of total elements of $\mathbf{A}$ is a Lawson-compact subset of $\mathbf{A}$.

Lemma For any finitary domain $\mathbf{D}$, the domain of types $\mathbf{Type_D}$ is Lawson-compact.

Proof A routine verification. □

Theorem If $\mathbf{A}_1$ is Lawson-compact, then for all $n>0$ the operations $\exists^n$ and $\forall^n$ are continuous.

Proof Assume that we are given a directed set of continuous functions $\mathbf{F} \subseteq [\mathbf{A}_1 \times ... \times \mathbf{A}_n \rightarrow \mathbf{Type_D}]$. We must show that $\exists^n(\sqcup F) = \sqcup_{f \in F} \exists^n(f)$ and $\forall^n(\sqcup F) = \sqcup_{f \in F} \forall^n(f)$ For each $f \in F$, let $f^+{:}\mathbf{A}_1 \times ... \times \mathbf{A}_n \rightarrow \mathbf{Cl_D}$ and $f^-{:}\mathbf{A}_1 \times ... \times \mathbf{A}_n \rightarrow \mathbf{Op_D}$ denote the continuous functions defined by:

$$f^+(x,y_2,...,y_n) = f(x,y_2,...,y_n)^+$$
$$f^-(x,y_2,...,y_n) = f(x,y_2,...,y_n)^- .$$

Since $\exists^n f$ and $\forall^n f$ are $(n-1)$-ary functions, the continuity of $\exists^n$ and $\forall^n$ reduces to showing that for arbitrary elements y_2, ..., y_n in $\mathbf{A}_2 \times ... \times \mathbf{A}_n$, the sets E and A defined by

$$E = \{< \sqcup_{x \in A_1 \dagger} f^+(x,y_2,...,y_n), \sqcap_{x \in A_1 \dagger} f^-(x,y_2,...,y_n)> \mid f \in F\}$$
$$A = \{< \sqcap_{x \in A_1 \dagger} f^+(x,y_2,...,y_n), \sqcup_{x \in A_1 \dagger} f^-(x,y_2,...,y_n)> \mid f \in F\}$$

satisfy the equations

(4) $\sqcup E =$
$$< \sqcup_{x \in A_1 \dagger} (\sqcup F)^+ (x,y_2,...,y_n), \sqcap_{x \in A_1 \dagger} (\sqcup F)^- (x,y_2,...,y_n)>$$

(5) $\sqcup A =$
$$< \sqcap_{x \in A_1 \dagger} (\sqcup F)^+ (x,y_2,...,y_n), \sqcup_{x \in A_1 \dagger} (\sqcup F)^- (x,y_2,...,y_n)> .$$

The positive components of equation (4) are clearly identical because

$$\sqcup_{x \in A_1 \dagger} f^+(x,y_2,...,y_n) = \sqcup_{x \in A_1} f^+(x,y_2,...,y_n)$$

for every continuous function f. By an analogous argument, the negative components of (5) are identical.

On the other hand, proving the equality of the negative components of (4) and the positive components of (5) requires a more delicate analysis. The proof critically depends on the fact that $\mathbf{A}_1$ is Lawson-compact.

The infinite total elements of a Lawson-compact finitary domain $\mathbf{A}$ correspond to the infinite paths through an infinite binary tree T where each branch point at level n indicates whether or not the finite element with index n approximates the infinite total element. As a result, for any Scott-continuous function $f{:}\mathbf{A} \rightarrow \mathbf{B}$ (where $\mathbf{B}$ is an arbitrary finitary domain), a finite element y approximates f(x) for all total elements $x \in A^\dagger$ iff there exists a finite binary tree—derived from T by pruning subtrees—such that every path from the root to a leaf is either inconsistent (with all *total* elements) or includes y in the image of its sup under f. Otherwise, by Konig's Lemma, there is an infinite path in T denoting an element $z \in \mathbf{B}$ such that $y \notin f(z)$.

By employing this construction, we can prove the following critical lemma.

Lemma Let $\mathbf{A}$ and $\mathbf{B}$ be effective domains where $\mathbf{A}$ is Lawson-compact. The function $\bigcap_{A \rightarrow B}{:}[\mathbf{A} \rightarrow \mathbf{B}] \rightarrow \mathbf{B}$ defined by

$$\bigcap_{A \rightarrow B}(g) = \sqcap_{x \in A \dagger} g(x)$$

is continuous.

