Continuous Functions as Models for Programs: Mathematics and Applications *

Michael Bukatin †

April 15, 1996

Abstract

We present evidence of continuous nature of many computational processes. We argue that, therefore, powerful techniques of continuous mathematics should be applicable in the studies of computational processes and, moreover, in the design of tools for software engineers.

We see the potential for both theoretical and applied advances along these lines, and we would like to incorporate both into our Thesis. Ideally theoretical advances should lead at once to the new applications, however, given the state of the art it does not seem feasible.

Therefore we opt for the research program consisting of two parts. Part 1 consists of advancing the mathematics of topological models of computation. It does take potential applications into account, but we do not expect to incorporate any such applications in this Thesis. The internal mathematical motivations will mostly guide the theoretical part.

Part 2 consists of applying the existing body of knowledge to software engineering, and, in particular, to the design and implementation of programming languages. We have found that paradigms employed in the design of most modern "academic" programming languages seem to be incompatible with mathematical models of computations we prefer. Therefore, we will perform experimental explorations in the design and implementation of programming languages. We will take into account both their conformity to the desired mathematical properties and potential practical usability. We will restrict ourselves to small prototypes of the real systems, so we do not expect to incorporate any "commercially usable" systems in this Thesis.

Preliminary results of the author are reviewed and the Appendices contain more technical details. The Conclusion contains the list of problems the Thesis will try to address.

*PhD Thesis Proposal. The list of references is not quite complete yet. Supported by NSF Grant CCR-9216185 and Office of Naval Research Grant ONR N00014-93-1-1015

†Department of Computer Science, Brandeis University, Waltham, MA 02254, USA. *bukatin@cs.brandeis.edu*

Structure of This Text

This text is written so that it could be read sequentially by someone who is not a specialist in this field. Sections 1-4 describe general motivations for the author's studies and objectives of this Thesis. Section 5 reviews the latest developments in mathematics of domains. Section 6 discusses the criteria for programming language design. Section 7 reviews author's preliminary results. Conclusion (section 8) contains the list of open problems. Appendix 1 describes technical details of our results in mathematics of domains. Appendix 2 contains details of author's first steps in programming language design.

If the reader is mainly interested in mathematics of domains, the recommended order of reading is as follows. Subsection 2.3 briefly describes the ideas of domain theory. Subsection 5.1 informally introduces information systems. Subsection 7.1 informally reviews our preliminary results in information systems, and Appendix 1 contains technical details not covered by [8, 7]. Subsections 5.2 and 7.2 informally discuss analysis on domains.

If the reader is mainly interested in the issues of programming language design, the recommended order of reading is as follows. Section 6 discusses the problems. Subsection 7.3 and Appendix 2 describe the author's preliminary steps to the solution of some of these problems.

Contents

1 Terminology

The notion of *approximation* plays a key role in this discourse. We must distinguish between two versions of this notion - informal approximation, and formal or strict approximation. *Formal approximation* gives, in general, incomplete information about the object being approximated, but it never gives any false information. *Informal approximation* just means that the approximation is, hopefully, close to the object being approximated, but it can contain information which is not correct about this object. In the main text we always use the notion of formal approximation explicitly, and the unqualified term "approximation" means informal approximation. In appendices "approximation" means formal approximation.

Also we use the term "Curry-Howard type systems" as a collective name for type systems which satisfy Curry-Howard isomorphism [21].

2 Background on Continuity in Computations

Here we present evidence of continuous nature of computations.

2.1 Continuity in software engineering

It seems appropriate to start with a real life observation before dealing with mathematical models. Virtually all software products used today contain a number of bugs - leading to refusal to work, crashes, and incorrect processing of data. Yet for a usable software product these unpleasant effects are observed relatively infrequently - most of the time the product works satisfactory. What we should say here is that the real product is sufficiently close to the ideal product, close enough to be used in lieu of the unavailable ideal product. This is a phenomenon of continuity. This continuity results from the process of testing and amending the software and is crucial for our ability to use any software at all. Moreover, it is quite possible that the ideal product cannot exist at all (e.g. if the specifications are for an undecidable problem), yet we still can consider models which contain representatives for such ideal products and for their real-life approximations.

The author so far has not encountered studies dealing with this phenomenon mathematically. Software reliability models do not seem to qualify, because they always abstract from the nature of the particular software involved and just study the dynamics of error rates. We would like to see the difference between the ideal software and its real-life approximation expressed semantically, error classification and rates being built on top of such semantical structure. We think that such approach to software design is potentially more promising than the concept of fully verified software.

2.2 Continuity in constructive mathematics

Another evidence of continuity of computations comes from effective versions of mathematical analysis (constructive, recursive, or intuitionistic analysis - e.g. [5, 23]). All constructive functions from constructive real numbers to constructive real numbers must be continuous. It is impossible to properly treat computations at the points of discontinuity. If one has to deal with non-continuous functions, they have to be partially defined - points of discontinuity cannot belong to their domain (although see [10]).

2.3 Continuity in denotational semantics of programming languages

The most important and mathematically rich example of continuous approach to computation is denotational semantics (see, for example, [36, 35, 38, 40, 42, 4]). We briefly introduce its main ideas here.

Denotational semantics of a formal language is a map S from the space of formal texts to the certain space of meanings (semantic space). $S[[P]]$ is a meaning of the formal text P; equivalently we say that P denotes $S[[P]]$.

To provide denotational semantics of the λ-calculus it was necessary to solve equations $D = [D \to D]$ for semantic spaces - the task which looked quite non-trivial and which does not have non-trivial solutions if we interpret $[D \to D]$ as the set of all functions from D to D. Topological ideas gave the solution [37, 40].

Continuity in computations always relates closely to the idea of formal approximation. Interval numbers formally approximate one another, partially defined functions do the same, etc. The relation of formal approximation usually is a partial order, usually denoted as $\sqsubseteq$. Properties vary in different approaches, but the existence of the least defined element and of lowest upper bounds of increasing chains (limits of sequences of better and better formal approximations) is usually required. Topology then is introduced in such a way, that the continuous functions are monotonic functions preserving limits of increasing chains. The resulting topological spaces are called *domains*.

It is natural to require from correctly defined computational processes that the better formal approximation of the input is known, the better formal approximation of the output should be computed (monotonicity). Also well defined computational schemes (e.g. in numerical methods) tend to preserve limits of formal approximations - we can get the result with an arbitrary small error if we take the input data with sufficient precision.

Thus continuous functions above are good candidates to serve as models of computable functions. The main paradigm of the field is: "continuous" is the right mathematical model for "computable". The resulting topological spaces possess a number of remarkable properties. Especially important for computer science applications are theorems of fixed points for functions and functors. The existence of canonical fixed points of continuous functions justifies all recursive definitions of functions. The existence of canonical fixed points of continuous functors justifies all recursive definitions of types; in particular, they give the solution of the equation $D = [D \to D]$, which is interpreted as homeomorphism, and $[D \to D]$ - as the space of all continuous functions from D to D in pointwise topology.

During the 25 years which elapsed since the introduction of the denotational semantics and its domains, they were applied to a number of programming languages and to other computational problems, and their mathematics underwent significant developments. We do not give the full review here, but the relevant developments will be introduced in the text below as needed.

3 Ultimate Goal

Here we give the description of what the author considers to be the ultimate goal of the field [1].

The school of formal verification of software maintains that in the future all important software will be precise and absolutely correct [13, 25]. The software

[1]The author believes that the formulation of this is his original contribution, he never encountered any similar formulation elsewhere.

engineers know better - the cost of obtaining absolutely correct and adequate software at best increases prohibitively fast with the increase in the complexity of the problems. In the worse cases ideal solution might be unachievable at finite cost, or even certain good approximations might be unachievable at finite cost.

