# THEORETICAL ASPECTS OF COMPUTER SOFTWARE
Proceedings, International Conference TACS'91
Sendai, Japan, September 1991
Springer-Verlag

T. Ito and A. R. Meyer, eds.

August 14, 1991

This is a preliminary release of the contents and abstracts of the Proceedings of the International Conference on Theoretical Aspects of Computer Software. There may still be a few bugs in it. If you find one, please send it to `dmjones@theory.lcs.mit.edu`.

## References


[1] Krzysztof R. Apt and Dino Pedreschi. Proving termination of general prolog programs. In Ito and Meyer [18], pages 265–289.


> We study here termination of general logic programs with the Prolog selection rule. To this end we extend the approach of Apt and Pedreschi [AP90] and consider the class of *left terminating* general programs. These are general logic programs that terminate with the Prolog selection rule for all ground goals. We introduce the notion of an *acceptable program* and prove that all acceptable programs are left terminating. This provides us with a practical method of proving termination.
>
> The converse implication does not hold by we show that under the assumption of non-floundering from ground goals every left terminating program is acceptable. Finally, we prove that various ways of defining semantics must coincide for acceptable programs. The method is illustrated by giving simple proofs of termination of a "game" program and the transitive closure program for the desired class of goals.

[2] Dieter Armbruster. Polynomial recursion analysis in pascal like programs. In Ito and Meyer [18], pages 447–458.

Besides being of theoretical interest the knowledge about a procedure's calling behavior is valuable for an optimizing compiler. It is well known, however, that such properties like recursivity or reachability of procedures are unfortunately *undecidable* for programs in *ALGOL–like* languages and are still worse than *P–Space Complete* in the *ISO–Pascal* case.

We extend this language hierarchy (with respect to parameter restrictions) at the lower end to *Wirth's Pascal* and show that there the recursivity problem for procedures is decidable within *polynomial time.* In order to establish this (rather unexpected) result we

1. reduce recursivity of a procedure to a reachability problem – with both properties being defined on an infinite tree representation of the program;
2. show the equivalence between reachability in such an *infinite tree* on the one hand and reachability in the *finite graph* representation on the other hand;
3. solve then the reachability problem in this graph in $O(ns)$ of a program with $n$ procedures as vertices and $s$ call statements as edges.

[3] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. In Ito and Meyer [18], pages 152–175.

A simple efficiency preorder for CCS processes is introduced in which $p \precsim q$ means that $q$ is at least as fast as $p$, or more generaly, $p$ uses at least as much resources as $q$. It is shown to be preserved by all CCS contexts except summation and it is used to analyse a non-trivial example: two different implementations of a bounded buffer. Finally we give a sound and complete proof system for finite processes.

[4] Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In Ito and Meyer [18], pages 651–674.

A type assignment with *union* and *intersection* types is introduced. Relevant syntactical and semantical properties of this system are proved.

[5] David A. Basin and Douglas J. Howe. Some normalization properties of Martin-Löf's type theory, and applications. In Ito and Meyer [18], pages 475–494.

For certain kinds of applications of type theories, the faithfulness of formalization in the theory depends on intensional, or structural, properties of objects constructed in the theory. For type theories such as LF, such properties can be established via an analysis of normal forms and types. In type theories such as Nuprl or Martin-Löf's polymorphic type theory, which are much more expressive than LF, the underlying programming language is essentially untyped, and terms proved to be in types do not necessarily have normal forms. Nevertheless, it is possible to show that for Martin-Löf's type theory, and a large class of extensions of it, a sufficient kind of normalization property does in fact hold in certain well-behaved subtheories. Applications of our results include the use of the type theory as a logical framework in the manner of LF, and an extension of the *proofs-as-programs* paradigm to the synthesis of verified computer hardware. For the latter application we point out some advantages to be gained by working in a more expressive type theory.

[6] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system $F$ with subtyping. In Ito and Meyer [18], pages 750–770.

System $F$ is a well-known typed $\lambda$-calculus with polymorphic types, which provides a basis for polymorphic programming languages. We study an extension of $F$, called $F_{<:}$, that combines parametric polymorphism with subtyping.

The main focus of the paper is the equational theory of $F_{<:}$, which is related to PER models and the notion of parametricity. We study some categorical properties of the theory when restricted to closed terms, including interesting categorical isomorphism. We also investigate proof-theoretical properties, such as the conservativity of typing judgements with respect to $F$.

We demonstrate by a set of examples how a range of constructs may be encoded in $F_{<:}$. These include record operations and subtyping hierarchies that are related to features of object-oriented languages.