Proof of Lemma To prove the lemma, we need to introduce several definitions.

Definition Let $< a_i \mid i \in N >$ be the enumeration of A^0 in $\mathbf{A}$. A ***path*** π over $\mathbf{A}$ is a finite, non-empty sequence p_0, ..., p_n where each element p_i is either a_i or $\neg a_i$. A path π over $\mathbf{A}$ is ***totally-consistent*** iff there exists a ***total*** element $e \in \mathbf{A}$ that conforms with the constraints specified by π:

$$\forall j{:}0 \leq j \leq n\ [(a_j \in \pi \rightarrow a_j \subseteq e)\ (\neg a_j \in \pi \rightarrow a_j \not\subseteq e)$$

A path is ***totally-inconsistent*** iff it is not totally-consistent. The ***meaning*** of a totally-consistent path π is $\sqcup \pi_+$ where $\pi_+ = \{a_j \in \pi \mid 0 \leq j \leq n\}$. If S is a set of paths over A, the meaning of S (denoted $\sqcup S$) is the set $\{\sqcup \pi \mid p \in S$ and p is totally-consistent$\}$. There is an obvious one-to-one correspondence between paths over A and finite, non-empty paths in a complete, infinite binary tree T.

Definition A ***uniform binary tree*** W is a finite binary tree in which every internal node has two sons.

Definition Let y be an finite element approximating $\sqcap_{x \in A \dagger} g(x)$. A g-***witness tree*** W for y is a uniform binary tree such that every totally-consistent path π in W from a root to a leaf ***yields*** y under g: $y \subseteq g(\sqcup \pi)$.

The proof of the lemma breaks down into a series of three claims.

Claim 1 For any continuous function g: $\mathbf{A} \rightarrow \mathbf{B}$, every finite element in $(\bigcap_{x \in A \dagger} g(x))^0$ has a g-witness tree.

---

[5] The one-step-function $u \mapsto v$ where $u,v \in D^0$ is defined by λx . **if** $u \subseteq x$ **then** v **else** $\perp$. It is the least function f such that $v \subseteq f(u)$. The one-step-functions of $\mathbf{D} \rightarrow \mathbf{D}$ form a sub-basis for $\mathbf{D} \rightarrow \mathbf{D}$.

Proof of Claim Assume that some finite element y in $(\bigcap_{x\in A^t} g(x))^0$ does not have a g-witness tree. Let T_y denote the uniform tree obtained by deleting all nodes from T (the complete infinite tree) below any path that is totally-inconsistent or yields y. T_y must be infinite; otherwise T_y is a witness tree for y. By Konig's Lemma, T_y contains an infinite path κ. By the definition of T_y, no initial segment of κ yields y or is totally-inconsistent. Let K be the set defined by

$$K = \{\sqcup\pi \mid \pi \text{ is an initial segment of } \kappa\} .$$

Since κ is totally-consistent, K must be a directed subset of **A**. In addition, $\sqcup K$ must be a total element of **A** because every finite element in $\mathbf{A}^0$ is either below $\sqcup K$ or totally-inconsistent with it.

Since g is continuous and K is directed, $g(\sqcup K) = \sqcup_{k\in K} g(k)$. But by the definition of the path κ, y does not approximate g(k) for any element k in K. Hence, y does not approximate $g(\sqcup K)$, contradicting the assumption that y belongs to $(\bigcap_{x\in A^t} g(x))^0$. □ (of Claim 1)

Claim 2 Let g,h: $\mathbf{A} \to \mathbf{B}$ and $g \subseteq h$. If W_b is a g-witness tree for $b\in B$, then W_b is an h-witness tree for b.

Proof (of Claim 2) The claim follows immediately from the fact that $\forall\, a\in\mathbf{A}^0, b\in\mathbf{B}^0\ b \subseteq g(a) \to b \subseteq h(a)$. □

Claim 3 For any directed set G of functions in $\mathbf{A} \to \mathbf{B}$, $\bigcap_{\mathbf{A}\to\mathbf{B}} (\sqcup G) = \sqcup_{g\in G} \bigcap_{\mathbf{A}\to\mathbf{B}}\, g$.

