Intuitionistic Ancestral Logic

Liron Cohen*
Tel Aviv University, Israel
Robert L. Constable†
Cornell University, Ithaca, NY, USA

Abstract

In this paper we define pure *intuitionistic Ancestral Logic* (iAL), extending pure intuitionistic First-Order Logic ($iFOL$). This logic is a *dependently typed abstract programming language* with computational functionality beyond $iFOL$ given by its realizer for the *transitive closure, TC*. We derive this operator from the natural type theoretic definition of TC using intersection. We show that provable formulas in iAL are uniformly realizable, thus iAL is sound with respect to constructive type theory. We further show that iAL subsumes Kleene Algebras with tests and thus serves as a natural programming logic for proving properties of program schemes. We also extract schemes from proofs that iAL specifications are solvable.

1 Introduction

Type theories implemented by proof assistants have become highly effective as specification languages for a wide range of computational tasks, from operating systems and compiler verification [26, 9] to the synthesis of correct-by-construction distributed protocols [36]. These type theories are rich logical systems which are difficult to grasp all at once. It is therefore interesting to see how they can be built from the ground up, starting with first-order logic as is the practice for set theory. This turns out to be quite challenging in the case of the *constructive type theories*. It was not known until quite recently whether intuitionistic first-order logic is complete with respect to its intended semantics [13]. In this article, we take another step toward an explanation of constructive type theory that is closer to the pattern known for set theory and other foundational theories of mathematics such as Peano Arithmetic (PA).

Pure First-Order Logic (FOL) is one of the most widely studied and taught systems of logic. There are many excellent textbooks that present

*liron.cohen@math.tau.ac.il
†rc@cs.cornell.edu

it. [1] It is the base logic in which two of the most studied mathematical theories, Peano Arithmetic (PA) and Zermelo/Fraenkel set theory with choice (ZFC), are presented. The intuitionistic versions of these systems, $iFOL$, Heyting Arithmetic (HA), Intuitionistic ZF (IZF) [15] and the related CZF [2], are also well studied. These intuitionistic logics are important in constructive mathematics, linguistics, philosophy and especially in computer science. Computer scientists exploit the fact that intuitionistic theories can serve as programming languages [6, 30] and that $iFOL$ can be read as an abstract programming language with dependent types.

In the famous paper with the telling name "On the Unusual Effectiveness of Logic in Computer Science" [18], it is forcefully noted that "at present concepts and methods of logic occupy a central place in computer science, insomuch that logic has been called 'the calculus of computer science' [29]". To demonstrate this claim, this paper then studies an impressive (yet explicitly non-exhaustive) list of applications of logics in different areas of computer science: descriptive complexity; database query languages; applications of constructive type theories; reasoning about knowledge; program verification and model checking. But *what* logic has such effectiveness? A simple check of the above list of applications from [18] reveals that first-order logic is sufficient for none of them. All these examples indicate that the crucial shortcoming of first-order logic is its inability to provide inductive definitions in general, and the notion of the transitive closure of a given binary relation in particular.

Since we are interested in natural extensions of $iFOL$ that clearly reveal the duality between logic and programming, and can capture general logical principles that have applicable computational content, it seems natural to investigate an intuitionistic version of *Ancestral Logic* (AL), which is a well known extension of FOL (e.g., [4, 10, 27]) appropriate for defining the transitive closure of binary relations[2]. In this paper we develop an intuitionistic version of AL, iAL, as a refinement of AL and an extension of $iFOL$, capable of giving computational explanations of the same commonly occurring fundamental notions. We believe that rather than $iFOL$, iAL should be taken as the basic logic which underlies most applications of logic to Computer Science. Many proofs in iAL turn out to have interesting computational content that exceeds that of $iFOL$ in ways of interest to computer scientists. We prove that iAL is sound with respect to constructive type theory by showing that provable formulas are *uniformly realizable*. Furthermore, we show that iAL subsumes Kleene Algebras with tests [23] and thus serves as a natural programming logic for proving properties of program schemes. We also extract schemes from proofs that iAL specifications are solvable.

[1] We use the term *pure* to indicate that equality, constants, and functions are not built-in primitives.

[2] Ancestral Logic is also sometimes called Transitive Closure Logic in the literature.

We adopt the presentation of $iFOL$ from *Intuitionistic Completeness of First-Order Logic* [13] where the computational content is made explicit using evidence semantics based on the propositions-as-types principle [31] aka the Curry Howard isomorphism [37]. A formal semantics of the logic we present could be based on extensional constructive type theories such as Intuitionistic Type Theory (ITT) [31] or Constructive Type Theory (CTT) [12, 3, 8]. However, the precise details of the semantical metatheory are not that critical to our results, so we remain informal. For other notions of truth and validity, one can refer to the accounts given in [38].

2 The System *iFOL*

2.1 Realizability Semantics for *iFOL*

This section reviews the semantics of evidence for pure $iFOL$ along the lines of [13]. This is just a compact type theoretic restatement of the propositions-as-types realizability semantics given in [31, 30, 12]. This semantics plays an important role in building correct-by-construction software and in the semantics of strong constructive typed systems, such as Computational Type Theory (CTT) [3], Intuitionistic Type Theory (ITT) [31], Intensional-ITT [8, 35], the Calculus of Inductive Constructions (CIC) [7], and Logical Frameworks such as Edinburgh LF [19].[3] The basic idea behind the semantics is that constructive proofs provide evidence terms (also called realizers) for the propositions they prove, and these realizers allow to directly extract programs from the proofs.

Let $\mathcal{L}$ be a first-order signature of predicates $P_i^{n_i}$ (with arity n_i) over a domain D of individuals of a model $\mathcal{M}$ for $\mathcal{L}$. The domain of discourse, D, can be any constructive type, $[D]_{\mathcal{M}}$.[4] Every formula A over $\mathcal{L}$ is assigned a type of objects denoted $[A]_{\mathcal{M}}$, called the *evidence* for A with respect to $\mathcal{M}$. We normally leave off the subscript $\mathcal{M}$ when there is only one model involved. Below is how evidence is defined for the various kinds of first-order propositional functions. The definition will also implicitly provide a syntax of first-order formulas.

Definition 1. ***(First-order formulas and their evidence)***

- **atomic propositional functions** $P_i^{n_i}$ are interpreted as functions from D^{n_i} into $\mathbb{P}$ the type of propositions, and for the atomic propo-

[3] All of these logics have been implemented by proof assistants such as Agda, Coq, Nuprl, and Twelf.