[7] Carl A. Gunter, Elsa L. Gunter and David B. MacQueen. An abstract interpretation for ML equality kinds. In Ito and Meyer [18], pages 112–130.

The definition of Standard ML provides a form of generic equality which is inferred for certain types, called *equality types,* on which it is possible to define a computable equality relation. However, the standard definition is incomplete in the sense that

there are interesting and useful types which are *not* inferred to be equality types but which nevertheless have a computable equality relation. In this paper, a refinement of the Standard ML system of equality types is introduced and is proven sound and *complete* with respect to the existence of a computable equality. The technique used here is based on an abstract interpretation of ML operators as monotone functions over a three point lattice. It is shown how the equality relation can be defined (as an ML program) from the definition of a type with our equality property. Finally, a sound, efficient algorithm for inferring the equality property which corrects the limitations of the standard definition in all cases of practical interest is demonstrated.

[8] Robert L. Constable. Type theory as a foundation for computer science. In Ito and Meyer [18], pages 226–243.

We discuss some of the boons as well as shortcomings of constructive type theory as a foundation for computer science. Certain new concepts are offered for tailoring these theories to this task including an idea for collecting objects into subtypes and a proposal for using logic variables and treating them as part of the definition of the logic.

[9] Pierre-Louis Curien and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$top reduction in $F_{\leq}$. In Ito and Meyer [18], pages 731–749.

We contribute to the syntactic study of $\mathbf{F}_{\leq}$, a variant of second order $\lambda$-calculus $\mathbf{F}$ which appears as a paradigmatic kernel language for polymorphism and subtyping. The type system of $\mathbf{F}_{\leq}$ has a maximum type Top and bounded quantification. We endow this language with the familiar $\beta$-rules (for terms and types), to which we add extensionality rules: the $\eta$-rules (for terms and types), and a rule (top) which equates all terms of type Top. These rules are suggested by the axiomatization of cartesian closed categories. We show that this theory $\beta\eta\text{top}_{\leq}$ is decidable, by exhibiting an effectively weakly normalizing and confluent rewriting system for it. Our proof of confluence relies on the confluence of a corresponding system $\mathbf{F_1}$ (the extension of $\mathbf{F}$ with a terminal type), and follows a general pattern that we investigate for itself in a separate paper.

[10] Kazuhiro Fuchi and Koichi Furukawa. Role of logic programming in the FGCS project. In Ito and Meyer [18], pages 311–325.

The research of the Fifth Generation Computer Project was conducted based on a single principle: Logic Programming. Logic programming unifies the ideas of retrieval and computation. Both of these can be regarded as forms of deduction. Research shows that logic programming plays a central role in the project. This role is as the foundation of a very high level programming language based on constraint logic programming, and as a formalization of a very powerful concurrent programming language, which also gives specifications for multi-processor architecture.

[11] Paola Giannini and Simona Ronchi Della Rocca. Type inference in polymorphic type discipline. In Ito and Meyer [18], pages 18–37.

No abstract.

[12] Masami Hagiya. From programming-by-example to proving-by-example. In Ito and Meyer [18], pages 387–419.

In the machine learning community, it is widely recognized that there are two fundamentally different approaches to learning: *inductive learning* and *deductive learning*. This paper formalizes both of the approaches to learning under a uniform framework of type theory and investigates the use of higher-order unification to solve learning problems in both.

I first introduce the simply typed $\lambda$-calculus with inductive definitions and give a unification procedure for the calculus. I then formalize a problem of *programming-by-example* (i.e., inductive learning) as a system of equations in the calculus and reformulate existing methods for programming-by-example as restricted versions of the unification procedure.

It is a new attempt to formalize a problem of *proving-by-example* (i.e., deductive learning) as an equation in a typed $\lambda$-calculus. For that purpose, I extend **LF** with inductive definitions and consider a unification procedure for it.

[13] Ryu Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In Ito and Meyer [18], pages 495–512.

In the preceding paper, the author proved that parametric natural models have many categorical data types: finite products, finite coproducts, initial and terminal fixed points. In this paper, we show the second order minimum model is parametric, and thus enjoys the property. In addition to that, we give representation of internal right and left Kan extensions. We also show that extensionally collapsed models of closed types/terms collection are partially parametric, and that they have a part of the categorical data types above.

[14] Susumu Hayashi. Singleton, union and intersection types for program extraction. In Ito and Meyer [18], pages 701–730.