Therefore, all real software development in the world - whether being done "by hand" (well, with the assistance of compilers and debuggers), or being done in semi-automated and automated higher-level IDEs and machine learning systems - has the same real goal: finding a reasonable practically acceptable approximation to the ideal solution [2]. The developers might have very imprecise ideas about their ideal solution or about approximation criteria, or they might believe that their goal is to produce absolutely correct software, or they might not care about the nature of their activities aside from the side effect of generating the paycheck. Regardless, in reality they are forced to build an approximation.

Despite the lack of precise information about approximation criteria, this problem has some mathematical relatives - the optimization problems, the problems arising in the calculus of variations, etc. Even when we solve precise mathematical problems we are often forced to mix well understood schemas with ad hoc methods. But whenever the convenient mathematical structure is present, such structure helps us to develop powerful well understood schemas. The experience tells us that such structure must belong to the realm of continuous mathematics - the problems of discrete optimization are much less tractable, and continuous mathematics provides much more powerful tools. On the other hand, even when we know approximation criteria they are often not effective enough and have to be replaced with heuristic approximations of these approximation criteria - so heuristic approximation criteria are nothing new.

So the experience tells us that finding approximations in the realm of continuous is much easier than in the realm of discrete, and it tells us that almost all software engineering, be it ordinary software design and development or machine learning, is about finding approximations. Therefore, the ultimate goal of the "Continuity in computations" field is to find methods facilitating the approximation building processes inherent in the software development, hopefully along the lines employed in building approximations in the problems of ordinary continuous mathematics.

4 Objectives of the Thesis

I want this Thesis to satisfy the following three main constraints. It should be a sound mathematical research, it should have practical relevance, and it should be defended by May 1997.

While the ultimate goal of the field would satisfy the first two constraints, there is no chance it can satisfy the third one. Not only such project would be

[2] Until recently the author believed this to be true only with respect to machine learning and considered the formal verification to be the best approach to "hand-made" software.

too ambitious; the necessary mathematics is unfortunately not well developed yet. Optimization methods in continuous mathematics require more than just topological structure. They require various structures of analysis which are usually built over ordinary metric topology (e.g. [34, 43]). Thus we need analysis over domain topology.

Only the first steps of building analysis over domain topology were made. We will give their overview in the next section of this Proposal. We cannot hope that within 1 year analysis over domain topology will become sufficiently developed to facilitate the process of software engineering.

Therefore we opt to divide our Thesis into two parts. The first part is mathematical research directed towards better understanding of domains, and hopefully contributing towards development of analysis over domains.

The second part is the application of the existing body of knowledge to the problems of software engineering along well known lines. The ultimate usefulness of denotational semantics was originally attributed to its ability to be a guiding force in the design of new programming languages. Unfortunately, the bulk of the applications was to describe the semantics of existing, and inevitably faulty, languages, which had considerably limited the power of the approach. We will try to remedy this situation somewhat and to explore desirable properties of programming languages. We will try to incorporate implementation issues and practical usability concerns into this part as well.

5 Mathematics of Domains - Latest Developments and Problems

5.1 Information systems

We believe that information systems [38, 30, 31] is the most important development in the domain theory since its inception in the late sixties and early seventies. This development allows us to use finite and understandable methods to study domains, lowers the level of abstraction, and provides necessary intuition.

Here we sketch the main ideas of the information systems approach. One starts with a *logical calculus of elementary statements*. One can form finite conjunctions of elementary statements and the usual notions of consistency and entailment are introduced. Such logical calculus is called an *information system*. Then we say that the *domain* corresponding to this logical calculus consists of its *theories*, i.e. consistent, deductively closed sets of elementary statements. The relation of *formal approximation* is ordinary *set-theoretical inclusion* of theories.

This formalizes the idea that an object x is a formal approximation of an object y iff all information known about x is known about y. Domain elements are identified with the information known about them under this approach.

The continuous functions are represented by *input-output entailment relations*. In general such relation connects two different logical calculi, A and B, and specifies how to infer the information about an output value of a function from the information about the argument. The technical term *approximable mapping* is often used instead of *input-output entailment relation* - it is not a very good terminology, especially because these relations are not functions, but it is used too widely to ignore it. A *continuous function* then just maps an argument (a theory in A) to the set of all elements which are inferable from the argument via the corresponding approximable mapping; this set turns out to be a theory in B.

The *functional space* is, in fact, a domain of input-output entailment relations, i.e. sets of such relations turn out to be sets of theories in the appropriate logical calculi (*functional information systems*). Observe, that continuous functions are not elements of a functional space in this approach, instead the input-output entailment relations representing them are.

Somewhat unexpectedly, domain equations are solvable precisely, and not just up to isomorphism under this approach. This means that, for example, we can find a solution of $D = [D \to D]$, such that the logical calculus giving the solution has precisely input-output entailment relations from itself to itself as theories.

Information systems approach is not sufficiently well known and used - my empirical observation is that about a half of researchers in the field do not use information systems, mainly because they do not know them well enough. This often leads to studies which are more difficult to understand and use than they would be if the information systems were utilized. The author's experience is that information systems help to obtain useful results as well [8, 7, 9] (see also subsection 7.1 and Appendix 1). One seemingly unexplored direction is to study information systems as programming languages - they seem to be well suitable for distributed non-deterministic implementations.

Original information systems were applicable only to algebraic Scott domains, i.e. algebraic lattices, where we are allowed to remove the top element when it is algebraic. We will just talk about lattices below for simplicity of speech, but we will always mean that algebraic tops can be removed.

There are two approaches to generalize information systems so that they would yield continuous and complete lattices. One was invented by Sazonov and Sviridenko [33] and involves infinitary logic. Another was invented by Hoofman [26] and further extended by the author [9] (see also Appendix 1). It involves non-reflexive logic.

Also there are studies generalizing information systems to non-lattice structures, like SFP domains [3]. It is necessary to include finite disjunctions of elementary statements in the logical calculi for this purpose.

We should also mention the work of Edalat and Smyth [18, 19] on information categories.

5.2 Analysis

The author is aware of two major developments in this direction, though in fairness we should say that analysis on domains is only making its first steps.

The first of such developments is *generalized metrization*. R.Kopperman explained how to obtain any topology via a generalized metric [28]. Then R.Flagg and R.Kopperman developed more intuitive generalized metrics for various classes of domains [29, 20].

A lot remains to be done along these lines. First, the purpose of generalized metrics is to make abstract topology easier, in particular, to make the notions of continuity and convergence more intuitive. The author does not think that current presentations of generalized metrics for domain topology have already achieved this goal. Another necessary study is to choose a reasonable language equipped with denotational semantics and compute generalized distances between particular pairs of programs. Then we will be able to see what is expressible via such distances.

The second of these developments is the *connection between domains and the theory of measure and integration* discovered by A.Edalat [15, 17]. The main emphasis of this development is so far on the applications of domains to the problems of computational measure theory (including fractals and integration), but some applications to computer science theory were developed as well (e.g. [16]).

What is necessary to do is to develop the measure theory for domains to such degree, that we can solve the problem of integration of a continuous function from a domain with a measure to a domain representing numbers with respect to this measure, and also the problem of choosing a random or pseudo-random element from a neighborhood in a domain with respect to a specific measure. We also need equivalents of "natural" measures for domains in the same sense as the n-dimensional volume is a "natural" measure for an Euclidean space.

5.3 Need for simplicity

One of the reasons denotational semantics is underused in practice is that many papers are too difficult to understand; intuition behind the definitions, theorems and their proofs is well hidden, and the level of complexity is high. The field of semantics of programming languages was developed in the late sixties and early seventies with the purpose to eventually help software engineers to control inherent complexity of programs. There are two (alas not mutually exclusive) ways in which mathematical models can deal with this complexity. They can express what happens using non-trivial ideas which would allow to achieve simplification, or they can just incorporate this complexity. The second way is unlikely to lead to mathematical models which would help to control the complexity of software engineering. We should try to apply the guidelines similar to the ones on what constitutes a good programming style to our studies of mathematical models of computation, if we were to hope that our results would eventually be applicable to software engineering.