[4] As a first approximation readers can think of types as *constructive sets* [5]. Peter Aczel [1] shows how to interpret constructive sets as types in ITT [30], and this approach was implemented in MetaPRL for CTT by Hickey [20]. Intuitionists might refer to *species* instead. We do not analyze the structure of the domain further and do not examine the equality relation on the type when dealing with the pure first-order theory, as is standard practice.

sition $P_i^{n_i}(a_1, ..., a_{n_i})$, the basic evidence must be supplied, say by objects p_i. In the uniform treatment, we consider all of these objects to be equal, and we denote them by the unstructured atomic element $\star$.[5] Thus if an atomic proposition is known by atomic evidence, the evidence is the single element $\star$ of the unit type, $\{\star\}$.[6]

- **conjunction** $[A \wedge B] \ = \ [A] \times [B]$, the Cartesian product.
- **existential** $[\exists x.B(x)] \ = \ x : [D]_{\mathcal{M}} \times [B(x)]$, the dependent product.
- **implication** $[A \Rightarrow B] \ = \ [A] \rightarrow [B]$, the function space.[7]
- **universal** $[\forall x.B(x)] \ = \ x : [D]_{\mathcal{M}} \rightarrow [B(x)]$, the dependent function space.
- **disjunction** $[A \vee B] \ = \ [A] + [B]$, disjoint union.
- **false** $[False] \ = \ \emptyset$ the void type.

Negation is defined by $\neg A := A \Rightarrow False$.

It is easy to prove classically that a formula A is satisfied in a model $\mathcal{M}$ if and only if there is evidence in $[A]_{\mathcal{M}}$ [13]. This shows that we can read this evidence semantics classically, and it will correspond to Tarski's semantics for FOL and AL.

2.2 Proof System for *iFOL* over Domain *D*

We next present the proof system $iFOL$ adopting the presentation style from [13] where the computational content is made explicit using evidence semantics based on the propositions-as-types principle [31] aka the Curry Howard isomorphism [37]. The rules of the system are presented in the "top down style" (also called refinement style) in which the goal comes first and the rule name with parameters generates subgoals. Thus the sequent style trees are grown with the root at the top. This top down, goal oriented style is common to all of the proof assistants which work in the highly successful tactic mechanism of the Edinburgh LCF proof assistant [16], and it is also compatible with the style for rules and proofs used in the Nuprl book [12]. Now there are many thousands of formal proofs done in that style, probably more than in any other style. It is impossible in practice to create proofs

[5] Officially this can be done using the set type $\{sq(p_i)|P_i^{n_i}(a_1, ..., a_{n_i})\}$, where $sq(p_i)$ "squashes" the evidence p_i to $\star$ [11].

[6] It might seem that we should introduce atomic evidence terms that might depend on parameters, say $p(x, y)$ as the *atomic evidence* in the atomic proposition $P(x, y)$ but this is unnecessary and uniformity would eliminate any significance to those terms. In CTT and ITT, the evidence for atomic propositions such as equality and ordering is simply an unstructured term such as $\star$.

[7] This function space is interpreted type theoretically and is assumed to consist of *effectively computable deterministic functions.*

that are bottom up, so the rules should match the proof requirements. This style is also compatible with the standard writing style in which a theorem is stated first followed by its proof.

We present the rules in the style of McCarthy's abstract syntax [32]. For each rule we provide a name that is the outer operator of a proof expression with slots to be filled in as the proof is developed. Each sequent in the rules is build so that the left hand side contains the context, and the right hand side contains a type and its evidence term. For the right hand sides we use the notation "type *by* term" (as opposed to "term : type"). The construction rules (often called introduction rules) apply to terms on the right hand side of the sequent and introduce the canonical proof terms. For each of these construction rules, the constructor needs subterms which build the component pieces of evidence. For each connective and quantifier we also have rules for their occurrence on the left of the sequent. These are the rules for decomposing a connective or quantifier. They tell us how to use the evidence that was built with the corresponding construction rules, and the formula being decomposed is always named by a label in the list of hypotheses, so there is a variable associated with each rule application. For a more detailed explanation of the syntax used in the proof rules see [13].

Definition 2. The proof system $iFOL$ is given in Fig. 1.

Note that we use the term d to denote objects in the domain of discourse D. In the classical evidence semantics, we assume that D is non-empty by postulating the existence of some d_0.

2.3 Constructive/Intuitionistic Metatheory

Formal semantics of the intuitionistic logics we present are based on extensional constructive type theories such as Intuitionistic Type Theory (ITT) [31, 30, 35] or Constructive Type Theory (CTT) [12, 3]. The precise details of the semantical metatheory are not that critical to our results, so we remain informal. Critical to constructive type theory and to all of our results is the underlying *computation system*, so we need to mention its key properties. This is essentially the untyped programming language underlying the type theory and thus underlying $iFOL$ and iAL (that is defined in the sequel).

The data and the programs of the computation system are given by closed terms. To say what we mean by a *closed term* we need to examine the structure of terms further. The precise definition of terms provides the syntax of the programming language.

[8] This notation shows that $ap(f; sl_a)$ is substituted for v in $g(v)$. In the CTT logic we stipulate in the rule that $v = ap(f; sl_a)$ *in* B.

[9] In the CTT logic, we use equality to stipulate that $v = ap(f; d)$ *in* $B(v)$ just before the hypothesis $v : B(d)$.