Proof (of Claim 3) The function $\bigcap_{\mathbf{A}\to\mathbf{B}}$ is obviously monotonic. Consequently, all that we have to show is $\bigcap_{\mathbf{A}\to\mathbf{B}} (\sqcup G) \subseteq \sqcup_{g\in G} \bigcap_{\mathbf{A}\to\mathbf{B}}\, g$. Given an arbitrary element $b\in\mathbf{B}^0$ such that $b \subseteq \bigcap_{\mathbf{A}\to\mathbf{B}}(\sqcup G)$, we must prove that $b \subseteq \sqcup_{g\in G}\bigcap_{\mathbf{A}\to\mathbf{B}}\, g$. By Claim 1, b must have a $\sqcup G$-witness tree W_b. Let ω_b be the finite function determined by pairing the meaning of each consistent path in W_b with b. ω_b obviously approximates $\sqcup G$. Since $\omega_b \subseteq \sqcup G$ and ω_b is finite, there exists an element $h\in G$ such that $\omega_b \subseteq h$. By Claim 2, W_b must be h-witness tree as well as a $\sqcup G$-witness tree, implying that $b\in\bigcap_{\mathbf{A}\to\mathbf{B}}\, h \subseteq \sqcup_{g\in G}\bigcap_{\mathbf{A}\to\mathbf{B}} g$. □ (of Claim, Lemma, and Theorem)

Although the preceding theorem establishes that the quantifiers $\exists^n$ and $\forall^n$ are continuous, it says nothing about whether or not they are computable. Since both quantifiers involve infinite intersections, a naive approach to computing the functions clearly will not work. Fortunately, the witness tree construction used in the proof of continuity provides the critical trick required to compute the infinite intersections. The only obstacle is deciding when finite sets of basis elements are totally-inconsistent. Although the total-inconsistency of finite sets of basis elements is not decidable in general for effective domains, it is decidable for most domains of practical interest including the domain $\mathbf{Type}_D$ of interval types over a arbitrary effective domain **D**

Definition An effective domain **D** is *totally-effective* iff the total-consistency of arbitrary finite subsets of $\mathbf{D}^0$ is decidable.

Theorem For every effective domain **D**, the domain $\mathbf{Type}_D$ is totally-effective.

Proof A routine verification. □

Theorem For all $n \geq 1$, the operations $\exists^n$ and $\forall^n$ are computable if the domain $\mathbf{A}_1$ is Lawson-compact and totally-effective.

Proof Assume that we are given a computable function f:$\mathbf{Type}_D{}^n \to \mathbf{Type}_D$. We must show that the sets of finite elements approximating $\exists^n f$ and $\forall^n f$ are recursively enumerable. Since $\exists^n f$ and $\forall^n f$ are (n−1)-ary functions, the computability of $\exists^n$ and $\forall^n$ reduces to showing that for arbitrary computable elements y_2, ..., y_n in **D**, the objects ϵ and α defined by the equations

$$\epsilon = < \sqcup_{x\in Idl_D} f^+(x,y_2,...,y_n) \,,\, \sqcap_{x\in Idl_D} f^-(x,y_2,...,y_n) >$$

$$\alpha = < \sqcap_{x\in Idl_D} f^+(x,y_2,...,y_n) \,,\, \sqcup_{x\in Idl_D} f^-(x,y_2,...,y_n) > .$$

are computable.

The set of finite elements of $\mathbf{Cl}_D$ approximating the positive component of ϵ is clearly recursively enumerable since

$$\sqcup_{x\in Idl_D} f^+(x,y_2,...,y_n) = \sqcup_{x\in Type_D} f^+(x,y_2,...,y_n)$$

and f^+ is a computable function from $\mathbf{Type}_D{}^n$ to $\mathbf{Cl}_D$. Similarly, the negative component of α is computable.

Since **A** is Lawson-compact, a finite element $e\in\mathbf{Op}_D{}^0$ approximates ϵ^- iff a witness-tree T_e exists for e. Consequently, to compute the the set of finite elements of $\mathbf{Op}_D$ that approximate the negative component of ϵ, we enumerate all pairs $<e,U>$ where $e\in\mathbf{Op}_D{}^0$ and U is a uniform tree and check to see if U is a witness-tree for e. Since **A** is totally-effective, we can decide for each finite path ω in U whether or not it is totally consistent. Similarly, for every totally-consistent finite path π in U we can enumerate all of the finite elements e that approximate the image under of f of the meaning of π. Hence, if U is a witness-tree for e, we will eventually discover that fact by determining for each path that it is totally-inconsistent or includes e in the image of its meaning. The computation enumerates a finite element e as soon as it discovers a witness tree for it.