In particular, we refrain from using the language of the theory of categories except in the cases where categorical ideas give obvious benefits in terms of intuition and clarity of presentation.

6 Applications: Design and Implementation of Programming Languages

At the current state of the theory it does not seem possible to apply ideas of continuity directly to programming and programming languages. Instead what is applicable is the idea that *programs mean mathematical functions*, and two corollaries of continuity - correctness of recursive function definitions and correctness of recursive type definitions. In particular, we will try to address the question whether we can have truly functional programming languages, i.e. languages which properly reflect the idea that programs mean mathematical functions. We argue that the modern functional programming languages are not quite adequate in this respect, and that further efforts are necessary in this direction.

6.1 How denotational semantics is used

The radical changes in the methodology of programming in the end of 1960s were driven by two ideas. The more well known one was that the correctness proof of a program should be developed in parallel with the program and should, in fact, guide step by step refinement of the program. The less well known idea was that the denotational semantics of a programming language should guide the development of the language, in the same way as a correctness proof guides the development of the program. Moreover, it was well recognized by the specialists in the field that the extent of success of the first idea depends on our ability to develop new languages with well understood semantics using the second idea.

Unfortunately, the bulk of the development did not go this way. Instead, researchers in denotational semantics centered their efforts on describing the languages developed without denotational models in view. It was always recognized, that the efforts to prove the correctness of a program developed initially without a correctness proof are not too fruitful; similar recognition with respect to the development of programming languages unfortunately was less widespread. This has adversely affected the success of the both ideas.

We are trying to return to the original concept here - to set semantics requirements first, to develop language features to match these requirements later.

6.2 How functional are functional programming languages?

The term "functional languages" has two standard meanings - old and current. There is also the third - the ideal meaning, and the current and the ideal meanings are often confused.

Old meaning is that the texts of programs ("functions") are first-class objects - one can handle program texts as ordinary data and apply interpreter (e.g. `eval`) to them when necessary. This is the meaning in LISP [2].

The ideal meaning is that mathematical functions associated with programs and mapping inputs to outputs are first-class objects - one can treat them as ordinary data. Of course, in the real world we have to deal with representations of these functions in the memory of a computer, but the same is true about all other data, e.g. real and even integer numbers. And the ability to abstract from the representation issues most of the time is what brings strength and makes languages useful.

The functional language community recognized the necessity to abstract from representations and the fact that the old meaning of functional programming is too representation-oriented. So the switch to the current meaning happened (e.g. in ML [24]; Scheme combines both meanings but leans toward the new one, especially since the `eval` function was excluded from the canonical Scheme recently [12]). Current meaning is often said to be ideal - at least this was one of the important goals of this switch. We here try to argue, that it in fact closer to the old meaning (despite the loss of certain flexibility, the old meaning allowed), than to the ideal one.

The main problem is that the meaning of functions is understood in terms of evaluation process of some kind - usually some discipline of beta-reductions or some kind of an environmental model. Environmental models are closer to real implementations, and this leads to the perception that beta-reductions are more mathematical (the author thinks this perception is unwarranted and incorrect). In both models we view computed functions as closures (we argue that this is true for substitution models either - just look at the implementation of lazy computations under applicative order [2]), i.e. as delayed evaluations, not as new functions.

On the other hand when a mathematician thinks about a function he means its graph, or, in fact, he has some simplified representation of the graph in mind. One might say that this difference is very scholastic - how does one put it into scientifically expressed statements? Here we disagree with [21] - we believe that thinking in terms of function graphs is preferable to intensional thinking.

The first alarming sign is when the denotational semantics and its formulas become too close to an implementation (I mean the "main" semantics, not the derivative ones specially produced to achieve the bridge with an implementation). And being too close to an implementation the denotational semantics becomes too complicated and fails to help us to visualize the graphs of involved functions.

It is difficult to pinpoint this difference better in the type-free world. But when types and different versions of typed lambda-calculus come into play, it becomes absolutely clear that the current meaning of the term "functional programming" has very little in common with mathematical functions, and has everything in common with intensional objects. One of the crucial differences is in understanding of functional types. It is central for the ideal meaning of functional programming that the functional types be interpreted as subspaces of the spaces of functions. This is incompatible with Curry-Howard type systems which are in the heart of the current approach to functional programming (see the next subsection).

Before we move to the discussion of the typed world, let us look at the implications of the current meaning of functional programming being closer to the old one, than to the ideal one. First of all, we failed to gain power associated with the use of right math, as we hoped to. Second, we have lost the flexibility of `eval`, and it is not clear how this sacrifice - limiting our ways to produce programs - pays off. Third, we understand functions as closures, and closures are a partial case of objects of classes - namely we are talking about classes with one member function `apply`. This means that the object-oriented programming is strictly more powerful than the current functional programming (the relationship between lambda calculus and Abadi-Cardelly object calculus confirms this [1]). Moreover, the notion of subtyping in Curry-Howard systems is extremely well suited to express the inheritance notion in object-oriented programming [11]. The logical conclusion is that if we stay with the current paradigm we might as well switch to civilized versions of object-oriented programming; as it is now, too many of functional programs simulate something which is naturally expressible in the object programs (iteration, state, message passing etc. [2]). The inverse simulation is rare, and the object-oriented languages are probably augmentable with functional features with ease (it should be possible to accomplish a lot here just on the level of syntactic sugar).

So the author believes that if we fail to explore and exploit potential mathematical advantages of functional programming the switch to object-oriented programming is inevitable.

6.3 Requirements for type systems

On pages 5 and 6 of [8] we explain why Curry-Howard type systems require the use of multiple representations for the same function $f : A \to B$ in the denotational semantics. Thus the semantics of data type $A \to B$ is no longer a mathematical space of functions, since such spaces are extensional - $(\forall x \in A. f(x) = g(x)) \Rightarrow f = g$. This is a mathematical reason for the author's dissatisfaction with Curry-Howard systems and for his conviction that truly functional programming cannot be based on Curry-Howard systems.

Let us now create a wish list of more practical and immediate nature for type systems (and we will see that unfortunately Curry-Howard systems do not satisfy this wish list even closely).

6.3.1 Effectiveness

The exponential lower bound for ML type checking [27] and undecidability of type checking for system F [41] are hardly acceptable. We aim for linear or almost linear complexity of type checking of reasonably annotated programs.

6.3.2 Modest type inference

When one has to write in C

```
int i = 4;
```

or

```
int f (int a)  return a + 2;
```

the need for "`int i`" and "`int f`" here is outrageous (but not the need for "`int a`" in our opinion). We want trivial type inference, which would eliminate redundancy, but the program should not make guesses of what the programmer was trying to write, and reasonable annotations should be required. If by some reason one would like to have non-trivial type inference, then the high complexity might be justified. Unfortunately even for the type checking of properly annotated programs the complexity is high in the Curry-Howard polymorphic systems, and this is not acceptable. The author thinks that only trivial type inference is valuable - the rest just represents a bad style of programming (like the default typing of identifiers in Fortran).

6.3.3 Separation of compile-time computations and type checking

The type checking became a mean to perform compile-time partial evaluation. The author does not think it is appropriate. Compile-time evaluation is a different notion which cannot be reduced to type-checking.

More to the point we might want to do *effective run-time* type checking in many circumstances. *Effective* here means, for example, that when you run a loop types get checked dynamically only during the first iteration, but not later, unless there are reasons to think that they have been changed dynamically. Moreover, we might want to run a program, which does not quite type-checks, only using its type-safe parts (see our discussion of the dataflow programming below for clarification). Also note that incremental implementation would allow to reuse run-time made type checks in the subsequent runs.

We would prefer to consider fundamental computational models with run-time type checking (after all, usually the run-time overhead involved is only of a constant factor), and to view compile-time type checking as optimizing compile-time partial evaluation. Moreover, we might want to treat errors here as warning messages and to produce an exception generating code which works if and only if the offending branch of computations is invoked (of course, this requires properly annotated programs).