Figure 1: The proof system $iFOL$

And Construction
$H \vdash A \wedge B$ *by* $pair(slot_a; slot_b)$
$H \vdash A$ *by* $slot_a$
$H \vdash B$ *by* $slot_b$

Implication Construction
$H \vdash A \Rightarrow B$ *by* $\lambda(x.slot_b(x))$ *new* x
$H, x : A, H' \vdash B$ *by* $slot_b(x)$

Hypothesis
$H, d : D, H' \vdash \ d \in D$ *by* $obj(d)$

$H, x : A, H' \vdash \ A$ *by* $hyp(x)$

Or Construction
$H \vdash A \vee B$ *by* $inl(slot_l)$
$H \vdash A$ *by* $slot_l$

$H \vdash A \vee B$ *by* $inr(slot_r)$
$H \vdash B$ *by* $slot_r$

Exists Construction
$H \vdash \exists x.B(x)$ *by* $pair(d; slot_b(d))$
$H \vdash d \in D$ *by* $obj(d)$
$H \vdash B(d)$ *by* $slot_b(d)$

All Construction
$H \vdash \forall x.B(x)$ *by* $\lambda(x.slot_b(x))$
$H, x : D, H' \vdash B(x)$ *by* $slot_b(x)$

And Decomposition
$H, x : A \wedge B, H' \vdash G$ *by* $spread(x; l, r.slot_g(l, r))$ *new* l, r
$H, l : A, r : B, H' \vdash G$ *by* $slot_g(l, r)$

Implication Decomposition
$H, f : A \Rightarrow B, H' \vdash G$ *by* $apseq\,(f; slot_g; v.sl_g\,[^{ap(f;slot_a)}/_v])$ *new* v[8]
$H \vdash A$ *by* $slot_a$
$H, v : B, H' \vdash G$ *by* $slot_g(v)$

Or Decomposition
$H, y : A \vee B, H' \vdash G$ *by* $decide(y; l.slot_{left}(l); r.slot_{right}(r))$
$H, l : A, H' \vdash G$ *by* $slot_{left}(l)$
$H, r : B, H' \vdash G$ *by* $slot_{right}(r)$

Exists Decomposition
$H, x : \exists y.B(y), H' \vdash G$ *by* $spread(x; d, r.slot_g(d, r))$ *new* d, r
$H, d : D, r : B(d), H' \vdash G$ *by* $slot_g(d, r)$

All Decomposition
$H, f : \forall x.B(x), H' \vdash G$ *by* $apseq(f; d; v.slot_g\,[^{ap(f;d)}/_v])$
$H \vdash d \in D$ *by* $obj(d)$
$H, v : B(d), H' \vdash G$ *by* $slot_g(v)$[9]

False Decomposition
$H, f : False, H' \vdash G$ *by* $any(f)$

We opt for a very simple syntax. All terms have an outer operator which determines whether a term is *canonical* or *non-canonical*. Below are the operators that we will use: the canonical ones, and the corresponding non-canonical ones associated with the canonical. For instance, we use the term $pair(a;b)$ (or more succinctly $< a, b >$) for And construction rule, and for the corresponding decomposition rule we use $spread(x; l, r.t(l, r))$ where the binding variables l, r have a scope that is the subterm $t(l, r)$. The reason to use *spread* instead of the more familiar operators for decomposing a pair p such as $first(p)$ and $second(p)$ (or $p.1$ and $p.2$) is that we need to indicate how the subformulas of a conjunction will be named in the hypothesis list.[10] The canonical terms correspond to the values of the computation system, the data. The non-canonical expressions evaluate to canonical ones under the evaluation rules. They correspond in a sense to programs applied to data. This terminology is used by Martin-Löf in relating his type theory to programming languages [30].

Canonical	**Non-canonical**
pair	*spread*
inl, inr	*decide*
λ	$ap, apseq$
[ ] (list constructor)	*concat*
$\star$	

Terms having a canonical operator are called *values* and those with non-canonical operators are *operations*. In building terms we use bound variables, e.g. the notation for functions has the form $\lambda(x.t(x))$ where the subterm x is a *bound variable* with binding operator λ. If the subterm $t(x)$ has an occurrence of the letter x , the expression $t(x)$ is neither an operator nor a value, it s an *open* term. Open terms have a special status in the computation system because computation proceeds by substituting terms for variables within open terms. Typically we think of substituting values for variables, but there are reasons that we must also consider substituting variables for variables.

Computation is defined as a sequence of *rewritings* or *reductions* of terms to other terms according to very explicit rules. Each rule form of the proof systems $iFOL$ and iAL when all its slots are filled in becomes a term in an applied lambda calculus, and there are computation rules that define how to reduce these terms. These rules are given in detail in several papers about Computational Type Theory and Intuitionistic Type Theory, e.g. [12, 30]. The computation systems of $iFOL$ and iAL have the property that all reductions will terminate in values which are said to be the value of the expression. Canonical terms, such as $\lambda(x.x)$ reduce to themselves. A non-canonical term such as $ap(\lambda(x.x); y)$ reduces in one step to y.

It is important to notice that all of the programs and data of these logics

[10] See [13] for a more detailed description of the operators.

are untyped. We also say that they are *polymorphically* typed because we will see that many terms, such as $\lambda(x.x)$ will have the type $A \Rightarrow A$ for any proposition A of the logic.

3 The System *iAL*

Ancestral Logic (AL) is a well known extension of FOL, obtained by adding to it a transitive closure operator (see, e.g., [4, 10, 27]). Its expressive power exceeds that of FOL, since in AL one can give a categorical characterization of concepts such as the natural numbers and the concept of finiteness, which are not expressible in FOL (hence AL is not compact). In [4] it was argued that AL provides a suitable framework for the formalization of mathematics as it is appropriate for defining fundamental abstract formulations of transitive relations that occur commonly in basic mathematics. AL is also fundamental in computer science as reasoning effectively about programs clearly requires having some version of a transitive closure operator so that one can describe such notions as the set of nodes reachable from a program's variables. Since we are interested in extensions of intuitionistic first-order logic that clearly show the duality between logic and programming and which can capture general logical principles that have applicable computational content, it seemed natural to develop an intuitionistic version of AL – iAL, as a refinement of AL and an extension of $iFOL$.

3.1 The Transitive Closure Operator

A standard mathematical definition of the transitive closure of a binary relation R, denoted by R^+, is as follows. Let $\mathbb{N}$ be the set of natural numbers. For $n \in \mathbb{N}$ define: $R^{(0)} = R$, $R^{(n+1)} = R^{(n)} \circ R$, where the composition of relations R and S is defined by $(S \circ R)(x, y)$ iff $\exists z\,(S(x, z) \wedge R(z, y))$.

Definition 3. The *transitive closure* R^+ of binary relation R is defined by

$$R^+(x, y) = \exists n : \mathbb{N}.R^{(n)}(x, y)$$

At appropriate places we use the notation xRy instead of $R(x, y)$.

Note that we are using an intuitionistic semantics in our metatheory, so, for instance, the definition of composition means that we can effectively find the value z. Moreover, the constructive nature of the definition entails that xR^+y implies we know a natural number witness for the number of iterations of the relation R. Hence we can prove in the semantics that given elements x and y in D, xR^+y iff we can *effectively find* a finite list of elements $x_1, ...x_n$ from D, such that $xRx_1 \wedge x_1Rx_2 \wedge ... \wedge x_nRy$.