Two type theories, ATT and ATTT, are introduced. ATT is an impredicative type theory closely related to the polymorphic type theory of implicit typing of MacQueen et al. [MPS86]. ATTT is another version of ATT that extends the Girard-Reynolds second order lambda calculus. ATT has notions of intersection, union and singelton types. ATTT has a notion of refinement types as in the type system for ML by Freeman and Pfenning [FP91], plus intersection and union of refinement types and singleton refinement types. We will show how singleton, union and intersection types serve for development of programs without unnecessary codes via a variant of the Curry-Howard isomorphism. More exactly, they give a way to write types as specifications of programs without unnecessary codes which is inevitable in the usual Curry-Howard isomorphism.

[15] Andreas V. Hense. Wrapper semantics of an object-oriented programming language with state. In Ito and Meyer [18], pages 548–568.

The semantics of class inheritance has first been given in operational form (method-lookup-semantics). While this semantics is well suited for implementing object-oriented programming languages, it may conceal the true nature of inheritance. The development of denotational semantics for object-oriented languages has culminated in object creation as fixed point operation. Cook gave a semantics on this basis, using so called *wrappers*. This semantics abstracts from the internal state of objects (*instance variables*).

In this paper we show how wrapper semantics can be extended to an object-oriented programming language *with state* while keeping the structure of the original definitions. For this purpose we define a direct denotational semantics of a small example language. The insertion of state into class definitions can be done before or after the related fixed point operation. The choice of the alternative considerably influences the semantic domains and clauses.

[16] Sachio Hirokawa. Principal type-schemes of BCI-lambda-terms. In Ito and Meyer [18], pages 633–650.

A BCI-$\lambda$-term is a $\lambda$-term in which each variable occurs exactly once. It represents a proof figure for implicational formula provable in linear logic. A principal type-scheme is a most general type to the term with respect to substitution. The notion of "relevance relation" is introduced for type-variables in a type. Intuitively an occurrence of a type-variable $b$ is relevant to other occurrence of some type-variable $c$ in a type $\alpha$, when $b$ is essentially concerned with the deduction of $c$ in $\alpha$. This relation defines a directed graph $G(\alpha)$ for type-variables in the type. We prove that a type $\alpha$ is a principal type-scheme of BCI-$\lambda$-term iff (a), (b) and (c) holds:

(a) Each variable occurring in $\alpha$ occurs exactly twice and the occurrences have opposite sign.

(b) $G(\alpha)$ is a tree and the right-most type variable in $\alpha$ is its root.

(c) For any subtype $\gamma$ of $\alpha$, each type variable in $\gamma$ is relevant to the right-most type variable in $\gamma$.

A type-schemes of some BCI-$\lambda$-term is minimal iff it is not a non-trivial substitution instance of other type-scheme of BCI-$\lambda$-term. We prove that the set of BCI-minimal types coincides with the set of principal type-schemes of BCI-$\lambda$-terms in $\beta\eta$-normal form.

[17] Hardi Hungar. Complexity of proving program correctness. In Ito and Meyer [18], pages 459–474.

The spectrum of a formula is the set of finite data structures in which it is valid. It is known that for some program logics the classes of spectra form complete subclasses of well known complexity classes. This means that for those logics we know how hard it is to *decide* the set of finite models. We extend those results by determining complexity classes corresponding to partial correctness assertions about programs from sublanguages of Clarke's language **L4**.

We proceed to show that syntax-directed proof systems are adequate tools for *proving* partial correctness assertions: It is not more difficult to construct a proof for a valid assertion than to decide its validity. This holds if the programs are simple while-programs or if they belong to some sublanguage of **ALGOL** like **L4**, for which relatively complete proof systems are rather sophisticated.

[18] T. Ito and A. R. Meyer, editors. *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1991.

[19] Trevor Jim and Albert R. Meyer. Full abstraction and the context lemma. In Ito and Meyer [18], pages 131–151.

A general notion of rewriting system of the kind used for evaluating simply typed $\lambda$-terms in Scott's PCF is defined. Any simply typed $\lambda$-calculus with PCF-like rewriting semantics is shown necessarily to satisfy Milner's Context Lemma. A simple argument demonstrates that any denotational semantics which is adequate for PCF and in which certain simple Boolean functionals exist, cannot be fully abstract for *any* extension of PCF satisfying the Context Lemma. An immediate corollary is that Berry's stable domains cannot be fully abstract for any extension of PCF definable by PCF-like rules. Thus, the idea of adding a combinator to PCF analogous to the "parallel-or" combinator which establishes full abstraction for the familiar cpo model cannot be generalized for models such as stable domains.