6.3.4 Data types as data

It is mathematically attractive to have both mathematical functions and computer programs as first-class values. The same can be said about types [39, 14]. It is mathematically attractive to be able to compute a type and then to use it to declare a variable. E.g. using C-like syntax we can define an array of 120 elements as

```
typedef int dynamic_type [ factorial (5) ];
dynamic_type a;
```

6.3.5 Understanding type polymorphism via types as parameters

This is connected with the idea of types as data. The author would like to treat polymorphic types and functions as having types as extra parameters, not as something that works blindly for any type substituted instead of a type variable as in ML [24]. On the other hand we would like to have only one piece of code for a polymorphic function, unless some very special optimization is desirable, not a separate piece of code for each possible use as in C++ templates [11]. The optimum is somewhere between Curry-Howard polymorphism and C++ templates, only it should be more powerful than the both of them. For example, we would like to be able to write

```
... function f (... type t ...)

    ...
    if (t is int)
    ...
    if (t is (* -> *))
    ...
```

Domain equations of denotational semantics should help us to provide meaning for type systems which conform to goals 6.3.3-6.3.5, as they helped to provide meaning for programming languages which allow functions to be run-time computable values.

The approaches to goals 6.3.3-6.3.5 known to the author [39, 14] involve the idea of typing an untyping world in the style of Curry-Howard systems and, thus, multiple representations of functions in function spaces, which is something we are trying to avoid.

6.4 Dataflow and incrementality, or how declarative are declarative programming languages?

The ideal of declarative (non-imperative) programming is that the programmer declares relations between data, and the interpretor finds a way to satisfy those relations. Dataflow architecture of the compiler design fits this ideal perfectly [6, 22]. Unfortunately, in the current state of the semantics of programming languages it is inevitable that the programmers will think about their programs operationally pretty frequently if not all the time. So let them think in terms of a dataflow implementation.

Dataflow models have in their heart the translation of a program into a graph of computational rules. The key idea is that when the sources for a rule are computed, the rule is allowed to be evaluated, if no inconsistencies are detected. Here is an example of an inconsistency:

```
i := 2
i := 3
```

This scheme is usually handled via the queue of rules awaiting evaluation. Advantages are numerous - a computer determines the order of evaluation, essentially yielding all useful values which can be computed and relieving a programmer from this burden; computations are easy to perform in parallel and the model matches the network architecture pretty well; it is convenient for incremental compilation and computations. If the program does not type checks we still can use the parts of the graph which do type check to perform some computations (it is also convenient to perform type inference in a data flow manner). Also it is convenient to think about such program together with computations performed as about a state of a data base.

Dataflow approach comes handy when we want to incorporate elements of programming via constraints for certain data types into our languages.

Languages should also be oriented to the dataflow way of thinking. For example, we support the requirement that the order of declarations should not matter, when the declarations are on one level (grouping is still allowed).

We would like to emphasize that we are talking about promoting and supporting dataflow orientation of programming languages, and dataflow architecture of compilers or interpretors. The author does not promote or support any special hardware architecture.

Right now programmers on Prolog, Scheme, or ML, or even versions of the λ-calculus have to refer too heavily to artificial orders and models of evaluation. This makes the claims that these languages are declarative somewhat artificial.

The author thinks that the dataflow architecture of compiler design is underused. We cannot discuss the historical reasons for that in details, but we'd like to name the two of these reasons which are, in our opinion, crucial. First, too many efforts were directed on dataflow implementation of classical imperative languages instead of more suitable declarative languages. Second, too much emphasis was on the specialized hardware for dataflow implementations, and the development of cheap central processors considerably undermined these efforts. Incredibly rapid rate of progress of the general purpose hardware means that a successful style of programming must be able to make successful use of existing computer architectures - in this case computers with one or several central processors linked into networks.

When one asks implementors of functional programming languages today why the dataflow architecture of the compiler was not chosen, the typical answer is that it have not been considered. We think that the potential here is big. Please observe that the dataflow approach does not fit well together with various kinds of substitution and rewriting models, but it does make environment-based models more mathematical and theoretically acceptable.

6.5 Dynamic data flow graphs and functions as values

The data flow approach means the representation of functions as data flow graphs. Thus, when we have functions as computed values we end up having dynamically computed data flow graphs. More precisely, the "main" data flow graph is changed dynamically by adding the subgraphs corresponding to the new functions. There is a slim possibility to avoid this when a strict discipline of generating the functional values is applied (if we can effectively uncurry all applications), but certainly not if we allow sufficiently unrestricted use of `eval`.

The author's experience is that dynamic data flow graphs are highly desirable in certain much more "benign" situations than functions as values. Essentially, as soon as the data flow analysis becomes sufficiently non-trivial, the requirement that it must be fully done at compile time becomes too restrictive.

There are obvious difficulties associated with the dynamic handling of data flow graphs. Decisions that a certain value is ready to be computed, or is properly computed and can be relied upon, are based on analysis of properties of the data flow graph being processed. Once such decision is made we rely on the corresponding graph property to remain valid in the future. If we allow unrestricted changes of the data graph, no decisions which are guaranteed to remain valid in the future are possible, thus no computations are possible unless we introduce a sophisticated backtracking mechanism (our desire to avoid this is very strong). The classical data flow models allow straightforward computations without backtracking, but do not allow dynamic changes in the data flow graph.

Our aim is to find an appropriate compromise, where the data flow analysis will take into account the possible future changes in the data flow graph, will accordingly block the computations which might depend on those changes but will not block anything else. Some very limited backtracking might be included. The language design must involve sufficient restrictions which would enable an interpretor to identify for a reasonable program enough of the computations ready to be performed at any given time, until the evaluation of the program is completed. Some restrictions on the use of `eval` still will be necessary. Namely, `eval` should be enclosed in the interface with says what in the outside environment it might change (the read-only use of the outside environment does not have to be controlled).

So far our advances in this direction are totally "ad hoc". If anyone who reads this is aware of a systematic approach to this problem, the author would greatly appreciate any leads. A limited version of dynamic data flow graphs is described in [6]. It has fixed targets and localized subgraphs (i.e. when a new subgraph is created, all vertices are new except for the target and source vertices), and this version is quite manageable. Unfortunately, its expressive power is quite limited.

7 Overview of Preliminary Results

7.1 Information systems

The author has a number of results on information systems (see Appendix 1).

We were able to describe the notion of a subdomain for Scott information system [8]. A subdomain of a Scott domain $|A|$ is simply a subset S of $|A|$, such that there is an information system B, such that $S = |B|$ (it is essential that this is an equality and not an isomorphism). We have shown that subdomains are obtainable by weakening the consistency and strengthening the entailment, and that they are precisely the sets of fixed points of closure operations on $|A_\top|$, excluding $\lambda x.\top$. We have studied the domain of subdomains and its properties

[3]. We have also explained why the sets of fixed points of finitary retractions and projections do not in general form subdomains.

We were able to give a new criterion of finitarity for retractions of Scott information systems $urw \Rightarrow \exists v.urv, vrv, vrw$, and to study the domains isomorphic to their sets of fixed points [7]. We also studied the domains of finitary projections and finitary retractions.

We were able to explain the intuition behind R.Hoofman's construction [26] of continuous information systems [7, 9]. We found that our results on subdomains and finitary retractions for classical information systems have very natural explanations from this point of view. We have also made steps to generalize Hoofman's approach to the case of complete lattices.

Right now we are trying to describe relationships between this approach and Sazonov and Sviridenko's description of continuous and complete lattices via infinitary logic [33].