While this definition is perfectly acceptable, it depends essentially on the type of natural numbers, $\mathbb{N}$, with its attendant notion of equality and induction. Thus, it requires invoking a version of intuitionistic ω-logic (e.g.

[33]) as an underlying logic. In search of simplicity we wish to avoid this constraint, thus our axiomatic definition will be in terms of finite lists without mentioning the natural numbers explicitly. This will allow us to frame iAL in a more generic and polymorphic way, without explicit mention of $\mathbb{N}$.

Observe the following (equivalent) definition for the transitive closure.

Proposition 4. *R^+ is the minimal transitive relation L such that $R \subseteq L$, i.e.*

$$R^+ = \bigcap_{R \subseteq L \& Transitive(L)} L$$

where a relation R is said to be transitive if $\forall x, y, z.(xRy \wedge yRz) \Rightarrow xRz$.

This definition uses the *intersection type* of Constructive Type Theory (CTT) used in [3], the type $\bigcap_{x:A} B(x)$. Its elements are those that belong to all of the types $B(x)$. It generalizes the binary intersection $A \cap B$, consisting of the elements that belong to both types A and B. For instance $\{x : \mathbb{N} | Even(x)\} \cap \{x : \mathbb{N} | Prime(x)\}$ is the unit type $\{2\}$.

We shall use this definition to form our axiomatic system. This is a key step toward a polymorphic account of iAL which will support our claim that a type theoretic semantics can be not only *elementary*, but even *uniform*.

3.2 Realizability Semantics for iAL

Instead of defining evidence for transitive closure using $\mathbb{N}$, we use more generic and polymorphic constructs to give evidence for the transitive closure, in the spirit of using polymorphic functions, pairs, and tags. To know $R^+(x, y)$ for elements x and y in D, we construct a *list* of elements of D, say $[d, ..., d']$, and a list of evidence terms $[r, ..., r']$ such that r is evidence for $R(x, d)$ and r' is evidence for $R(d', y)$ and the intermediate terms form an *evidence chain*. That is, if d^+ is the list of elements and r^+ is the list of evidence terms, we have that the first element of d^+ is d_1 and first of r^+ is r_1 where $r_1 \in [R(x, d_1)]$, the next element of d^+ is d_2 and the next element of r^+ is r_2, evidence for $R(d_1, d_2)$, and so forth. These relationships hold because of the way the evidence is built up, so we do not need the numerical indices to define the relationship, only to describe it intuitively. It is crucial to notice that the concept of lists is subsumed into the realizers and does not appear in the logic itself.[11]

Notice that any well-formed formula (wff) together with a pair of distinct variables may be viewed as defining a binary relation. The notation $A_{x,y}$ will be used to specify that we treat the formula A as defining a binary relation with respect to variables x and y (x and y distinct variables), and other free variables that may occur in A are taken as parameters. Thus, one

[11] It should be noted that using lists instead of the naturals to form the transitive closure operator is a common method, already used in other logical frameworks, e.g., Isabelle [34].

may apply the transitive closure operator not only to atomic predicates, but to any wff. We write $A_{x,y}(u,v)$ for the formula obtained by substituting u for x and v for y in A. For simplicity of presentation, in what follows the subscript x,y is omitted where there is no chance of confusion.

Definition 5. ***(iAL formulas and their evidence)***
iAL formulas are defined as $iFOL$ formulas with the addition of the following clauses:

- If A is a formula, x,y distinct variables, and u,v variables, then $A^{+}_{x,y}(u,v)$ is a formula.
- The evidence type for $A^{+}_{x,y}(u,v)$ consists of lists of the form

$$[\langle d_0,d_1,r_1\rangle,\langle d_1,d_2,r_2\rangle,...,\langle d_n,d_{n+1},r_{n+1}\rangle]$$

where $n\geq 0$, $d_0=u$, $d_{n+1}=v$, $d_0,d_1,...,d_{n+1}:[D]_{\mathcal{M}}$, and $r_i\in[A_{x,y}(d_{i-1},d_i)]$ for $1\leq i\leq n+1$.

Notice that the realizers for transitive closure formulas are all polymorphic and thus independent of realizers for particular atomic formulas.

Recall that according to Def. 3, $R^{+}(x,y)$ iff $\exists n\left(N(n)\wedge R^{(n)}(x,y)\right)$. This is not a legal formula in our language, but this is intuitively what we mean, if we had the natural numbers at our disposal. The realizer for this "formula" will be of the form: $\langle n,\langle isnat(n),\langle x,d_1,...,d_n,y,\langle r_1,...,r_{n+1}\rangle\rangle\rangle\rangle$ where $isnat(n)$ realizes $N(n)$. The realizer of the transitive closure correlates nicely to this realizer. A realizer for the formula $R^{+}(x,y)$ of the form $\langle n,\langle isnat(n),\langle x,d_1,...,d_n,y,\langle r_1,...,r_{n+1}\rangle\rangle\rangle\rangle$ can be easily converted into the form $[\langle x,d_1,r_1\rangle,\langle d_1,d_2,r_2\rangle,...,\langle d_n,y,r_{n+1}\rangle]$ simply by rearranging the data. For the converse, the data can also be rearranged, but some additional data is required: n – which is the length of the list minus 1; and the realizer for it being a natural number – which is available as the length of a list is always a natural number.[12]

The underlying computation system is essentially the untyped programming language underlying the type theory and thus underlying *iAL*. The computation systems of *iAL* has the property that all reductions will terminate in values which are said to be the value of the expression. Moreover, all of the programs and data of *iAL* are untyped. We also say that they are *polymorphically* typed because many terms, such as $\lambda(x.x)$ will have the type $A\Rightarrow A$ for any proposition A of the logic.

[12]It is interesting to notice that by interpreting the naturals as lists on the unit type, the definition of the transitive closure operator by means of the natural numbers is an instance of our definition using lists.

3.3 Proof System for iAL over Domain D

We present a proof system for iAL which extends $iFOL$ [13] by adding construction and decomposition rules for the transitive closure operator. We here use the standard canonical operator [ ] for list constructor, and the non-canonical operator associated with it, *concat*, for concatenating two lists.

Definition 6. The proof system iAL is defined by adding to $iFOL$ the following rules for the transitive closure operator.