[20] Bengt Jonsson and Joost N. Kok. Towards a complete hierarchy of compositional dataflow models. In Ito and Meyer [18], pages 204–225.

A dataflow network consists of nodes that communicate by passing data over unbounded FIFO channels. For dataflow networks containing only deterministic nodes, Kahn has presented a simple and elegant semantic model. However, the generalization of this model is not compositional for nondeterministic networks. Past work has shown that compositionality can be attained by models based on traces. In the paper, we investigate trace models of dataflow networks, with the aim of characterizing compositional and non-compositional models. We study several compositional trace models, which differ in whether they model liveness, termination or divergence. We relate the models into a hierarchy, according to their capability to distinguish networks. A hierarchy is called *complete* if any gap between two models in the hierarchy contains no compositional models. Our main contribution is to prove that most of the gaps in our hierarchy do not contain compositional models. Several full abstraction results in the literature follow directly from the gaps in our hierarchy. We also show that by restricting the networks to contain less powerful nondeterministic processes, additional models become compositional. This means that additional models are added to the hierarchy.

[21] Max I. Kanovich. Efficient program synthesis: Semantics, logic, complexity. In Ito and Meyer [18], pages 615–632.

The problem of program synthesis is considered.

1. A computational semantics is introduced for relational knowledge bases. Our semantics naturally arises from practical experience of databases and knowledge bases.

2. It is stated that the corresponding logic coincides exactly with the intuitionistic one.
3. Our methods of proof of the general theorems turn out to be very useful for designing new efficient algorithms.
   In particular, one can construct a program synthesizer that runs in linear space.
   As a corollary, we can explain why there exist programs that solve PSPACE-complete problems “in a reasonable time” despite of their theoretical exponential uniform lower bound.

[22] Jean-Louis Lassez. From LP to LP: Programming with constraints. In Ito and Meyer [18], pages 420–446.

> Constraint methods for problem solving have a long history. Recently the problem of introducing constraints as primitive constructs in programming languages has been addressed. A main task that the designers and implementers of such languages face is to use and adapt the concepts and algorithms from the extensive studies on constraints done in areas such as Mathematical Programming, Symbolic Computation, Artificial Intelligence, Program Verification and Computational Geometry. Borrowing from these areas and synthesizing the various notions leads to an emerging conception of programming with constraints that we will describe here informally.

[23] M. Abadi, M. Burrows, C. Kaufman and B. Lampson. Authentication and delegation with smart-cards. In Ito and Meyer [18], pages 326–345.

> The authentication of users in distributed systems poses special problems because users lack the ability to encrypt and decrypt. The same problems arise when users wish to delegate some of their authority to nodes, after mutual authentication.
>
> In most systems today, the user is forced to trust the node he wants to use. In a more satisfactory design, the user carries a smart-card with sufficient computing power to assist him; the card provides encryption and decryption capabilities for authentication and delegation.
>
> Authentication is relatively straightforward with a powerful enough smart-card. However, for practical reasons, protocols that place few demands on smart-cards should be considered. These protocols are subtle, as they rely on fairly complex trust relations between the principals in the system (users, hosts, services). In this paper, we discuss a range of public-key smart-card protocols, and analyze their assumptions and the guarantees they offer.

[24] Mark Ryan, José Fiadeiro and Tom Maibaum. Sharing actions and attributes in modal action logic. In Ito and Meyer [18], pages 569–593.

> Distributed systems may be specified in Structured Modal Action Logic by decomposing them into *agents* which interact by sharing *attributes* (memory) as well as *actions*.
>
> In the formalism we describe, specification texts denote theories, and theories denote the set of semantic structures which satisfy them. The semantic structures are Kripke models, as is usual for modal logic. The "possible worlds" in a Kripke model are the states of the agent, and there is a separate relation on the set of states for each action term.
>
> Agents potentially share actions as well as attributes in a way controlled by locality annotations in the specification texts. These become locality axioms in the logical theories the texts denote. These locality axioms provide a refined way of circumscribing the effects of actions.
>
> Safety and liveness conditions are expressed (implicitly) by deontic axioms, which impose obligations and deny permissions on actions. We show that "deontic defaults" exist so that the specifier need not explicitly grant permissions or avoid obligations in situations where normative behaviour is not an issue.

[25] John C. Mitchell. On abstraction and the expressive power of programming languages. In Ito and Meyer [18], pages 290–310.