7.2 Analysis

We made very modest advances in the development of topological algebra for domains. Our approach was as follows. The difficulties in the development of algebra are well illustrated by interval numbers. While both addition and multiplication on interval numbers form commutative monoids, operations of taking the inverse work less well, and only a weak distributivity property holds. We concentrate here on an addition operation. Unary minus is a quasi-inverse operation: $x + (-x) \sqsubseteq 0$. E.g. $[1, 2] + (-[1, 2]) = [1, 2] + [-2, -1] = [-1, 1] \sqsubseteq [0, 0]$. Not only does this prevent us from literally transferring results from classical mathematics, it is not trivial to define interval unary minus just by looking at the interval plus. Indeed, $x + y \sqsubseteq 0$ iff intervals y and $-x$ have non-empty intersection. The problem is to formulate the rule which would select $-x$ from the set of such y's. It cannot be done just by looking at x, we have to look at all interval numbers at once. The interval unary minus then is defined as the maximal continuous function $f : R_{interval} \to R_{interval}$, such that for any interval x the inequality $x + f(x) \sqsubseteq [0, 0]$ holds.

In retrospect this line of research seems to be considerably misdirected. This approach came from the author's desire to import as much of classical topological algebra into domains as possible. It turns out, however, that metric and measure structures need just ordered commutative monoids (essentially, some version of non-negative numbers) [29, 15], while the ability to compute inverses and even to multiply seems not to be too important. So while quasi-inverse operations on domains attract our curiosity, it seems unlikely that their studies could contribute significantly to the development of analysis on domains.

[3] The main theorem of [8] is the Theorem 3.1. The paper [8] contains a complicated inductive proof of this theorem - the technique used most frequently in the theoretical computer science. Recently the author and Michael Alekhnovich, a freshmen at Moscow State University and Moscow Independent University, developed a new elegant proof based on the intuition that the domain of subdomains is the domain of ways to strengthen our axioms and rules of inference. This should also facilitate further studies of the domain of subdomains and domain equations involving this construction.

7.3 Programming languages

Appendix 2 describes our preliminary steps in the design of small languages.

The author also has the following relevant experience. It does not exactly fit the "Preliminary Results" title, since the developments here were application-oriented and the author does not intend to include them in the Thesis. Yet, they allowed the author to strengthen his background in the design and implementation of programming languages, and he hopes to use this background to develop prototypes of general-purpose systems.

The author participated in the design and implementation of an incremental data flow programming language with higher-order functions and with linear time type inference. All these features have made it to our wish list of desirable properties of programming languages. We are now trying to handle dynamic data flow graphs systematically and we are considering departing from the idea of completing type checking before any computations can start (the current framework potentially allows dynamic computations of types).

8 Conclusion: List of Problems

Here we enumerate the problems our Thesis will try to address. The problems touched above are just named. The problems which are not touched above are detailed.

This is essentially a wish list. The goal for a complete Thesis is to include answers to 25-40% of the problems mentioned either in the Preliminary Results section or in this list. To compare, our preliminary results in information systems (the only results that can be readily used) will yield about 15% of these problems after we properly record all technical details.

8.1 Mathematics of information systems

Classification of retractions and continuous transformations. Studies of classes of quasi-minimal and strongly finitary retractions (Appendix 2). Subdomains for continuous and complete lattices. Further studies of the domain of subdomains construction. Domain equations using the domain of subdomains construction. Relationships between non-reflexive and infinitary logics for continuous and complete lattices.

8.2 Analysis on domains

8.2.1 Generalized distances

More intuitive presentations of generalized distances. Computation of generalized distances for particular pairs of programs.

8.2.2 Connections with measure theory

Explore connections between domains and measure theory discovered by A.Edalat and his applications of these connection to machine learning, and develop them into the measure theory for domains.

8.2.3 Connection with bases of orthogonal functions

Explore analogies between software development and decompositions of functions via bases of almost orthogonal functions.

One of the important methods of solving extremal problems in functional analysis is the decompositions of functions via the basis of orthogonal functions. One can think about possible generalization of this approach to the bases consisting of almost orthogonal functions. The process of successful software development has surprisingly much in common with such generalization.

8.3 Fundamentals of the theory of computations

8.3.1 Information systems as programming languages

The hope here is to find very simple programming languages with the expressive power comparable to the one of λ-calculus and with clear concept of computational cost (see also 8.3.3 and Appendix 2). Eager and lazy interpretations are possible here.

8.3.2 Non-determinism and parallelism

Can we avoid the use of powerdomains in the semantics of non-deterministic and parallel computations? Can we have simple denotational models of non-deterministic and parallel computations? The reasoning underlying these goals is that mathematics currently used in denotational description of non-deterministic and parallel computations seems to be very complicated compared to the sequential case. In particular, it seems to be too complicated for real applications. We think that power domain models try to describe the computations too precisely [4]. We might want to trade the completeness of description of computations in our denotational models for simplicity - for example, considering exact upper and/or lower bounds of alternatives. After all, the key idea underlying denotational semantics is to find models of computations which abstract from the less important details and not to find another way to describe what happens with every bit.

8.3.3 Computational complexity

Is there a simple way to express computational complexity in the denotational framework? We would like to note that this question is similarly open for the lambda calculus [32]. There are at least two ways to deal with it in denotational

[4] We do not object against the use of probabilistic powerdomains

semantics - to include resource consumption counters in the semantics, or to explore the expressability of the complexity of inference in the informations systems. The first way is closer to the real life applications. The second seems more promising theoretically. The fulfillment of the goal 8.3.1 might provide us with the ability to combine these two ways.

8.4 Design and implementation of programming languages

We need to design and implement prototypes of programming languages which would satisfy as much of the following properties as possible. Good denotational semantics and, in particular, functional types as spaces of functions. Data types as data, and type polymorphism understood via data types as parameters. Simple and efficient type checking and inference. The ability to move type checking into run time without loss of efficiency. Data flow orientation. Incrementality of compilation and computations. Proper treatment of dynamic data flow graphs.

8.5 Links with other programming paradigms

8.5.1 Links with logical programming

Information systems provide the potential to establish links with logical programming. The current denotational models of logical programming usually work in terms of sets of ground terms forgetting the polymorphic structure of the results of logical programs evaluation - there is a potential for improvement here.

8.5.2 Links with object-oriented programming

Object-oriented programming is a paradigm which became popular significantly after the foundations of denotational semantics were developed. This paradigm has proved itself to be very fruitful in the practical software engineering and seems to fit well together with Curry-Howard concept of data types, at least in spirit.

In this text we have argued at length that we should focus our attention on the alternatives to the Curry-Howard paradigm. Yet we cannot ignore the fact that the properties of inheritance in object-oriented programming are similar to the properties of subtyping in Curry-Howard systems. How to combine the strength of the object-oriented approach and, in particular, inheritance with our desire to treat functional types as extensional spaces of functions is an important open question.

8.6 Analogies to explore

Here we list some analogies which are not very well understood. They may be just a pure coincidence, but there are sufficient possibilities of interesting connections to ignore them altogether. We just present them here with the intention to keep them in mind.

8.6.1 Analogies with Kripke models

Analogy between domains as sets of theories and Kripke models, especially Kripke models of intuitionistic logic.

8.6.2 Analogies with non-standard analysis

Analogy between the main axiom of the domain theory - upper bounds of directed sets - and the directness principle of non-standard analysis. That is, in some sense domains possess some of the good properties of non-standard models.

8.6.3 Analogies between closures and monads

Some categorical analogies. In particular, analogy between the exclusive role of closure operations and properties of certain monads. We recently found a non-categorical explanation of exclusive role of closure operations (see Preliminary Results section), which was a mystery for us for a number of years. Yet, this analogy still might be more than just a coincidence.

9 Acknowledgments

The author would like to thank Alexander Saevsky and Alexander Shen', for without many helpful discussions with them his initial progress in the field would be impossible. The author is also greatful to Harry Mairson for arranging the financial support and for numerious discussions and advices, which considerably influenced author's research. And finally, the author is greatful to many members of the Theoretical Computer Science community of Boston, and especially to the participants of seminars lead by Mitch Wand and Albert Meyer, for the fruitful research environment they have created.

