Extensions of Structural Synthesis of Programs

Tarmo Uustalu
Dept of Teleinformatics, The Royal Institute of Technology
Electrum 204, S-164 40 Kista (Stockholm), Sweden
email `tarmo@it.kth.se`

Abstract

Structural synthesis of programs (SSP) is an approach to deductive synthesis of functional programs using types as specifications and based on the Curry-Howard correspondence and on an intensional treatment of the notion of type. The implemented programming environments employing SSP have been based on a fragment of intuitionistic propositional logic (simple type theory) and on a natural-deduction proof system. In the paper, we indicate that the proof search strategy used in these systems is applicable to a variety of natural-deduction proof systems (not necessarily of intuitionistic or propositional logics), and that the object-oriented user front-end specification language of the NUT programming environment can, in fact, be given a useful logical semantics in terms of intuitionistic first-order logic.

1 Introduction

Structural synthesis of programs (SSP) as proposed and described by Mints and Tyugu [MT82] is an approach to deductive synthesis of functional programs using types as specifications and based on the Curry-Howard correspondence and on an intensional treatment of types as descriptors of the conceptual roles of values in computations. The approach has a practical orientation; a feasible balance has been sought between expressiveness of the specification language and efficiency of proof search. The implemented systems of SSP have been based on a fragment of intuitionistic propositional logic (simple type theory). The work on SSP is related to automated software engineering, programming in type theories, and automated theorem proving.

In the paper, we indicate that the two cornerstones of SSP — the particular proof search strategy underlying the synthesis procedure, and idea of using types as specifications in program synthesis — are both special applications of something quite general.

The proof search technique employed in SSP has been described in different formulations by a number of authors: Maslov (the inverse method) [Mas64] (for an accessible exposition, see [Lif89]), Mints (general resolution) [Min90a, Min90b, Min93], Stålmarck (the assure-method) [Stå90]. We present the SSP proof search strategy as an instance of a generic strategy of forward-directed search in natural-deduction calculi where conclusion generation is limited to sequents that potentially may participate in a normal derivation of the goal sequent and where conclusion generation under an extra assumption is tried only if no new conclusions can be derived otherwise and where always the most general conclusion is drawn (unification). In the propositional case, decision of proof-theoretically easy goal formulae (this notion is formalized!) is polynomial-time and specifications one usually would write in practice are proof-theoretically easy.

As an example of taking advantages of the first-orderness in specifications and of the unification mechanism in proof search, we show how the object-oriented user front-end specification language of the NUT system (the most recent system in the Priz family of SSP-based programming environments), which used to be translated into propositional logic, can also be translated into predicate logic. The second translation yields faster proof search and shorter proofs.

The paper is organized as follows. Section 2 is a brief exposition of the kernel ideas of structural synthesis of programs. In Section 3, we define what kind of proof systems we consider to be natural-deduction systems, describe the proof search strategy, and present the main theoretic results about the strategy (concerning completeness and complexity). In Section 4, we present a semantics of the specification language of NUT in terms of intuitionistic first-order predicate logic. In Section 5, we formulate our conclusions.

2 Structural Synthesis of Programs

As we stated already, structural synthesis of programs is an approach to deductive synthesis of functional programs where types are used as specifications. It is based on the Curry-Howard correspondence [How80] and on an intensional treatment of types as descriptors of the conceptual roles of values in computations. The idea was proposed in the beginning of 1980s by Mints and Tyugu [MT82, MT90], and has since then been tried out, in combination with other ideas, in a family of programming environments (Priz, ExpertPriz, CPriz, NUT) [MT88, Tyu91, Tyu94]. Since the theoretical justification of the technique, together with descriptions of the concrete programming systems and examples of applications can be found in the cited sources, we will only mention some essentials.