> We present a tentative theory of programming language expressiveness based on reductions (language translations) that preserve observational equivalence. These are called "abstraction-preserving" because of a connection with a definition of "abstraction" or "information-hiding" mechanism. If there is an abstraction-preserving reduction from one language to another, then essentially every function on natural numbers that is definable in the first is also definable in the second. Moreover, regardless of the set of first-order functions definable in either language, no programming language with an abstraction mechanism can be reduced to a language without. Since Lisp with user-defined special forms does not have an abstraction mechanism, it is therefore not "universal" in this theory, in spite of the ability to define every partial recursive function on the natural numbers. Several examples and counter-examples to abstraction-preserving reductions are given. We do not know whether there is a natural universal language with respect to abstraction-preserving reduction.

[26] Phil Pfeiffer and Rebecca Parsons Selke. On the adequacy of dependence-based representations for programs with heaps. In Ito and Meyer [18], pages 365–386.

Program dependence graphs (*pdgs*) are popular tools for reasoning about a program's semantics. This report proves two fundamental theorems about the representational soundness of *pdgs* for languages with heap-allocated storage and reference variables. The first, the *Pointer-Language Equivalence Theorem*, asserts that *pdgs* adequately represent a program's threads of computation. These theorems are demonstrated with two new lemmas about the semantics of *pdgs* for languages that lack pointer variables. These lemmas, the *Dynamic Equivalence* and *Dynamic Slicing Theorems*, state that an edge can safely be removed from a program's *pdg* if this edges represents a static dependence that does not arise at run-time.

[27] Wesley Phoa. From term models to domains. In Ito and Meyer [18], pages 88–111.

Let $\mathsf{B}$ be the closed term model of the $\lambda$-calculus in which terms with the same Böhm tree are identified. We investigate which partial equivalence relations (PERs) on $\mathsf{B}$ can be regarded as predomains or domains. Working inside the realizability topos on $\mathsf{B}$'s, such PERs can be regarded simply as sets in a particular model of constructive set theory.

No well-behaved partial order has been identified for any class of PERs; but it is still possible to isolate those PERs which have 'suprema of chains' in a certain sense, and all maps between such PERs in the model preserve such suprema of chains. One can also define what it means for such a PER to have a 'bottom'; partial function spaces provide an example. For these PERs, fixed points of arbitrary endofunctions exist and are computed by the fixed point combinator $\mathsf{y}$. There is also a notion of meet-closure for which all maps are stable.

The categories of predomains are closed under the formation of total and partial function spaces, polymorphic types and convex powerdomains. (Subtyping and bounded quantification can also be modelled.) They in fact form reflective subcategories of the realizability topos; and in this set-theoretic context, these constructions are very simple to describe.

[28] Gordon Plotkin. A semantics for type checking. In Ito and Meyer [18], pages 1–17.

Curry's system for F-deducibility is the basis for implicit type checking for programming languages such as ML. If a natural "preservation of types by conversion" rule is added it becomes undecidable, but complete relative to a variety of model classes. We show completeness for F-deducibility itself, relative to an extended notion of model which validates reduction but not conversion. Both term model and filter model proofs are given, and the extension to polymorphic typing is also considered.

[29] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In Ito and Meyer [18], pages 244–265.

This paper presents a proposal for the definition of a step in the execution of a statechart. The proposed semantics maintains the *synchrony hypothesis*, by which the system is infinitely faster than its environment, and can always finish computing its response before the next stimulus arrives. However, it corrects some inconsistencies present in previous definitions, by requiring global consistency of the step.

[30] Alexander Rabinovich and Boris A. Trakhtenbrot. On nets, algebras and modularity. In Ito and Meyer [18], pages 176–203.

We aim at a unified and coherent presentation of net models for concurrency like Petri nets and dataflow networks from the perspective of modularity and substitutivity. The major goal is to achieve a better understanding of the links between modularity issues for nets and laws (or anomalies) in algebras of processes and algebras of relations. To this end we develop Mazurkiewicz's compositional approach which requires a careful analysis of homomorphisms from algebras of nets into algebras of processes and relations.

[31] John C. Reynolds. The coherence of languages with intersection types. In Ito and Meyer [18], pages 657–700.

When a programming language has a sufficiently rich type structure, there can be more than one proof of the same typing judgement; potentially this can lead to semantic ambiguity since the semantics of a typed language is a function of such proofs. When no such ambiguity arises, we say that the language is *coherent*. In this paper we prove the coherence of a class of lambda-calculus-based languages that use the intersection type discipline, including both a purely functional programming language and the Algol-like programming language Forsythe.