References

[1] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, pages 296–320. Springer-Verlag LNCS 789, 1994.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* Springer, 1991.

[3] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[4] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.

[5] Errett Bishop and D. Bridges. *Constructive Analysis.* Springer-Verlag, 1985.

[6] Anton Bohm. *Dataflow Computations*, volume 6 of *CWI tract.* Centrum voor Wiskunde en Informatica, Amsterdam, 1984.

[7] Michael Bukatin. *Information Systems and Retractions: an Elementary Treatment.* Available via e-mail requests to bukatin@cs.brandeis.edu, December 1994.

[8] Michael Bukatin. *Subdomains for Algebraic Information Systems.* Available via e-mail requests to bukatin@cs.brandeis.edu, November 1994.

[9] Michael Bukatin. Information systems and complete lattices. In *11th Summer Conference on General Topology and Applications. August 10-13, 1995, University of Southern Maine, Gorham, Maine. ABSTRACTS*, 1995.

[10] Ho Chun-Kuen. Beyond recursive real functions. *Information and Computation*, 124:113–126, 1995.

[11] Marshall Cline and Greg A. Lomow. *C++ FAQs: Frequently Asked Questions.* Addison-Wesley, 1994.

[12] William Clinger and Jonathan A. Rees. The revised^4 report on the algorithmic language scheme. *ACM LISP Pointers*, 4(3), 1991.

[13] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[14] James Donahue and Alan Demers. Data type are values. Technical report, Xerox Corporation, 1984.

[15] Abbas Edalat. Domain theory and integration. *Theoretical Computer Science*, 151:163–193, 1995.

[16] Abbas Edalat. Domain theory in learning processes. In S. Brooks et al., editor, *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1995.

[17] Abbas Edalat. Dynamical systems, measures and fractals via domain theory. *Information and Computation*, 120(1):32–48, July 1995.

[18] Abbas Edalat and Michael B. Smyth. Categories of information systems. In D. Pitt et al., editor, *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, 1991.

[19] Abbas Edalat and Michael B. Smyth. I-categories as a framework for solving domain equations. *Theoretical Computer Science*, 115:77–106, 1993.

[20] Bob Flagg and Ralph Kopperman. Fixed points and reflexive domain equations in categories of continuity spaces. In S. Brooks et al., editor, *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[21] Jean-Yves Girard. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989.

[22] Hugh Glazer, Chris Hankin, and David Till. *Principles of Functional Programming*. Prentice-Hall International, 1984.

[23] R. L. Goodstein. *Recursive Analysis*. North-Holland, Amsterdam, 1961.

[24] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[25] David Gries. *The Science of Programming*. Springer, New York, 1981.

[26] R. Hoofman. Continuous information systems. *Information and Computation*, 105(1):42–71, July 1993.

[27] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. *Unification and ML Type Reconstruction*, pages 444–478. MIT Press, 1991.

[28] Ralph Kopperman. All topologies come from generalized metrics. *American Mathematical Monthly*, 95:89–97, 1988.

[29] Ralph Kopperman and Bob Flagg. The asymmetric topology of computer science. In S. Brooks et al., editor, *Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 544–553. Springer-Verlag, 1993.

[30] K. G. Larsen and G. Winskel. Using information systems to solve recursive domain equations effectively. In D. B. MacQueen G. Kahn and G. Plotkin, editors, *Semantics of Data Types*, pages 109–130, Berlin, 1984. Springer-Verlag. Lecture Notes in Computer Science Vol. 173.

[31] Kim Guldstrand Larsen and Glynn Winskel. Using information systems to solve recursive domain equations. *Information and Computation*, 91(2):232–258, April 1991.

[32] Julia L. Lawall and Harry G. Mairson. *Optimality and Inefficiency: What Isn't a Cost Model of the Lambda Calculus?* to appear, 1996.

[33] Vladimir Sazonov and Dmitri Sviridenko. Abstract deducibility and domain theory. Technical report, Rutgers University, 1996. Expected to be published as DIMACS Technical Report in 1996.

[34] Martin Schechter. *Principles of Functional Analysis*. Academic Press, 1971.

[35] Dana Scott. Lattice theory, data types and semantics. In Randall Rustin, editor, *Formal Semantics of Algorithmic Languages*, pages 65–106. Prentice Hall, 1972.

[36] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings, 21st Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971. Also, Programming Research Group Technical Monograph PRG–6, Oxford University.

[37] Dana S. Scott. Continuous lattices. In F. William Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, volume 274 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, Berlin, Heidelberg, and New York, 1972.

[38] Dana S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[39] A. Shamir and W. W. Wadge. Data types as objects. In *Automata, Languages and Programming*, volume 52 of *Lecture Notes in Computer Science*, pages 465–479. Springer-Verlag, 1977.

[40] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, Cambridge, Massachusetts, 1977.

[41] Joe Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, Paris, France, July 4–6, 1994. The IEEE Computer Society Press.

[42] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.

[43] K. Yosida. *Functional Analysis.* Springer-Verlag, 1980.

10 Appendix 1: Information Systems for Domains of Fixed Points

10.1 Definitions and notation

In this brief review, we mainly follow [30]. Notions and results of this section should be mostly credited to Dana Scott [38]. These sources contain extensive discussions of the subject.

Notation 1 *If D is a set, $P(D)$ is a set of all its subsets, $P_{Fin}(D)$ is a set of all its finite subsets. $u \subseteq_{Fin} x$ means that a set u is a finite subset of a set x.*

Definition 1 *$A = (D_A, Con_A, \vdash_A)$, where D_A is a set of distinctive* tokens *(*assertions*), $Con_A \subseteq P_{Fin}(D_A)$ (the* consistent *finite conjunctions of assertions), $\vdash_A \subseteq Con_A \times D_A$ (the* entailment *relation), is called an* information system *(*IS*) if*

(i) $\emptyset \in Con_A$ (non-degeneracy; empty domains are barred);

(ii) $\forall d \in D_A.\{d\} \in Con_A$ (no "junk");

(iii) $\forall u \subseteq v \in Con_A.u \in Con_A$;

(iv) $\forall d \in u \in Con_A.u \vdash_A d$ (reflexivity of entailment);

(v) $\forall u \in Con_A, d \in D_A.u \vdash_A d \Rightarrow u \cup \{d\} \in Con_A$ (entailment preserves consistency);

(vi) $\forall u, v \in Con_A, d \in D_A.(\forall d' \in v.u \vdash_A d'), v \vdash_A d \Rightarrow u \vdash_A d$ (transitivity of entailment).

Notation 2 *We avoid the usual overloading of $\vdash_A$. Define $\vdash_A^* \subseteq Con_A \times Con_A$ by $u \vdash_A^* v$ iff $\forall d \in v.u \vdash_A d$.*

Definition 2 *Consider an information system A.*

$|A| = \{x \subseteq D_A \mid$

(i) $\forall u \subseteq_{Fin} x.u \in Con_A$ (consistency);

(ii) $\forall u \subseteq_{Fin} x.\forall d \in D_A.u \vdash_A d \Rightarrow d \in x$ (deductive closure) $\}$

is called a domain *described by the information system A.*

For $x, y \in |A|$ define $x \sqsubseteq_A y$ iff $x \subseteq y$.

Definition 3 *For any $x \subseteq D_A$ possessing the property of consistency ($\forall u \subseteq_{Fin} x.u \in Con_A$), $\overline{x} = \overline{x_A} = \{d \in D_A \mid \exists u \subseteq_{Fin} x.u \vdash_A d\}$ is called a* deductive closure *of x (in A).*

One can see that $\overline{x} \in |A|$.

If $u \in Con_A$ then $\overline{u}$ is called a *finite element* of $|A|$.