Structural synthesis was intended as a software engineering technique, for application in programming-in-the-large. Thus, in structural synthesis, one usually does not synthesize programs from the scratch, but from knowing some realizations of some specifications. Such background knowledge about specifications and programs is called a computational model. Given the knowledge that specifications $s_1, \ldots, s_n$ are realized by programs $p_1, \ldots, p_n$, respectively, and given a specification s (a goal specification), the synthesis of a program p realizing s proceeds in two steps. At the first step, a proof is sought for s in the intuitionistic theory one obtains by regarding $s_1, \ldots s_n$ as extralogical axioms. Any such proof represents the structure of some program realizing s. At the second step, this program is extracted from the proof, using the realizability semantics of intuitionistic logic and the knowledge about realizations for the extralogical axioms.

Structural synthesis can be regarded as a kind of logic programming on functional values; one might perhaps call it programming with types. Similarly to conventional logic programming, which we will call relational programming for a moment in order to avoid confusion, computation in programming with types proceeds by proof search, and the result of a computation is something extracted from the proof found. The difference lies in what namely is extracted. In relational programming, the goal is an existential formula and the value extracted is a witness of its validity (semantically speaking, the value extracted is a witness of non-emptiness of a relation). In programming with types, the value extracted is a realization of the goal formula (semantically speaking, a witness of non-emptiness of a type).

Quite obviously, the work on structural synthesis is related to work on programming in type theories as advocated in e.g. [NPS90]. The difference lies in the emphases. In programming in type theories, the main interests are verification of programs against specifications, and transformational development of specifications into programs. The type languages used are of a very rich expressive power. In structural synthesis, the main objective has been efficient automatic synthesis, which obviously sets very restrictive constraints on the type language.

3 Proof Search in Natural-Deduction Proof Systems for Propositional Logics

In this section, we will show that the proof search strategy employed in SSP is applicable to a variety of natural-deduction proof systems of propositional logics; namely, to all those enjoying some form of a subformula property or some other property ensuring that the number of different formulae that potentially may be needed in a proof of a given formula is polynomial in the size of this formula.

We shall present the strategy and the theoretic results about it on the example of two proof systems: the "canonical" natural-deduction calculus for intuitionistic propositional logic, formulated by Gentzen and Prawitz [Pra65], and the Dilemma-system (a natural-deduction-inspired proof system for classical propositional logic), formulated by Stålmarck [Stå90].

SSP is an example of an application of the proof search strategy to the first proof system. Independently, Stålmarck formulated the strategy and the results for the second proof system and has applied it in verification of state-transition systems (where both implementations and specifications of such systems are formalized in the language of propositional logic). His method has been patented for commercial use.

Natural-deduction proof systems are a special case of sequent-based proof systems. Rules of such proof systems act upon sequents, which are figures of the form $\Gamma \vdash \Delta$, where Γ, Δ are finite sets of formulae. The left-hand side of a sequent is called its antecedent, whereas the right hand side of it is called its succedent. In every rule application, some formulae (more exactly, formula occurrences) in the premisses and in the conclusion appear as active, whereas the others serve as a passive context.

There exists no universally accepted definition of a natural-deduction proof system. For us and here, natural-deduction proof systems are sequent-based proof systems satisfying the following conditions:

- the succendents of sequents contain exactly one formula;
- no proof rule contains active formulae in the antecedent of the conclusion;
- just a few rules contain active formulae in the antecedents of the premisses.

The rules that have all their active formulae in the succedents of the premisses and of the conclusion are called simple rules. The rules that have some active formulae in the antecedents of the premisses are called assumption rules.

The kernel of a typical natural-deduction proof system is formed by introduction and elimination rules of the logical operators present in the language of the given logic. The active formulae of a rule for a certain operator are a formula with the given operator as the outermost operator (the major formula) and its immediate subformulae (the minor formulae). In an introduction rule, the major formula appears in the succedent of the conclusion, in an elimination rule, the major formula appears in the succedent of one of the premisses.