An analogous procedure will enumerate all of the finite elements $a\in\mathbf{Cl}_D{}^0$ such that $a \subseteq \alpha^+$. □ (of Theorem)

Notation The expressions $\exists\ t_1,...,t_n\ \tau$ and $\forall\ t_1,...,t_n\ \tau$, where $t_1,...,t_n$ are distinct variables, abbreviate the expressions $\exists^n\ \lambda\ t_1,...,t_n\ .\ \tau$ and $\forall^n\ \lambda\ t_1,...,t_n\ .\ \tau$, respectively.

Definition Let $\mathbf{A}_1$, ..., $\mathbf{A}_n$ be finitary domains. For each function domain $\mathbf{A}_1\times...\times\mathbf{A}_n \to \mathbf{A}_1\times...\times\mathbf{A}_m$ $n\geq m\geq 1$, the fixed point operator $\mu_{m,n}$ is defined by:

$$\mu_{n,n}\ f = \mathbf{Y}\ \lambda\ [x_1,...,x_n]\ .\ f(x_1,...,x_n) = \mathbf{Y}\ f$$

$$\mu_{m,n}\ f = \lambda[x_{m+1},...,x_n]\ .\ \mathbf{Y}\ \lambda[x_1,...,x_m]\ .\ f(x_1,...,x_n)\ .$$

where **Y** denotes the standard least fixed point operator.

Remark $\mu_{m,n}$ is simply a notational generalization of the **Y** operator that accommodates free variables in the expression denoting the input function.

Lemma For all $n\geq m\geq 1$, $\mu_{m,n}$ is continuous. If **D** is effective, then $\mu_{m,n}$ is computable.

Proof An immediate consequence of the corresponding properties of **Y**. □

4.8. Solving Recursive Type Equations

Definition Let **D** be an effective domain. A *system of type equations* Σ *over* **D** is a set of equations

$$\{\ t_1 = \tau_1,\ ...,\ t_n = \tau_n\ \}$$

where $t_1,...,t_n$ are distinct variables and $\tau_1,...,\tau_n$ are expressions constructed from continuous operations on $\mathbf{Type}_D$ and the variables $t_1,...,t_n$. The *function* σ:$D^n \to D^n$ determined by Σ is defined by the equation

$$\sigma = \lambda\ [t_1,...,t_n]\ .\ [\tau_1,...,\tau_n]\ .$$

Σ is *computable* iff σ is a computable function. A *solution* to the system of type equations Σ is an n-tuple $[d_1,...,d_n]$ of elements of **D** such that $\sigma([d_1,...,d_n]) = [d_1,...,d_n]$.

Theorem Every set of type equations Σ over a finitary domain **D** has a least solution consisting of the tuple of intervals $\mathbf{Y}\sigma$ where σ is the function determined by Σ. Moreover, if all the functions appearing in Σ are computable, then the solution $\mathbf{Y}\sigma$ is computable.

Proof The theorem is an immediate consequence of the definition of the least fixed point operator Y and the fact that every closed expression constructed from computable functions denotes a computable function. □

Although the preceding theorem shows that every system of recursive type equations has a least solution, that solution is not necessarily total. Since programmers are almost always interested in defining types that are total, the practical value of the theory of types as intervals rests on whether the least solutions to type equations are total in typical cases. Fortunately, the situation is roughly analogous to that which prevails in practical programming languages: although type definitions (programs) are not necessarily total, those that programmers typically write—even when they contain errors—typically are.

The following examples illustrate the potential problem.

Definition Let Σ be an n-ary system of type equations over a data space **D**, and let σ denote the function determined by Σ. An *ideal solution* of Σ is a solution that is a total element of $\mathbf{Type_D}$, i.e. a tuple of ideal images $[I_1,\ldots,I_n]\in \mathrm{Idl}_D{}^n$ su ch that $\sigma([I_1,\ldots,I_n]) = [I_1,\ldots,I_n]$.