Notation 3 $\perp_A = \overline{\emptyset_A}$. *It is easy to see, that $\perp_A$ is the least element of $|A|$.*

Exercise 1 *Show that for $x_1, \ldots, x_n, \ldots \in |A|$, such that $x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$, there is the exact upper bound $\sqcup\{x_i \mid i \in N\} = \cup\{x_i \mid i \in N\} \in |A|$.*

We have defined *information systems* and *domains*, and now proceed to the definition of *continuous functions*. We start with the definition of *input-output entailment relations* which are also called *approximable mappings* (mathematicians should be warned that these *mappings* are *not* functions).

Definition 4 *Consider information systems A and B. $f \subseteq Con_A \times Con_B$ is called an* approximable mapping *(or* input-output entailment relation*) between A and B (we write $f : A \to B$) if*

(i) $\emptyset f \emptyset$;

(ii) $ufv, ufw \Rightarrow (v \cup w) \in Con_B, uf(v \cup w)$;

(iii) $u' \vdash_A^* u, ufv, v \vdash_B^* v' \Rightarrow u'fv'$ *(transitivity of the input-output entailment relation with the entailment relations in A and B).*

Definition 5 *Let f be an approximable mapping. Then $|f| : |A| \to |B|$ defined by $|f|(x) = \{d \in D_B \mid \exists u \subseteq_{Fin} x.uf\{d\}\}$ is called a* continuous function *from $|A|$ to $|B|$.*

Exercise 2 *Show that $f(x) \in |B|$ (correctness). Show that $|f|(x) = \cup\{v \in Con_B \mid \exists u \subseteq_{Fin} x.ufv\}$ yields an equivalent definition of a continuous function. Show that f is monotonic and preserves the exact upper bounds of sequences from the previous exercise. Show that $f \subseteq g \Leftrightarrow \forall x \in |A|.|f|(x) \sqsubseteq_B |g|(x)$.*

Lemma 1 *1) $| \vdash_A^* | = id$. 2) $|g| \circ |f| = |g \circ f|$, where $\forall u \in Con_A, w \in Con_C.u(g \circ f)w \Leftrightarrow \exists v \in Con_B.ufv, vgw$. In particular, a continuous function $|r| : |A| \to |A|$ is a retraction ($|r| = |r| \circ |r|$) iff $\forall u, w \in Con_A.urw \Leftrightarrow \exists v \in Con_A.urv, vrw$.*

Definition 6 *Consider information systems A and B. Then the* information system of approximable mappings, *$A \to B$, is defined as follows:*

$$D_{A \to B} = Con_A \times Con_B;$$

$$\{(u_1, v_1), \ldots, (u_n, v_n)\} \in Con_{A \to B} \quad \textit{iff}$$
$$\forall I \subseteq \{1, \ldots, n\}.\cup\{u_i \mid i \in I\} \in Con_A \Rightarrow \cup\{v_i \mid i \in I\} \in Con_B;$$

$$\{(u_1, v_1), \ldots, (u_n, v_n)\} \vdash_{A \to B} (u, v) \textit{ iff } \cup\{v_i \mid u \vdash_A^* u_i\} \vdash_B^* v.$$

Exercise 3 *Prove that $A \to B$ satisfies the definition of an information system. Prove that $|A \to B| = \{f \subseteq Con_A \times Con_B \mid f$ is an approximable mapping between A and $B\}$. Prove that different approximable mappings define different continuous functions, and therefore, approximable mappings and continuous functions are in 1-1 correspondence.*

Notation 4 *$|A \to B|$ is the domain of all approximable mappings between A and B, but, due to the 1-1 correspondence between approximable mappings and continuous functions, it is said that $|A \to B|$ is the* domain of continuous functions *between $|A|$ and $|B|$, and it is called the* functional space.

Definition 7 *A pair of continuous functions $|i| : |B| \to |A|$, $|j| : |A| \to |B|$ is said to be a* retraction *of $|A|$* onto *$|B|$ if $|j| \circ |i| = id_B$.*

Lemma 2 *If $|i|, |j|$ form a retraction of $|A|$ onto $|B|$, then $|r| = |i| \circ |j|$ is a retraction of $|A|$ ($|r| = |r| \circ |r|$).*

Definition 8 *If $|i|, |j|$ form a retraction of $|A|$ onto $|B|$, and the retraction $|r|$ from Lemma 1.15 is a projection ($|r| \sqsubseteq id_A$), then we say that $|i|, |j|$ form a* projection *of $|A|$* onto *$|B|$.*

Definition 9 *Consider a domain $|A|$ and a retraction (resp. projection) $|r| : |A| \to |A|$. If there exist a domain $|B|$ and a pair of continuous functions $|i|, |j|$ that forms a retraction of $|A|$ onto $|B|$, such that $|r| = |i| \circ |j|$, then $|r|$ is called a* finitary retraction *(resp.* finitary projection*).*

10.2 From continuous transformations to generalized entailments

Here we explain intuition behind Hoofman's generalization of information systems onto continuous domains [26] and generalize it to the bounded complete cpo's.

As we already know, continuous functions can be described by approximable mappings which are input-output entailment relations. If we consider a continuous transformation, $f : A \rightarrow A$, then the corresponding approximable mapping $f \subseteq Con_A \times Con_A$ resembles the canonical entailment relation $\vdash_A^* \subseteq Con_A \times Con_A$.

We define a generalized information system by changing axioms (iv)-(vi) of Definition 1. ((iv)-(v) become like Hoofman's – see below – and (vi) goes away). To justify this think in terms of $f = \vdash^*_{future}$, take into account that $\ni \subseteq \vdash$, and apply the definition of f as an approximable mapping.

By imposing certain restrictions on continuous transformations we can obtain additional axioms. For example, $id \sqsubseteq f$ corresponds to reflexivity (axiom (iv) of Definition 1). A weaker restriction, $f \sqsubseteq f \circ f$, corresponds to the *inverse transitivity* rule of Hoofman, $A \vdash C \Rightarrow \exists B.\ A \vdash B, B \vdash C$.

Alternatively, $f \circ f \sqsubseteq f$ corresponds to transitivity (axiom (vi) of Definition 1). A stronger restriction, $f \sqsubseteq id$, would yield a certain *inverse reflexivity* condition.

If one decomposes $f = f \circ f$ into $f \circ f \sqsubseteq f$ and $f \sqsubseteq f \circ f$, then one obtains transitivity and inverse transitivity, which taken together constitute axiom (vi) of Hoofman's system, which is defined as follows [26].

Definition 10 *$A = (D_A, Con_A, \vdash_A)$, where D_A is a set of distinctive* tokens *(*assertions*), $Con_A \subseteq P_{Fin}(D_A)$ (the* consistent *finite conjunctions of assertions), $\vdash_A \subseteq Con_A \times D_A$ (the* entailment *relation), is called an* continuous information system *(*IS*) if*

(i) $\emptyset \in Con_A$ (non-degeneracy; empty domains are barred);

(ii) $\forall d \in D_A.\{d\} \in Con_A$ (no "junk");

(iii) $\forall u \subseteq v \in Con_A.u \in Con_A$;

(iv) $\forall u, v \in Con_A.u \subseteq v, u \vdash_A d \Rightarrow v \vdash_A d$ (monotonicity of entailment);

(v) $\forall u \in Con_A, d_1, \ldots, d_n \in D_A.u \vdash_A d_1, \ldots, u \vdash_A d_n \Rightarrow \{d_1, \ldots, d_n\} \in Con_A$ (entailment yields consistent sets);

(vi) $\forall u, d \in D_A.(\exists v \in Con_A.(\forall d' \in v.u \vdash_A d'), v \vdash_A d) \Leftrightarrow u \vdash_A d$ (transitivity and inverse transitivity of entailment).

Definition 11 *Removing condition (vi) above we define a* complete information system*.*

We construct a complete information system from any approximable mapping f by defining $u \vdash_{new} d \Leftrightarrow uf\{d\}$. If f is a retraction, a continuous information system results.