Intuitionistic propositional natural deduction

Iden	$\dfrac{}{\boxed{A} \vdash \boxed{A}}$	Weak	$\dfrac{\Gamma \vdash C}{\Gamma, D \vdash C}$
$\wedge_I$	$\dfrac{\Gamma \vdash \boxed{A} \quad \Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \wedge B}}$	$\wedge_E$	$\dfrac{\Gamma \vdash \boxed{A \wedge B}}{\Gamma \vdash \boxed{A}} \quad \dfrac{\Gamma \vdash \boxed{A \wedge B}}{\Gamma \vdash \boxed{B}}$
$\vee_I$	$\dfrac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{A \vee B}} \quad \dfrac{\Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \vee B}}$	$\vee_E$	$\dfrac{\Gamma \vdash \boxed{A \vee B} \quad \Gamma, \boxed{A} \vdash C \quad \Gamma, \boxed{B} \vdash C}{\Gamma \vdash C}$
$\supset_I$	$\dfrac{\Gamma, \boxed{A} \vdash \boxed{B}}{\Gamma \vdash \boxed{A \supset B}} \quad \dfrac{\Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \supset B}}$	$\supset_E$	$\dfrac{\Gamma \vdash \boxed{A \supset B} \quad \Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{B}}$
$\neg_I$	$\dfrac{\Gamma, \boxed{A} \vdash \bot}{\Gamma \vdash \boxed{\neg A}}$	$\neg_E$	$\dfrac{\Gamma \vdash \boxed{\neg A} \quad \Gamma \vdash \boxed{A}}{\Gamma \vdash C}$

In this proof system, $\supset_I$, $\neg_I$ and $\vee_E$ are assumption rules; all other rules are simple. This proof system enjoys the following subformula property: only subformulae of the goal sequent are potentially needed in a proof; moreover, a formula A is potentially necessary in the position of a major formula of an application of an I-rule (resp. E-rule) only if the goal sequent contains a positive (resp. negative) occurrence of A. A formula A is potentially necessary as an antecedent formula only if the goal sequent contains a positive occurrence of some formula of the form $A \supset B$ or $\neg A$ or a negative occurrence of some formula of the form $A \vee B$ or $B \vee A$.