Observation A system of type equations may not have an ideal solution.

Proof A simple counterexample is the type equation

$$C = \neg C$$

over the flat domain $\mathbf{Bool} = \{\text{true, false}, \bot\}$ where $\neg$ denotes the computable function defined by the equation

$$\neg([b,B]) = [\{\bot\}\cup\bar{B},\{\bot\}\cup\bar{b}]\ .$$

The preceding equation has no ideal solution because it complements the set of total elements on each side of the interval. The only interval solution is the least interval $[\{\bot\},\mathbf{Bool}]$. □

Observation A system of type equations may have a unique ideal solution that is distinct from the least interval solution.

Proof Let **Bool** denote the same flat Boolean domain as above and let if-then-else and is-defined denote the standard ternary conditional and unary definedness functions, respectively, on BBool. The type equation

T = *if* is-defined(T) *then* **Bool** else T

has the unique ideal solution $[\mathbf{Bool},\mathbf{Bool}]$ but the least interval solution is the least interval $[\{\bot\},\mathbf{Bool}]$. □

Fortunately, these pathologies do not often arise in the context of standard domains because standard domains form a very special kind of metric space that ensures the solutions to most type equations are total. In fact, we can prove a theorem that asserts that the least solution to every formally contractive system of equations is total. The proof of theorem is based on essentially the same metric space analysis that MacQueen, Plotkin, and Sethi used to prove the *existence* of solutions to formally contractive systems of equations over the space of ideals. As groundwork for this theorem, this section of paper develops a metric space theory for intervals based on the corresponding theory for ideals presented in in [MacQ84a].

The most surprising feature of the new theory is that the natural generalization of every theorem in the original theory holds in the new theory—even though there are systems of equations that are contractive on ideals but not on intervals. The explanation is that the basic type operations over a standard domain satisfy are contractive on intervals—not just ideals (a weaker property). Consequently, the syntactic notion of *formal contractiveness* proposed in [MacQ84a] not only ensures that a system of equations is contractive on ideals, but on intervals as well.

The metric space of intervals is more elegant and robust than the original for two reasons. First, it attaches stronger semantic content to the notion of formal contractiveness. Second, it has simpler, sturdier formal foundations because is formulated entirely in terms of continuous functions on finitary domains. In contrast, the theory of ideals involves discontinuous functions and domains (such as the set of all functions over a finitary domain) that are not finitary. As a consequence, the solutions to formally contractive systems of type equations are computable in the framework of intervals, but not in the framework of ideals.

Definition A *ranked domain* $<\mathbf{D},r>$ is a pair consisting of a finitary domain and a *rank function* r mapping the finite elements of **D** into the natural numbers such that (i) $r(\bot) = 0$, and (ii) $r(x) > 0$ for all $x\neq\bot$.

We will frequently use the following three mechanisms for constructing composite ranked domains from simpler ranked domains. The underlying intuition is that the rank of a finite element in a domain defined by an equation should correspond to the index in the ascending chain of approximate solutions where the element first appears.

Definition Let $\Delta = <\mathbf{D},d_A>$ and $\Gamma = <\mathbf{G},g_B>$ be ranked domains.

(i) The ranked Cartesian product domain $\Delta\times\Gamma$ is the pair consisting of the domain $\mathbf{D}\times\mathbf{G}$ and the rank function r defined by

$$r(<x,y>) = \max\{d(x),g(y)\}\ .$$

(ii) The ranked function domain $\Delta\rightarrow\Gamma$ is the pair consisting of the function domain $\mathbf{D}\rightarrow\mathbf{G}$ and the rank function r defined by

$$r(f) = \max\{\ \max(x,y) \mid x\in D^0, y\in G^0 \quad x\mapsto y \text{ is essential in } f\ \ y \subseteq f(x)\}$$

where a one-step function $x\mapsto y$ is *essential in the finite function* f iff $\forall u\in D^0,\ v\in G^0\ u\mapsto v$ implies $u\mapsto v\ \subseteq\ x\mapsto y$.

(iii) The *ranked domain of types* $\mathbf{Type}_\Delta$ is the pair $<\mathbf{Type_D},r>$ consisting of the domain $\mathbf{Type_D}$ and the rank function defined by

$$r([t,T]) = \max\{d(d) \mid d \text{ is maximal in a}\ \ d \text{ is minimal in } \bar{A}\}$$

This definition is meaningful because the finiteness of $[a,A]$ implies that the maximal elements of a and minimal elements of $\bar{A}$ must be finite elements of **D**.