10.3 Generalized domains are sets of fixed points of continuous transformations

We want to define a domain $|A|$ corresponding to a continuous or complete information system A in such manner, that we would obtain $|A| = Fix(f)$, i.e. $\forall x \in |A|. x = f(x)$. Decomposing this into $f(x) \sqsubseteq x$ and $x \sqsubseteq f(x)$ we obtain, respectively, ordinary deductive closure and *inverse deductive closure* as in [26].

Definition 12 *Consider a complete information system A.*

$|A| = \{x \subseteq D_A \mid$

(i) $\forall u \subseteq_{Fin} x. u \in Con_A$ *(consistency);*

(ii) $\forall u \subseteq_{Fin} x. \forall d \in D_A. u \vdash_A d \Rightarrow d \in x$ *(deductive closure)*

(iii) $\forall d \in D_A. d \in x \Rightarrow \exists u \subseteq_{Fin} x. u \vdash_A d$ *(inverse deductive closure)* $\}$

is called a complete domain *described by the information system A.*

If we start with a continuous information system this construction yields a continuous domain as in [26].

10.4 Application: exclusive role of closure operations

In [8] we found that in the algebraic case subdomains are obtainable via closure operations, and, in general, even finitary projections or retractions do not yield subdomains.

The reason for closure operations playing such an exclusive role from the point of view described above, is that closure operations yield reflexive, transitive entailments. Thus, the resulting complete information systems and complete domains form informations systems and domains from the classical point of view.

10.5 Application: finitary retractions

In [7] we studied finitary retractions (i.e. retractions with algebraic sets of fixed points). We found that a retraction $|r|$ is finitary if and only if $\forall u, w \in Con.\ urw \Rightarrow \exists v \in Con.\ urv, vrv, vrw$ holds for the corresponding approximable mapping r.

The intuition here is that we have enough of reflexive statements v, so that all entailments go via such statements. Thus we can throw out all other statements and use v's to construct a classical information system, which would yield a domain isomorphic to the original one. The alternative is to get rid of reflexivity in our logic.

10.6 Conclusion

Now we know exactly what is the meaning and the value of algebraicity - it corresponds to the reflexivity of the logic in question. Of course, reflexivity of logic is a very desirable condition, hence algebraicity is a very desirable property.

The next thing to do is to study morphisms (complete approximable mappings). For continuous information systems Hoofman obtained the equivalence between categories of continuous domains and continuous functions, and continuous information systems and continuous approximable mappings. It is not quite clear, whether such equivalence is possible in the more general case of complete domains. The author is working on this now.

Also there is a way to preserve reflexivity and still describe bounded complete cpo's. This way requires non-finitary inference rules [33]. The author is now trying to understand why the loss of reflexivity of inference yields the same results as the loss of finitarity of inference rules.

11 Appendix 2: Exercises in Programming Language Design

These are very first steps in this project, and these are just unpolished lab notes.

11.1 A basic version of a dataflow language with types as data

11.1.1 Syntax

```
<Program> ::=  <Definition>
```

Program is a collection of definitions. It may be empty, The order of definitions is not important.

```
<Definition> ::= <Variable> = <Expression> ;

<Expression> ::= <Constant> |
                 <Variable> |
                 <Function Application> |
                 <Infix Application > |
                 <Where Expression> |
                 <If Expression> |
                 ( <Expression> )

<Function Application> ::= <Function Name> ( <Expression List> )

<Expression List> ::= [ <Expression> { , <Expression> } ]

<Infix Application> ::= <Expression1> <Infix> <Expression2>
```

Just a syntactic sugar for function applications. An equavalent is $<Infix> (<Expression1>, <Expression2>). [In the subsequent versions: User can define his own infixes - a preambule to the program must contain the list of all infixes (including the built-in ones - the user is just allowed to edit the default preambule) with their properties (respective priorities and associativity)]

```
<Where Expression> ::= <Expression> where { <Program> }
```

Variables defined in the <Program> are used in <Expression>, and may screen the names from the containing environment.

```
<If Expression> ::= if <Boolean Expression> then <Expression 1>
                                             else <Expression 2>
```

```
<Constant> ::= <Special Constant> |
               <Constant Name> |
               <Function Definition>
```

Integer numbers are a good example of special constants.

```
<Function Definition> ::= function ( <Parameter List> )
                                    : <Body Expression>
```

Parameter names are used in <Body Expression> during the function calls and may screen the names from the containing environment

```
<Parameter List> ::= [ <Parameter Definition>
                     { , <Parameter Definition> } ]
```

```
<Parameter Definition> ::= <Type Expression> <Parameter Name>
```

Type expression is an expression evaluating to the type. There are type constants and built-in functions returning type, and the user can define his own functions returning type. The word "type" is reserved to denote the type of all types.

11.1.2 Model of computations

We start with a predefined external environment. One can think of a <Constant Name> as a variable mysteriously defined outside. We have a program and also the list of variables ("list of requests") we would like to compute. Nevertheless, we use data-driven, rather than demand-driven, model of computation. This means that we compute all we can, printing information about changing in the status of variables from our list of requests.

The <Definition> can be executed (added to the execution queue) when its <Expression> can be evaluated. <Special Constant¿ and <Constant Name> can always be evaluated. <Variable> can be evaluated if its definition was successfully executed. <Where Expression> is translated into a collection of definition ("flattening" it), just the local names should be properly prefixed to avoid name conflicts. <Function Definition> is ready to be evaluated when all <Type Expressions> in its <Parameter List> are evaluated.

The <Function Application> (ditto for <Infix Application>). The <Function Name> should either be <Constant Name> or should be a <Variable> whose definition was successfully executed (In the next version we probably will allow to make anonymous functions, i.e. to use <Function Expression> in place of <Function Name>, like we've done with types in this version. Then the requirement will be that <Function Expression> is successfully evaluated). Also all argument expressions should be successfully executed. Then the <Body Expression> is translated to the new expression which now must be evaluated when ready (at once when it is a built-in function) (at this stage the type checking happens, by calling the special substitutability check utility - if it fails, then function application fails).

The <If Expression> can be evaluated when its <Boolean Expression> can be evaluated, and if the evaluation succeed either <Then> or <Else> clause is substituted instead.

11.1.3 Substitutability check utility

Right now built-in type returning functions will just build trees from their arguments, and we are going to require that this trees are equal. This part will undergo big changes.

11.2 Modifications of the first version

11.2.1 Meaning of functions

In the version above function is just a set of ordered pairs. It conforms well to the old-fashioned point of view, that if $f : A \to B$ and $B \subseteq C$, then $f : A \to C$. But this makes it impossible to know range effectively, and we have to say that theoretically the range of any function is all possible values. We might wish to restrict this by introducing

```
<Function Definition> ::= function <Return Type Expression>
                                    ( <Parameter List> )
                                    : <Body Expression>
```

Then the dynamic type checking is made when a function returns a value.

11.2.2 Abstract types

The power of our type is, roughly speaking, equivalent to ML `datatype` polymorphic construction. E.g. all interesting data types must be predefined outside. We need to introduce type constants, then we will have, roughly speaking, an equivalent of `abstype` construction. We understand now how to do this for monomorphic types, but not for type-yielding functions (our equivalent of polymorphic types). We are working on this right now.

11.2.3 Truly dynamic data flow graphs

We can allow operators under `then` and `else` and within `where` environments and function bodies. Until now we did need dynamic data flow graphs, but only their limited version, like in [6]. Now it is not enough. To make things more complicated we might wish to add arrays with variable indices here.

Actually, there is a theoretical solution. We think of rules in the graph as input-output entailment machines (approximable mappings in Scottery), and nodes do perform internal entailment according to their type. We will try to make a precise model of this kind within the next several weeks.

From the practical point of view this is not sufficient. Pretty often we deal with rules which should not be activated until it is known that some information cannot be inferred (if it is inferred then the rule should not be activated at all). This is a case of anti-monotonic calculations, and they are "internally uncomputable", and might be pretty often actually undecidable. We do know actual applied examples of this kind.