The Dilemma-system

Iden	$\dfrac{}{\boxed{A}\vdash\boxed{A}}$	Weak	$\dfrac{\Gamma\vdash C}{\Gamma, D\vdash C}$
Dilemma	$\dfrac{\Gamma,\boxed{A}\vdash C \quad \Gamma,\sim\boxed{A}\vdash C}{\Gamma\vdash C}$	Contrad	$\dfrac{\Gamma\vdash\boxed{A}\quad \Gamma\vdash\sim\boxed{A}}{\Gamma\vdash C}$
$\wedge_I$	$\dfrac{\Gamma\vdash\boxed{A}\quad \Gamma\vdash\boxed{B}}{\Gamma\vdash\boxed{A\wedge B}}$	$\wedge_E$	$\dfrac{\Gamma\vdash\boxed{A\wedge B}}{\Gamma\vdash\boxed{A}}\qquad\dfrac{\Gamma\vdash\boxed{A\wedge B}}{\Gamma\vdash\boxed{B}}$
$\sim\wedge_I$	$\dfrac{\Gamma\vdash\sim\boxed{A}}{\Gamma\vdash\sim\boxed{A\wedge B}}\qquad\dfrac{\Gamma\vdash\sim\boxed{B}}{\Gamma\vdash\sim\boxed{A\wedge B}}$	$\sim\wedge_E$	$\dfrac{\Gamma\vdash\sim\boxed{A\wedge B}\quad \Gamma\vdash\boxed{B}}{\Gamma\vdash\sim\boxed{A}}$
			$\dfrac{\Gamma\vdash\sim\boxed{A\wedge B}\quad \Gamma\vdash\boxed{A}}{\Gamma\vdash\sim\boxed{B}}$
$\vee_I$	$\dfrac{\Gamma\vdash\boxed{A}}{\Gamma\vdash\boxed{A\vee B}}\qquad\dfrac{\Gamma\vdash\boxed{B}}{\Gamma\vdash\boxed{A\vee B}}$	$\vee_E$	$\dfrac{\Gamma\vdash\boxed{A\vee B}\quad \Gamma\vdash\sim\boxed{B}}{\Gamma\vdash\boxed{A}}$
			$\dfrac{\Gamma\vdash\boxed{A\vee B}\quad \Gamma\vdash\sim\boxed{A}}{\Gamma\vdash\boxed{B}}$
$\sim\vee_I$	$\dfrac{\Gamma\vdash\sim\boxed{A}\quad \Gamma\vdash\sim\boxed{B}}{\Gamma\vdash\sim\boxed{A\vee B}}$	$\sim\vee_E$	$\dfrac{\Gamma\vdash\sim\boxed{A\vee B}}{\Gamma\vdash\sim\boxed{A}}\qquad\dfrac{\Gamma\vdash\sim\boxed{A\vee B}}{\Gamma\vdash\sim\boxed{B}}$
$\supset_I$	$\dfrac{\Gamma\vdash\sim\boxed{A}}{\Gamma\vdash\boxed{A\supset B}}\qquad\dfrac{\Gamma\vdash\boxed{B}}{\Gamma\vdash\boxed{A\supset B}}$	$\supset_E$	$\dfrac{\Gamma\vdash\boxed{A\supset B}\quad \Gamma\vdash\sim\boxed{B}}{\Gamma\vdash\sim\boxed{A}}$
			$\dfrac{\Gamma\vdash\boxed{A\supset B}\quad \Gamma\vdash\boxed{A}}{\Gamma\vdash\boxed{B}}$
$\sim\supset_I$	$\dfrac{\Gamma\vdash\boxed{A}\quad \Gamma\vdash\sim\boxed{B}}{\Gamma\vdash\sim\boxed{A\supset B}}$	$\sim\supset_E$	$\dfrac{\Gamma\vdash\sim\boxed{A\supset B}}{\Gamma\vdash\boxed{A}}\qquad\dfrac{\Gamma\vdash\sim\boxed{A\supset B}}{\Gamma\vdash\sim\boxed{B}}$
$\neg_I$	$\dfrac{\Gamma\vdash\sim\boxed{A}}{\Gamma\vdash\boxed{\neg A}}$	$\neg_E$	$\dfrac{\Gamma\vdash\boxed{\neg A}}{\Gamma\vdash\sim\boxed{A}}$
$\sim\neg_I$	$\dfrac{\Gamma\vdash\boxed{A}}{\Gamma\vdash\sim\boxed{\neg A}}$	$\sim\neg_E$	$\dfrac{\Gamma\vdash\sim\boxed{\neg A}}{\Gamma\vdash\boxed{A}}$

In this proof system, Dilemma is the single assumption rule; all other rules are simple. This proof system enjoys the following subformula property: only subformulae of the goal sequent are potentially needed in a proof; moreover, a formula A is potentially necessary in the position of a major formula of an application of an I- or $\sim E$- rule (resp. E- or $\sim I$-rule) only if the goal sequent contains a positive (resp. negative) occurrence of A. A formula A is potentially necessary as an antecedent formula only if it is a propositional variable occurring in the goal sequent.

We define the degree of assumption interaction of a proof as the maximum number of formulae in the antecedents of the sequents in the proof. We define an i-easy sequent as a formula sequent which has a proof with degree of assuption interaction $\leq i$. We define an i-hard sequent as a sequent which has no proof with degree of assumption interaction $< i$.

The proof search strategy exploits the subformula property, and its key ideas are the following:

- the search is forward-directed: one generates sequents that potentially may be needed in a proof of the goal sequent;
- a proof with minimum degree of assumption interaction is sought; i.e. at first, all potentially necessary 1-easy sequents are generated, then all potentially necessary 2-easy sequents are generated, etc., until the goal sequent is arrived at; this means that the simple rules are always applied in the first place, and the assumption rules are applied only if the simple rules cannot longer yield anything new and potentially necessary;
- for every potentially necessary antecent that is taken into consideration during the search, two datastructures are initialized and subsequently kept up-to-date: a bitvector which tells which potentially necessary sequents with the given antecedent have been generated already, and a queue of potentially necessary simple rule applications which have not yet been tried with the given antecedent, but whose premisses have been generated already;
- atomic names are invented for the subformulae of the goal formulae; instead of working with subformulae directly, one works with their names (to my knowledge, this idea was first put forward by Tseitin [Tse68]);