Definition Let $\Delta = <\mathbf{D},r>$ be a ranked domain. For any two distinct elements $x,y\in\mathbf{D}$, a *witness* for (x,y) is a finite element $w\in\mathbf{D}$ such that $w\subseteq x$ but $w\not\subseteq y$ or vice-versa (since x and y are distinct, such an element must exist). The *affinity* of two distinct elements $x,y\in\mathbf{D}$ (denoted $|x,y|$) is the least rank of a witness for (x,y). The *metric space* determined by Δ is a pair $<D,d>$ consisting of the domain universe **D** and the function d mapping $\mathbf{D}^2$ into the real numbers defined by

$$d(x,x) = 0$$

$$d(x,y) = 2^{-|x,y|} \quad \text{if } x,y \text{ are distinct}\ .$$

d is called the *rank metric* determined by r.

Remark In this paper, we will confine our attention exclusively to rank metrics. For economy of notation, we will universally use the symbol d to denote the rank metric corresponding to a ranked domain. The intended domain and rank function should be clear from context.

Definition Let $<\mathbf{A},a>$, $<\mathbf{B},b>$ be ranked domains. A function $f:\mathbf{A}\rightarrow\mathbf{B}$ is *contractive on* $C \subseteq A$ iff

(i) f *preserves totality*: for every total element $a\in\mathbf{A}^\dagger$ $f(a)\in\mathbf{B}^\dagger$.

(ii) $\forall x,y\in C\ d(f(x),f(y)) \leq r * d(x,y)$ for some constant $r < 1$.

The function f is *non-expansive on* $C \subseteq \mathbf{A}$ iff f preserves totality and condition (ii) holds for some constant $r \leq 1$.

Definition Let $<A_1,a_1$ ..., $<A_n,a_n>$, and B be ranked domains and let $C \subseteq A_i$. An n-ary function $f:A_1 \times \ldots \times A_n \rightarrow B$ is *contractive* is *contractive* (*non-expansive*) *on* C *in argument* i iff

(i) f *preserves totality*.

(ii) The curried function f_i defined by

$$\lambda x_i . \lambda [x_1, ..., x_{i-1}, x_{i+1}, ..., x_n] . f(x_1,...,x_n)$$

is contractive (non-expansive) on C.

Lemma An n-ary function is contractive (non-expansive) in each argument i iff the corresponding unary function is contractive.

Proof An immediate consequence of the definitions of the Cartesian and function domain rank functions and the rank metric. □

Definition Let $<D,r>$ be a ranked domain, and let $<\mathbf{Type}_D,t>$ be the ranked domain of types determined by $<D,r>$. A continuous function $f:\mathbf{Type}_D{}^m \rightarrow \mathbf{Type}_D{}^n$ is *ideal-contractive* (*interval-contractive*) iff f is contractive on $\mathrm{Idl}_D{}^m$ $(\mathrm{Type}_D{}^m)$.

Lemma If a function f is interval-contractive, then it is ideal-contractive.

Proof An immediate consequence of the definitions. □

Theorem Let $<D,r>$ be a ranked domain and let Σ be a system of type equations over **D**. If the function σ determined by Σ is interval-contractive, then Σ has a unique solution which *must be total*.

Proof By the Banach fixed-point theorem [Bana22], σ has a unique ideal solution and a unique interval solution. Since every ideal solution is an interval solution, the two must coincide. □

Definition Let **D** be a subspace over a universal domain **U** determined by a domain equation Σ and let σ denote the function mapping subspaces to subspaces determined by Σ. The *constructive rank r* of a finite element $d \in D$ is the least k such that $\sigma^k(\bot)(d) = d$. The *constructive metric* on $\mathbf{Type}_D$ is the rank metric determined by the constructive rank function *r*.

Although there are many type constructors that are not contractive or non-expansive on intervals under any metric, the basic type operations $\{ \supset, \times, +, \cup, \cap, \forall, \exists, \mu \}$ on a *standard domain* **D** are very well-behaved in this regard.

Definition A *standard ranked domain* $<D,r>$ is a standard domain **D** together with the constructive rank function *r* determined by the domain equation defining **D**.