- **TC Base**

$H, x : D, y : D, H' \vdash A^+(x, y)\ by\ [\langle x, y, slot\rangle]$
$H, x : D, y : D, H' \vdash A(x, y)\ by\ slot$

- **TC Trans**

$H, x : D, y : D, H' \vdash A^+(x, y)\ by\ concat\,(slot_l, slot_r)$
$H, x : D, z : D, H' \vdash A^+(x, z)\ by\ slot_l$
$H, y : D, z : D, H' \vdash A^+(z, y)\ by\ slot_r$

- **TC Ind**

$H, x : D, y : D, r^+ : A^+(x, y), H' \vdash B(x, y)\ by\ tcind\,(r^+; u, v, w, b_1, b_2.tr(u, v, w, b_1, b_2);$
$u, v, r.st(u, v, r))$
$H, u : D, v : D, w : D, b_1 : B(u, v), b_2 : B(v, w), H' \vdash B(u, w)\ by\ tr(u, v, w, b_1, b_2)$
$H, u : D, v : D, r : A(u, v), H' \vdash B(u, v)\ by\ st(u, v, r)$
where u, v, w are fresh variables.

Rule TC Base states that the list consisting of the triple $[\langle x, y, r\rangle]$ where r realizes $A(x, y)$ is the realizer for the transitive closure $A^+(x, y)$. The crucial point about Rule TC Trans is that it does not nest lists of triples for the same goal; instead we "flatten the lists out" as proofs are constructed. This means that proofs of transitive closure have a distinguished realizer. Furthermore, it provides an adequate mechanism for creating a flat chain of evidence needed for the transitive closure induction rule. We have found this to be the minimal, most natural structure needed for handling a TC-chain.

The realizer for Rule TC Ind computes on the list r^+ and is recursively defined as follows:
$tcind(r^+; u, v, w, b_1, b_2.tr(u, v, w, b_1, b_2); u, v, r.st(u, v, r))$ computes to:
$if\ base(r^+)\ then\ st(r^+.1_1, r^+.1_2, r^+.1_3)$
$else$
$tr(r^+.1_1, r^+.1_2, r^+.2_2, tcind\,(rest(r^+); u, v, w, b_1, b_2.tr(u, v, w, b_1, b_2); u, v, r.st(u, v, r))$.
The operator $base(r^+)$ is true when r^+ is simply the singleton triple. We use the notation $r^+.u$ to denote the uth element in the list r^+, and the

subscript $r^+.u_i$ selects the ith elements of the triple ($i \in \{1,2,3\}$). The operator $rest(r^+)$ returns the list r^+ without its first element.

Note that the more commonly used induction rule (see [4, 27]) is derivable in our system.

Proposition 7. *The following rule is derivable in iAL:*
$H, x : D, y : D, r^+ : A^+(x,y), g : G(x), H' \vdash G(y)$
$H, u : D, v : D, r : A(u,v), g' : G(u), H' \vdash G(v)$
where u, v are fresh variables.

Proof. Immediately follows from TC Ind by taking $A(u,v)$ to be the formula $G(u) \Rightarrow G(v)$.

It is easy to verify the following Lemma. □

Lemma 8. *The following are provable in iAL:*

$$A(x,z), A^+(z,y) \vdash A^+(x,y) \tag{1}$$
$$A^+(x,z), A(z,y) \vdash A^+(x,y) \tag{2}$$

We next demonstrate that iAL is an adequate system for handling the transitive closure operator by showing that fundamental, intuitionistically valid statements concerning the TC operator are provable in iAL. Given a signature with a binary relation R, intuitively we may think of its interpretation as a directed graph whose vertices are the elements of the domain and two vertices are adjoined by an edge iff their interpretations are in the interpretation of the relation R. The transitive closure of R is then interpreted by the existence of a path between two vertices.

Observe the following basic statement: "if there is a path between x and y in a graph G, then either x and y are neighbors, *or* there is a neighbor z of x, such that from z there is a path to y". This statement is classically valid, and though at first sight one may doubt that it is intuitionistically valid (as it contains a disjunction), it is provable in iAL.

Proposition 9. *The following are provable in iAL:*

$$A^+(x,y) \vdash A(x,y) \vee \exists z \left(A(x,z) \wedge A^+(z,y)\right) \tag{3}$$
$$A^+(x,y) \vdash A(x,y) \vee \exists z \left(A^+(x,z) \wedge A(z,y)\right) \tag{4}$$

Proof. Denote by $\varphi(x,y)$ the formula $\exists z\,(A(x,z) \wedge A^+(z,y))$. For (3) apply TC Ind on the following two subgoals:

Goal 1: $A(u,v) \vdash A(u,v) \vee \varphi(u,v)$

Goal 2: $A(u,v) \vee \varphi(u,v), A(v,w) \vee \varphi(v,w) \vdash A(u,w) \vee \varphi(u,w)$

Goal 1 is clearly provable in $iFOL$. For Goal 2 it suffices to prove the following four subgoals, from which Goal 2 is derivable using Or Decomposition and Or Composition:

$$A(u,v), A(v,w) \vdash \varphi(u,w) \ , \ A(u,v), \varphi(v,w) \vdash \varphi(u,w)$$
$$\varphi(u,v), A(v,w) \vdash \varphi(u,w) \ , \ \varphi(u,v), \varphi(v,w) \vdash \varphi(u,w)$$

We prove $\varphi(u,v), \varphi(v,w) \vdash \varphi(u,w)$, the other proofs are similar. It is easy to prove (using Lemma 8) that $\exists z (A(v,z) \wedge A^+(z,w)) \vdash A^+(v,w)$. By TC Trans we can deduce $d : D, A(u,d), A^+(d,v), A^+(v,w) \vdash A^+(d,w)$, from which $\exists z (A(u,z) \wedge A^+(z,w))$ is easily derivable.

The proof of (4) is similar.

□

Another basic statement in graph theory is: "if there is a path between x and y in a graph G, then x and y are not isolated". Again, while it may seem to be intuitionistically invalid because of the existential nature of the argument, it turns out to be provable in iAL.

Proposition 10. *The following are provable in iAL:*

$$A^+(x,y) \vdash \exists z A(x,z) \quad (5)$$
$$A^+(x,y) \vdash \exists z A(z,y) \quad (6)$$

Proof. (5) is derivable applying TC Ind on the following two subgoals:

Goal 1: $A(x,y) \vdash \exists z A(x,z)$, which is easily provable in $iFOL$.