Two important results can be stated about the proof search strategy:

- it is complete, i.e. if a goal sequent is provable, then the search procedure finds a proof for it;
- the time-complexity of finding a proof of an i-easy goal sequent is

  $$O(NL^i)$$

  where

– N is the number of formulae potentially necessary in the succedents in a proof of the goal sequent, i.e. the number of subformulae of the goal formula (this is proportional to the length of the goal sequent),

– L is the number of formulae potentially necessary in the antecedents in a proof of the goal sequent; in the case of intuitionistic natural deduction, this is the number of implications and negations that occur positively in the goal sequent plus twice the number of disjunctions that occur negatively in the goal sequent; in the case of the Dilemma-system, this is the number of propositional variables occurring the goal sequent; in both cases, $N < L$.

The general decision problem is well-known to be NP-complete for classical propositional logic and PSPACE-complete for intuitionistic propositional logic. The last result tells us that, despite of this, proof search is polynomial-time for both logics, if we can be sure that we do not need high degrees of assumption interaction. Fortunately, it turns out that the goal sequents arising from application problems (tasks of program synthesis in the case of intuitionistic natural deduction, or tasks of state-transition system verification in the case of the Dilemma-system), although often large, are seldom very entangled (since they are usually based on a simple design idea), and, if provable, possess proofs of a quite low degree of assumption interaction. The experience shows that a bound of 2..4 leaves the proof search procedure "practically complete".

4 A Predicate Logic Semantics of an Object-Oriented Structural Specification Language

The NUT system, mentioned in Section 2, uses a specification language built upon the following idioms and principles of object-oriented programming:

- the primary entities of the language are objects; objects can be primitive or compound;
- every primitive object signifies a value with a certain conceptual role (being thus comparable with a "logical variable" in of a Prolog-like language);
- every compound object is associated to a finite set of its components, which are objects in their own right and are identified via component names (component a of object x is denoted as $x.a$ — the usual "dot-notation");
- every object is an instance of some class, classes being templates of objects with similar properties;

- all primitive objects are instances of the predefined class `prim` (in the real implementation, there are more primitive classes— different sorts of primitive values, such as numerics, strings, booleans etc.—, a detail that we shall ignore here);
- compound objects are instantiated from user-defined classes; the class of a compound object determines the names and classes of the components of the object, and may impose a number of realizabilities and definitional equalities on the deep components of the object[1];
- a user-defined class may have superclasses from where it inherits properties; inheritance in NUT is cumulative, not overriding.

From the point of view of specification and synthesis, primitive objects are types, and the object hierarchy is just a way of modular organization of the world of types. Every object is associated to a type variable.

Language

In the declarative part of the NUT language, the specification of the computational domain is presented as a collection of class definitions.

A class definition consists of a finite number of declarations of various kinds. NUT allows following kinds of declarations:

Superclass declarations	`super` cl'	where cl' is a class name
Component declarations	`comp` $a : cl'$	where a is a component name, and cl' is a class name
Equality declarations	`rel` $b_1 = b_2$	where b_1, b_2 are deep component names
Realizability declarations	`rel` $f\ \{t\}$	where f is a propositional formula built of deep component names, and t is a functional expression

Informal semantics

Given a definition of some class cl. Declarations in the definition have the following semantics:

[1]A deep component of an object is, by definition, a component or a deep component of a component of the object.

`super` cl'	means that every instance x of class cl has all properties of instance of class cl'
`comp` $a : cl'$	means that every instance x of class cl has a component a — object $x.a$ — which is an instance of class cl'
`rel` $b_1 = b_2$	means that, for every instance x of class cl, the primitive objects (types) $x.b_1$ and $x.b_2$ are definitionally equal
`rel` f $\{t\}$	means that, for every instance x of class cl, the formula (type) $x.f$ is realized by term t, where $x.f$ is a shorthand notation for the formula arising from f by prefixing every deep component name b in f with x

Translation into propositional logic

Formalization of the informal semantics in terms of propositional logic (simple theory of types) is straightforward.