Theorem In the domain of types determined by a standard ranked domain $<D,r>$, the type constructors $\{\supset, \times, +\}$ are interval-contractive. Similarly, the type constructors $\{\cup, \cap\}$ are non-expansive on intervals. The higher order operations $\{\exists^n, \forall^n\}$ preserve the contractiveness (non-expansiveness) of a function $f:\mathbf{Type}_D{}^n \rightarrow \mathbf{Type}_D$ in each argument position $1 < i \leq n$. Similarly, if f is contractive in its first m arguments, then the fixed point operator $\mu_{m,n}$ preserves the contractiveness (non-expansiveness) of f in each argument position $i > m$.

The definition of formal contractiveness of type expressions presented in [MacQ84a] is based directly on the precise analog (for ideals) of preceding theorem and the obvious properties of contractive/non-expansive functions under composition and tupling. With the exception of a simple extension to accommodate the mutually recursive fixed-point operator $\mu_{m,n}$, the definition for the theory of intervals is identical to that presented in [MacQ84a]. Consequently, the following theorem holds.

Theorem If a type expression τ with free variables t_1, ..., t_n is formally contractive in the variable t_i, then the function $\lambda [t_1, ..., t_n] . \tau$ is interval-contractive in its ith argument.

Definition A system of type equations $\{ t_1 = \tau_1, ..., t_n = \tau_n \}$ is *formally contractive* iff each expression τ_1, ,..., τ_n is formally contractive in each of the variables $t_1,...,t_n$.

Since all of the operations in formally contractive type expressions are computable over the domain of intervals, the following corollary is an immediate consequence of the definition of formal contractiveness and the fact that an interval-contractive system has a unique solution.

Claim If Σ is a *formally* contractive system of type equations, then the ideal solution is computable.

Proof By the preceding theorem, the ideal solution must be the least interval solution, which is obviously computable. □

5. Generalizing the Formal Theory

In Section 4, we focused our attention on showing that all of the basic interval type operations are computable and tha the formulation of types as intervals subsumes the formulation of types as ideals. Now we briefly shift our attention to discussing constructions within the theory of types as intervals that have no analog (to the author's knowledge) within the theory of types as ideals.

The primary advantage of formalizing types as intervals is that it supports a richer class of type definitions and type operations including programmer defined type constructors and extended forms of quantification—all of which are computable. Since interval types are ordinary data values and all the basic operations on intervals are computable, a system of type equations is simply a stylized form of higher order recursive program. In this framework, there is no reason to limit the objects defined by a system of type equations to just types; the type system accommodate the definition of arbitrary computable objects and operations which may be useful in declaring the types of program operations.

Two important illustrations of this extra power—programmer defined type constructors and generalized quantification— were discussed briefly in Section 3.4. Only two minor extensions to the formal theory are required to justify these generalizations.

First, the domain of interval types $\mathbf{Type}_D$ must be included as one of the disjoint components in the equation defining the domain **D**. This extension makes types part of the domain of values **D** that the programmer can access within programs. It also makes the ideal of types $[\mathbf{Type}_D, \mathbf{Type}_D]$ into a type that can be manipulated in type definitions and programs. Since the interval type constructor is a very simple function, none of the properties of the domain (such as total-effectniveness) are compromised by its addition.

Second, the formal definition of type quantification must be generalized to accommodate quantication over the total elements $t^\dagger$ (relative to the domain **D**) of any total type $[t,t]$ where $t^\dagger$ is Lawson-compact. This extension provides a formal foundation for generalized quantifiers (that take any Lawson-compact type as a parameter specifying the quantification set) discussed in Section 3.4. Since it is decidable for any computable total type $[t,t]$ whether an arbitrary finite element e belongs to t, an essentially identical witness-tree construction works in the general case. The only difference is that the decision procedure for determining the total-consistency of paths uses the negative information embedded in the total type t as well as the information on the path to determine whether or not the path is consistent. This strategy reduces the total-consistency of finite paths over the type t to the total-consistency of finite paths in the parent domain **D**.

Although a systematic classification of the closure properties of various type constructors with respect to Lawson-compactness is an open research problem, it is easy to show that all total subtypes of any type that is freely generated by non-strict constructors is Lawson-compact. Moreover, it is clearly possible to write a higher order program that implements the required construction. If a programmer applies this program to a type that is not Lawson-compact, the function will still produce a well-defined result (possibly divergence); it simply does not match the infinitary definition of quantification.