Goal 2: $\exists z A(u,z), \exists z A(v,z) \vdash \exists z A(u,z)$, which is valid due to Hypothesis.

The proof of (6) is symmetric.

□

The above proposition is based on the more general fact that the existential quantifier is definable by the transitive closure operator (see [4]).

Proposition 11. *The following is provable in iAL:*

$$\vdash \exists x A \leftrightarrow \left(A\left\{\frac{u}{x}\right\} \vee A\left\{\frac{v}{x}\right\}\right)^+_{u,v}(u,v)$$

where u and v are fresh variables.[13]

[13]The notation $A\left\{\frac{u}{x}\right\}$ denotes substituting u for x in A.

Proof. Denote by $\varphi(u,v)$ the formula $A(u,\overrightarrow{y}) \vee A(v,\overrightarrow{y})$. The right-to-left implication follows from Prop. 10 since $\exists z(A(u,\overrightarrow{y}) \vee A(z,\overrightarrow{y})) \vdash \exists x A(x,\overrightarrow{y})$ can be easily proven in $iFOL$, and Prop. 10 entails that $\varphi^+(u,v) \vdash \exists z \varphi(u,z)$. For the left-to-right implication it suffices to prove $d : D, A(d,\overrightarrow{y}) \vdash \varphi^+(u,v)$. Clearly, in $iFOL$, $d : D, A(d,\overrightarrow{y}) \vdash \varphi(d,v)$ is provable, from which we can deduce by TC Base $d : D, A(d,\overrightarrow{y}) \vdash \varphi^+(d,v)$. Since we also have $d : D, A(d,\overrightarrow{y}) \vdash \varphi(u,d)$, by Lemma 8 we obtain $d : D, A(d,\overrightarrow{y}) \vdash \varphi^+(u,v)$. □

It is important to notice that there is a strong connection between our choice for the realizer of the transitive closure and the standard realizers for $iFOL$. For example, Prop. 11 entails that the existential quantifier is definable using the transitive closure operator. It is interesting to see how the realizers for the defining formula correlate to the realizer for $\exists x P(x)$. The standard realizer for $\exists x P(x)$ is a pair $\langle d, \star \rangle$, since P is an atomic relation. The realizer for the defining formula, $(P(u) \vee P(v))^+(u,v)$, is of the form $[\langle d_0, d_1, r_1 \rangle, \langle d_1, d_2, r_2 \rangle, ..., \langle d_n, d_{n+1}, r_{n+1} \rangle]$ where $d_0 = u$, $d_{n+1} = v$, and each r_i is a realizer for $P(d_i) \vee P(d_{i+1})$. Now, suppose we have a realizer of the form $\langle d, \star \rangle$ of $\exists x P(x)$. The realizer for the defining formula in iAL will be $[\langle u, d, inr(\star) \rangle]$. For the converse, suppose we have a realizer of the form $[\langle d_0, d_1, r_1 \rangle, \langle d_1, d_2, r_2 \rangle, ..., \langle d_n, d_{n+1}, r_{n+1} \rangle]$ where $d_0 = u$, $d_{n+1} = v$. Then we can create a realizer for $\exists x P(x)$ in the following way: if r_1 is $inl(\star)$ return $\langle u, \star \rangle$, else return $\langle d_1, \star \rangle$.

Proposition 12. *The following is provable in iAL:*

$$\left(A^+\right)^+(x,y) \vdash A^+(x,y)$$

Proof. Applying TC Ind on $A^+(u,v), A^+(v,w) \vdash A^+(u,w)$ (which is derivable using TC Trans) and $A^+(x,y) \vdash A^+(x,y)$ (which is clearly provable) results in the desired proof. □

3.4 Soundness for *iAL*

We next prove that iAL is sound by showing that every provable formula is realizable, and even uniformly realizable. We do this by giving a semantics to sequents and then proceed by induction on the structure of the proofs. It is important to note that the realizers are all polymorphic, they do not contain any propositions or types as subcomponents and thus serve to provide evidence for any formulas built from any atomic propositions.

Given a type D (empty or not) as the domain of discourse, and given atomic propositional functions from D to propositions, $\mathbb{P}$, for the atomic propositions, and given the type theoretic meaning of the logical operators and the transitive closure operator, we can interpret an iAL sequent over dependent types by saying that a sequent $x_1 : T_1, x_2 : T_2(x_1), ..., x_n :$

$T_n(x_1, ..., x_{n-1}) \vdash G(x_1, ..., x_n)$ defines an effectively computable function from an n-tuple of elements of the dependent product of the types in the hypothesis list to the type of the goal, $G(x_1, ..., x_n)$.

Theorem 13. ***(Realizability Theorem for iAL)*** *Every provable formula of iAL is realizable in every model.*

Proof. The proof is carried out by induction on the structure of proofs in iAL. The proof rules for iAL show how to construct a realizer for the goal sequent given realizers for the subgoals. Also, the atomic (axiomatic) subgoals are of the form $x_1 : T_1, x_2 : T_2(x_1), ..., x_n : T_n(x_1, ..., x_{n-1}) \vdash T_j(x_1, ..., x_n)$, which are clearly realizable. □

Since propositions-as-types realizability is the definition of constructive truth, this theorem allows us to also say that every provable formula is true in every constructive model.

Theorem 14. ***(Soundness Theorem for iAL)*** *Every provable formula of iAL is intuitionistically valid.*

Corollary 15. ***(Consistency Theorem for iAL)*** *iAL is consistent, i.e. False is unprovable in iAL.*

4 Programming in *iAL*

4.1 Kleene Algebra

Kleene algebra (KA) [22] arises in many areas of computer science, such as automata theory, the design and analysis of algorithms, dynamic logic, and program semantics. There are many interesting models of KA, yet the theory of relational Kleene algebra (RKA) is of practical interest, particularly for programming language semantics and verification [24, 23].

Definition 16. A *Kleene algebra* (KA) is a structure $(K, +, \cdot, ^*, 0, 1)$, such that $(K, +, \cdot, 0, 1)$ forms an idempotent semiring which satisfies the following axioms:

$$(1)\ \ 1 + xx^* \leq x^* \qquad (2)\ \ 1 + x^*x \leq x^*$$
$$(3)\ \ xp \leq x \rightarrow xp^* \leq x \qquad (4)\ \ px \leq x \rightarrow p^*x \leq x$$

We omit $\cdot$, writing xy for $x \cdot y$. Elements of K, denoted by $p, q, r, x, ...$, are called *programs*. The upper semilattice structure induces a natural partial order on any idempotent semiring: $x \leq y \Leftrightarrow x + y = y$.