[32] Masahiko Sato. Adding proof objects and inductive definition mechanisms to frege structures. In Ito and Meyer [18], pages 53–87.

> A constructive theory RPT (Reflective Proof Theory) of proofs which has the following three features is introduced. (1) Proofs as objects. (2) Hierarchies of propositions and truths. (3) The mechanisms of inductive definitions of predicates. Three kinds of structures called Frege structures with inductively defined predicates, Frege structures with proof objects and proof structures are also introduced. These structures are obtained by generalizing certain aspects of RPT and they are all closely related to Frege structures.

[33] Dana S. Scott. Will logicians be replaced by machines? (Abstract). In Ito and Meyer [18], page 771.

> Many workers have been and will be soon displaced by machines, and we have to ask whether it is only a matter of time before teachers are as well. The lecture will review the march of technology and how developments are apt to affect teaching and research in many subjects. To be able to understand the emerging situation, we also have to reflect on the nature of studies in the Foundations of Mathematics.

[34] Bernhard Steffen. Data flow analysis as model checking. In Ito and Meyer [18], pages 346–364.

> The paper develops a framework that is based on the idea that modal logic provides an appropriate framework for the specification of data flow analysis (DFA) algorithms as soon as programs are represented as models of the logic. This can be exploited to construct a DFA-*generator* that generates efficient implementations of DFA-algorithms from modal specifications by partially evaluating a specific model checker with respect to the specifying modal formula. Moreover, the use of a modal logic as specification language for DFA-algorithms supports the compositional development of specifications and structured proofs of properties of DFA-algorithms. – The framework is illustrated by means of a real life example: the problem of determining optimal computation points within flow graphs.

[35] Makoto Tatsuta. Monotone recursive definition of predicates and its realizability interpretation. In Ito and Meyer [18], pages 38–52.

> The main aim of the paper is to construct a logic by which we can formalize properties of programs. Inductive definition or

recursive definition plays a very important role for this purpose. Inductive definition has been studied for untyped theories, predicative typed theories and impredicative typed theories. Monotone recursive definition in an untyped theory is studied in this paper. The main point is realizability interpretation of monotone recursive definition.

Untyped predicative theory $\mathbf{TID_0}$ and $\mathbf{TID_1}$ are presented, which have monotone recursive definition of predicates. $\mathbf{TID_1}$ has full monotone recursive definition and $\mathbf{TID_0}$ has only restricted monotone recursive definition. $\mathbf{q}$-realizability interpretation of $\mathbf{TID_0}$ and $\mathbf{TID_1}$ is defined. It is proved that the realizability interpretation of $\mathbf{TID_0}$ is sound and that the realizability interpretation of $\mathbf{TID_1}$ is not sound, though $\mathbf{TID_1}$ and its interpretation seem very natural.

[36] Ralf Treinen. First order data types and first order logic. In Ito and Meyer [18], pages 594–614.

This paper concerns the relation between parameterized first order data types and first order logic. Augmenting first order logic by data type definitions yields in general a strictly stronger logic than first order logic. Some modeltheoretic properties of the new logic are investigated. While the new logic always fulfills the downward Skolem-Löwenheim property, compactness is fulfilled if and only if for the given data type definition the new logic has the same expressive power than first order logic. This last property is shown to be undecidable.

[37] David Walker. $\pi$-calculus semantics of object-oriented programming languages. In Ito and Meyer [18], pages 532–547.

The $\pi$-calculus provides a foundation for the study of computational systems with evolving communication structure. A system is viewed as a collection of agents which may share named communication links. Agents interact by passing to one another along shared links the names of other links. Semantics for a pair of parallel object-oriented programming languages are presented by translation into the $\pi$-calculus. The semantics are compared briefly with existing semantics of related languages.

[38] Shin ya Nishizaki. Programs with continuations and linear logic. In Ito and Meyer [18], pages 513–531.

A programming language with continuations is studied in the framework of Girard's linear logic. The execution of a program with continuations is in general *non-deterministic*: the result of

computation depends on the evaluation strategy, e.g., call-by-value evaluation, call-by-name evaluation, ..., etc. In this paper, we first introduce $\lambda_c^{\rightarrow}$, a programming language with continuations, and then define the translation from $\lambda_c^{\rightarrow}$ to linear logic, which eleminates the non-determinism of $\lambda_c^{\rightarrow}$. The relation between computation of $\lambda_c^{\rightarrow}$ and normalization of linear logic is also shown.