6. Directions for Future Research

Although the theory of types as intervals is mathematically elegant and theoretically instructive, its value as the basis for a practical type system has not yet been demonstrated. For this reason, a research group at Rice is designing a new version of the executable specification language TTL [Cart80] to support interval types. The next stage in the research project will be study the problem of type inference much more carefully and build a heuristic type checking system for the new version of TTL.

7. Acknowledgments

I am indebted to Alan Demers for sharpening my understanding of domain theory, to Dana Scott for providing gentle guidance and encouragement, to Richard Statman for helping debug some of the critical definitions and proofs, and to Gordon Plotkin for focusing my attention on the problem of type inference systems for interval types.

8. References

[ADJ77] ADJ (J. Goguen, J. Thatcher, E. Wagner, J. Wright). Initial Algebra Semantics and Continuous Algebras, *JACM* **24**, 68-95.

[Bana22] Banach, S. "Sur les operations dans les ensembles abstraits et leurs applications aux equations integrales," *Fund. Math. 3* (1922), pp. 7-33.

[Burs84] Burstall, R. and B. Lampson, The Cedar Kernel Language, unpublished draft, Xerox Palo Alto Research Center, 1984.

[Cart80] Cartwright, R. A Constructive Alternative to Axiomatic Data Type Definitions. *Proceedings 1980 LISP Conference*, Stanford, 1980.

[Cart82] Cartwright, R. and J. Donahue. The Semantics of Lazy (and Industrious) Evaluation. *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 253-264.

[Ende72] Enderton, H.B. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Gier80] Gierz, G. et.al. *A Compendium of Continuous Lattices.* Springer-Verlag, Berlin, 1980.

[Gord77] Gordon, M., R. Milner and C. Wadsworth. Edinburgh LCF. CSR-11-77. Computer Science Department, Edinburgh University.

[Gutt78] Guttag, J. V. and J. J. Horning. The Algebraic Specification of Data Structures, *Comm. ACM* **20**, 396-404.

[Hind69] Hindley, R. The Principal Type Scheme of an Object of Combinatory Logic. *Trans. AMS 146*, pp. 29-60 (December 1969).

[MacQ82] MacQueen, D. and R. Sethi. A Semantic Model of Types for Applicative Languages. *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 243-252

[MacQ84a] MacQueen, D., G. Plotkin, and R. Sethi. An Ideal Model for Recursive Polymorphic Types. *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 165-174.

[MacQ84b] MacQueen, D. Modules for Standard ML. To appear in *Conference Record of 1984 ACM Symposium on LISP and Functional Programming.*

[McCr79] McCracken, N.J. An Investigation of a Programming Language with a Polymorphic Type Structure. Ph.D. Thesis, Dept. of Computer and Information Science, Syracuse University, June 1979.

[Miln78] Milner, R. A Theory of Type Polymorphism for Programming, *JCSS 17(3)*, pp. 348-375.

[Nels79] Nelson, G. and D. Oppen; "Simplification by Cooperating Decision Procedures," *ACM TOPLAS* **1**, pp. 245-257.

[Plot78] Plotkin, G. T^{ω} as a Universal Domain. *Journal of Computer and System Sciences 17* (1978), pp. 209-236.

[Reyn74] Reynolds, J. Towards a Theory of Type Structure. *Programming Symposium Paris*, Lecture Notes in Computer Science 19, Springer-Verlag, Berlin, 1971.

[Scot76] Scott, D. Data Types as Lattices. *SIAM J. Computing 5*(1976), pp. 522-587.

[Scot81] Scott, D. Lectures on a Mathematical Theory of Computation. Technical Monograph PRG-19, Oxford University Computing Laboratory, Oxford.

[Scot83] Scott, D. Domains for Denotational Semantics. Technical Report, Computer Science Department, Carnegie-Mellon University, 1983.

[Sham77] Shamir, A. and W. Wadge. Data Types as Objects, *Automata, Languages, and Programming, Fourth Colloquium, Turku*, Lecture Notes in Computer Science 52, Springer-Verlag, Berlin, 1977.

[Stoy77] Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.