Definition 17. For an arbitrary set U, the set $P(U \times U)$ of all binary relations on U forms a Kleene algebra $R(U)$ with the interpretations $\cup$ for $+$, composition $\circ$ for $\cdot$, empty relation for 0, identity relation for 1 and reflexive transitive closure for *. A Kleene algebra is *relational* (RKA) if it is a subalgebra of $R(U)$ for some U.

Due to the prominence of relational models in programming language semantics and verification, it is of high interest to characterize them axiomatically or otherwise. We next show that iAL with equality ($iAL_=$) forms an adequate proof system for RKA, as RKA can be embedded in $iAL_=$ in such a way that any valid RKA expression is translated into a provable $iAL_=$ formula.

Definition 18. The system $iAL_=$, for languages with equality, is obtained from iAL by adding the following:

- **Reflexivity Axiom**
  $H \vdash x = x \, by \, Eq$
- **Paramodulation Rule**
  $H, x : D, y : D, r : A, H' \vdash A' \; by \; r$
  $H, x : D, y : D, H' \vdash x = y \; by \; slot$
  where A' is obtained from A by replacing free occurrences of x in A with y, with the standard restrictions.

Note that the axioms for symmetry and transitivity of the equality relation are derivable in $iAL_=$. Moreover, it can be proven easily that all of the atomic relations respect equality.

Let $\mathcal{L}$ be a pure first-order signature of *binary* predicates, $P_1, P_2, ...$ with equality. The translation from a RKA expression E into an $iAL_=$ formula, denoted by $|E|$, is defined inductively as follows. We use the notation $A_{rep(x,y)}$ to denote the formula obtained from A by replacing the free occurrences of x in A with y (applying the α-rule if necessary).

- For atomic p assign a distinct predicate P_i and define: $|p| := P_i\,(x, y)$
- $|0| := False$
- $|1| := x = y$
- $|E_1 + E_2| := |E_1| \vee |E_2|$
- $|E_1 \cdot E_2| := \exists z \left(|E_1|_{rep(y,z)} \wedge |E_2|_{rep(x,z)} \right)$ where z is a fresh variable.
- $|E^*| := x = y \vee |E|^+$
- $|E_1 = E_2| := (|E_1| \Rightarrow |E_2|) \wedge (|E_2| \Rightarrow |E_1|)$

Note that $|p \leq q|$ is the formula: $((|p| \vee |q|) \Rightarrow |q|) \wedge (|q| \Rightarrow (|p| \vee |q|))$, which is provably equivalent to $|p| \Rightarrow |q|$. For convenience, in what follows we shall use this as the translation of $p \leq q$.

Theorem 19. *For E a valid expression of RKA, $|E|$ is provable in $iAL_=$.*

Proof. The only rule of RKA is the transitivity of equality which translates to the transitivity of $\Leftrightarrow$, which is clearly provable in $iFOL$. It is easy to see that the translation of all the axioms for the idempotent semiring are provable using the proof rules of $iFOL$. It remains to show that the translation of axioms (1) – (4) in Def. 16 are provable in $iAL_{=}$.

(1): The translation is: $[x = y \vee \exists z\,(xPz \wedge (z = y \vee zP^{+}y))] \Rightarrow (x = y \vee xP^{+}y)$. If $x = y$, by inl we get a proof of $x = y \vee (xP^{+}y)$. Otherwise, assume $\exists z\,(xPz \wedge (z = y \vee zP^{+}y))$. If $z = y$ then by the Paramodulation rule we get xPy, from which, by TC Base, we can get $xP^{+}y$. If $zP^{+}y$ then by Lemma 8 we get $xP^{+}y$. Applying inr results in the desired proof.

(2): Symmetric to the proof of (1).

(3): We need to show that the following rule is derivable in $iAL_{=}$:
$H, x : D, y : D, H' \vdash \exists z\left(xP_1z \wedge \left(z = y \vee zP_2^{+}y\right)\right) \Rightarrow xP_1y$
$H, x : D, y : D, H' \vdash \exists z\,(xP_1z \wedge zP_2y) \Rightarrow xP_1y$
Assume $\exists z\,(xP_1z \wedge zP_2y) \Rightarrow xP_1y$ and $\exists z\left(xP_1z \wedge \left(z = y \vee zP_2^{+}y\right)\right)$. If $z = y$ and xP_1z then xP_1y by the Paramodulation rule. Otherwise, assume $\exists z\left(xP_1z \wedge zP_2^{+}y\right)$. From $\exists z\,(xP_1z \wedge zP_2y) \Rightarrow xP_1y$ we can derive $xP_1z, zP_2y \vdash xP_1y$, from which, using the rule from Prop. 7, we can obtain $xP_1z, zP_2^{+}y \vdash xP_1y$. This entails that $\exists z\left(xP_1z \wedge zP_2^{+}y\right) \vdash xP_1y$. In both cases xP_1y is derivable from the assumptions.

(4): Symmetric to the proof of (3) .

□

4.2 Kleene Algebra with Tests

Many of the applications of KA are enhanced using Kleene algebra with tests (KAT) [23], which is an equational system for program verification that combines KA with Boolean algebra. The presence of tests allows KAT to model basic programming language constructs such as conditionals, while loops, verification conditions, and partial correctness assertions.

Definition 20. A *Kleene algebra with tests* (KAT) is a KA with an embedded Boolean subalgebra, i.e., a two-sorted structure $(K, B, +, \cdot, ^{*}, ^{-}, 0, 1)$ such that:

1. $(K, +, \cdot, ^{*}, 0, 1)$ is a KA,
2. $(B, +, \cdot, ^{-}, 0, 1)$ is a Boolean algebra,
3. $B \subseteq K$.

Elements of B, denoted by $b, c, ...$, are called *tests,* and the Boolean complementation operator $^-$ is defined only on them.
Relational Kleene algebra with test ($RKAT$) is RKA with test, where tests are simply subsets of the identity relation on the domain U. The Boolean complementation operator on tests gives the set-theoretic complement in the identity relation.