Given a computational model, consisting of definitions of classes $cl_1, \ldots, cl_n$, the propositional logic semantics of these classes wrt the computational model are sets of realizability assertions defined as follows:

$$
\begin{aligned}
\mathbf{clsem}(\texttt{prim}) &= \emptyset \\
\mathbf{clsem}(cl) &= \bigcup\{decl \in \mathbf{cldef}(cl) \mid [\![decl]\!]_{cl}\} \\
[\![\texttt{super } cl']\!]_{cl} &= \mathbf{clsem}(cl') \\
[\![\texttt{comp } a : cl']\!]_{cl} &= a.\mathbf{clsem}(cl') \\
[\![\texttt{rel } b_1 = b_2]\!]_{cl} &= \{b_1 \equiv b_2 \ \{(\mathrm{id}, \mathrm{id})\}\} \\
[\![\texttt{rel } f \ \{t\}]\!]_{cl} &= \{f \ \{t\}\}
\end{aligned}
$$

where $a.\mathbf{clsem}(cl')$ is a shorthand notation for the set of realizability assertions obtained from $\mathbf{clsem}(cl')$ by prefixing all deep component names in formulae in $\mathbf{clsem}(cl')$ with a.

Finding a term that realizes $x.f$ for a generic instance x of class cl amounts to proving f from $\mathbf{clsem}(cl)$.

A similarly-formulated semantics of the Priz system (where differently from NUT, there are no classes; instead of instantiation from classes, cloning from objects is used) can be found in [MST91].

Translation into predicate logic

In the predicate logic semantics, we represent objects and deep component names by means of terms, and classes by means of binary predicates.

Given a computational model, consisting of definitions of classes $cl_1, \ldots, cl_n$, the predicate logic semantics of these classes are sets of realizability assertions defined as follows:

$$
\begin{array}{lcl}
\mathbf{clsem}(\texttt{prim}) & = & \emptyset \\
\mathbf{clsem}(cl) & = & \bigcup\{decl \in \mathbf{cldef}(cl) \mid [\![decl]\!]_{cl}\} \\
[\![\texttt{super}\ cl']\!]_{cl} & = & \{cl(x,z) \equiv cl'(x(cl),z)\ \{(\mathrm{id},\mathrm{id})\}\} \\
[\![\texttt{comp}\ \ a : cl']\!]_{cl} & = & \{cl(x,a.z) \equiv cl'(x(cl).a,z)\ \{(\mathrm{id},\mathrm{id})\}\} \\
[\![\texttt{rel}\ \ b_1 = b_2]\!]_{cl} & = & \{cl(x,b_1) \equiv cl(x,b_2)\ \{(\mathrm{id},\mathrm{id})\}\} \\
[\![\texttt{rel}\ \ f\ \{t\}]\!]_{cl} & = & \{cl(x,f)\ \{t\}\}
\end{array}
$$

where $cl(x, f)$ is a shorthand notation for the formula obtained from f by replacing every deep component name b in f with $cl(x, b)$.

All formulae above are implicitly universally closed wrt variables x, y.

Finding a term that realizes $x.f$ for a generic instance x of class cl amounts to proving $cl(x, f)$ from $\mathbf{clsem}(cl_1) \cup \ldots \cup \mathbf{clsem}(cl_n)$. Intuitively, provability of $cl(x_0, b)$ means that if object x_0 is of class c then object $x_0.b$ has a value.

Observe that the formula-translations of superclass and component declarations `super` cl' and `comp` $a : cl'$ and are skolemized versions of the formulae $cl(x, z) \equiv \exists y.cl'(y, z)$ and $cl(x, a.z) \equiv \exists y.cl'(y, z)$, respectively.

These two translations of the user front-end specification language into logical languages can be easily seen to be equivalent in the sense that the same synthesis tasks will be solvable and will lead to the same programs. The difference is that if one uses the first translation, synthesis will be done on the deep components of the generic instance x of class cl; if one uses the second translation, synthesis will be essentially done on their classes. The second translation yields faster proof search and shorter proofs, if x has many deep components of same classes.

5 Conclusions

We have shown that the proof search strategy employed in SSP is an instance of a generic strategy applicable to natural-deduction proof systems for various propositional logics (classical or intuitionistic, with modal, fixpoint operators etc.). For us, natural-deduction proof systems are sequent-based proof systems where succedents always consist of exactly one formula, where no proof rule contains active formulae in the antecedent of the conclusion, and where just a few rules may contain active formulae in the antecedents of the premisses. We have noted that the allowed degree of assumption interaction in proofs is a natural measure of hardness of tautologies from the point of view of at least two applications: synthesis of functional programs (by proof search in intuitionistic propositional logic) and verification of state-transition systems (by proof search in classical propositional logic). We have shown that the strategy ensures a polynomial time complexity of proof search if a bound is set on the allowed degree of assumption interaction. The applications of modal and fixpoint logics in structural synthesis are a direction for future study which we have started to look into.

We have also shown that the object-oriented structural specification language which has been employed in the most recent PRIZ-type system — NUT —, and used to be semantically explained terms of intuitionistic propositional logic, has a more concise semantic explanation in terms of intuitionistic predicate logic.

References

[How80] W. A. Howard. The formula-as-types of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, London, 1980.

[Lif89] Vladimir Lifshitz. What is the inverse method? *Journal of Automated Reasoning*, 5(1):1–23, 1989.

[Mas64] S. Yu. Maslov. The inverse method for establishing deducibility in classical predicate calculus. *Soviet Math. Doklady*, 5:1420–1423, 1964.

[Min90a] Grigory Mints. Gentzen-type systems and resolution rule, Part I: Propositional logic. In P. Martin-Löf and G. Mints, editors, *Proc. Int'l Conf. on Computer Logic, COLOG-88, Tallinn, Dec 1988*, pages 198–231, Berlin, 1990. Springer-Verlag. LNCS 417.

[Min90b] Grigory Mints. Gentzen-type systems and resolution rule, Part II: Predicate logic. In *Logic Colloquium '90*, 1990.

[Min93] Grigory Mints. Resolution calculus for the first order linear logic. *Journal of Logic, Language, and Information*, 2(1):59–83, 1993.

[MST91] Grigory Mints, Jan M. Smith, and Enn Tyugu. Type-theoretic semantics of some declarative languages. In J. Bārzdiņš and D. Bjørner, editors, *Baltic Computer Science: Selected Papers*, pages 18–32, Berlin, 1991. Springer-Verlag. LNCS 502.

[MT82] Grigory Mints and Enn Tyugu. Justification of the structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[MT88] Grigory Mints and Enn Tyugu. The programming system PRIZ. *Journal of Symbolic Computation*, 5(4):359–375, 1988.

[MT90] Grigory Mints and Enn Tyugu. Propositional logic programming and the PRIZ system. *Journal of Logic Programming*, 9(2–3):179–193, 1990.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Clarendon Press, Oxford, 1990.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study.* Almqvist & Wiksell, Stockholm, 1965. Stockholm Studies in Philosophy 3.

[Stå90] Gunnar Stålmarck. The Assure method: A proof procedure for propositional logic. Technical Report 1, Logikkonsult NP AB, Stockholm, 1990.

[Tse68] G. S. Tseytin. On the complexity of propositional calculus. *Zapiski Nauchnykh Seminarov LOMI*, 8:235–259, 1968. In Russian.

[Tyu91] Enn Tyugu. Three new-generation software environments. *Communications of the ACM*, 34(6):46–59, 1991.

[Tyu94] Enn Tyugu. Using classes as specifications for automatic construction of programs in the NUT system. *J. Automated Software Engineering*, 1(3–4):315–334, 1994.