Let $\mathcal{L}$ be a first-order signature of *binary* predicates, $P_1, P_2, .., B_1, B_2, ...$ with equality. We expand the translation from a RKA expression into an $iAL_=$ formula, to a translation of a $RKAT$ expression into an $iAL_=$ formula by adding the following clause:

- For each atomic test b assign a distinct predicate symbol B_i and define:

$$|b| := B_i(x,y) \wedge x = y$$
$$\left|\bar{b}\right| := \neg B_i(x,y) \wedge x = y$$

Definition 21. The system $iALT_=$ is obtained by adding to $iAL_=$ axioms of the form $B_i(x,y) \vee \neg B_i(x,y)$ for each predicate B_i.

The translation of each test may be viewed in $iALT_=$ as a decidable unary predicate. This is since using the Paramodulation rule from any formula of the form $B_i(x,y) \wedge x = y$ one can deduce $B_i(x,x)$.[14]

Theorem 22. *For E a valid expression of $RKAT$, $|E|$ is $iALT_=$ provable.*

Proof. Clearly all the translations of the axioms of RKA remain provable in $iALT_=$. It remains to show that the translated axioms for the boolean algebra are provable in $iALT_=$. The translated axioms of associativity, commutativity and distributivity of $+$ and $\cdot$ are provable as in the case of RKA. We next show that the translations of the remaining axioms are provable.

- $b \cdot \bar{b} = 0$: The translation is $\exists z\,(B(x,z) \wedge x = z \wedge \neg B(z,y) \wedge z = y) \Leftrightarrow False$. The right to left implication is clearly provable. For the converse, $\exists z\,(B(x,z) \wedge x = z \wedge \neg B(z,y) \wedge z = y)$ easily entails $B(x,x) \wedge \neg B(x,x)$, from which $False$ is provable, since $\neg B(x,x)$ is an abbreviation of $B(x,x) \Rightarrow False$.

- $b + \bar{b} = 1$: The translation is $(B(x,y) \wedge x = y) \vee (\neg B(x,y) \wedge x = y) \Leftrightarrow x = y$, which is provably equivalent to $(x = y \wedge (B(x,y) \vee \neg B(x,y))) \Leftrightarrow x = y$. Thus, the left to right implication is clearly provable. As $B(x,y) \vee \neg B(x,y)$ is an axiom of the system $iALT_=$, the right to left implication is also provable.

[14] Tests in constructive systems are the properties which are decidable, thus the system includes instances of the Law of Excluded Middle for each test (and only for them). This correlates to the fact that in Kleene algebra with tests the Boolean complementation operator is defined only on tests.

- $b + (b \cdot c) = b$: The translation of the axiom is the formula: $[(B_1(x,y) \wedge x = y) \vee \exists z.B_1(x,z) \wedge x = z \wedge B_2(z,y) \wedge z = y] \Leftrightarrow B_1(x,y) \wedge x = y$. The right to left implication is immediate. For the other direction, $\exists z\,(B_1(x,z) \wedge x = z \wedge B_2(z,y) \wedge z = y)$ entails $B_1(x,y) \wedge B_2(x,y)$ and $\exists z\,(x = z \wedge z = y)$, from which $B_1(x,y)$ and $x = y$ are provable.

- $b \cdot (b + c) = b$: The translation of the axiom is the formula: $\exists z\,(B_1(x,z) \wedge x = z \wedge ((B_1(z,y) \wedge z = y) \vee (B_2(z,y) \wedge z = y))) \Leftrightarrow (B_1(x,y) \wedge x = y)$. The left to right implication is similar to the proof of the last axiom. For the converse direction, observe that from $B_1(x,y) \wedge x = y$ the formula $\exists z\,(B_1(x,z) \wedge x = z \wedge B_1(z,y) \wedge z = y)$ is derivable, and from it the left hand side is derivable.

□

$RKAT$ is especially interesting because it closely models our intuition about programs. For instance, the **if** and **while** program constructs are encoded in $RKAT$ as in propositional Dynamic Logic:

$$\mathbf{if}\, b\, \mathbf{then}\, p\, \mathbf{else}\, q := bp + \bar{b}q$$
$$\mathbf{while}\, b\, \mathbf{do}\, p := (bp)^* \bar{b}$$

By the above translation, the construct $\mathbf{if}\, b\, \mathbf{then}\, p\, \mathbf{else}\, q$ can be expressed by a formula equivalent to $(B(x,x) \wedge P_1(x,y)) \vee (\neg B(x,x) \wedge P_2(x,y))$, and the construct $\mathbf{while}\, b\, \mathbf{do}\, p$ is expressible in $iALT_=$ by a formula equivalent to $\left((B(x,x) \wedge P(x,y))^+ \vee x = y\right) \wedge \neg B(y,y)$.

Another connection to programming derives from the relation between $iALT_=$ and the theory of flowchart schemes (e.g., [28]). A central question in the theory of flowchart schemes is scheme equivalence. In [28] Manna presents examples of equivalence proofs done by transformations on the graphs of the schemes. In [23] KAT was used to recast much of the theory of flowchart schemes into an algebraic framework by assigning to each flowchart scheme a KAT expression. Thus, the question of scheme equivalence was replaced by the question of equality between KAT expressions. The translation algorithm given in this section shows that the problem of scheme equivalence amounts to the question of equivalence in $iALT_=$ between two formulas.

There are a number of benefits to reasoning about programs in $iALT_=$ as oppose to $RKAT$. First, while $RKAT$ can be embedded into $iALT_=$, the language of $iALT_=$ is far richer than that of $RKAT$. Hence, there are many meaningful statements about programs that cannot be formulated in $RKAT$ but can be captured in $iALT_=$(e.g. "there is a state to which each run gets to (on any input)") Another key feature of iAL is that proofs of specifications in iAL carry their computational content in the realizer.

Thus, proving an iAL formula results in a realizer which can be thought of as holding the computational element of a program. Even simple assertions, such as $A^+(x,y) \Rightarrow \exists z A(z,y)$, have interesting realizers that depend on the tcind realizer, and thus correspond to recursive programs. Moreover, since $iALT_=$ is an effective, constructive proof system, it is more amenable to implementation.

5 Further Research

We argue that iAL is a natural "next step" one needs to take, starting from $iFOL$, in order to capture many applications in computer science and mathematics. In this work we have demonstrated a few of them, yet further work is still needed to provide more evidence for this claim and to explore the natural scope of iAL. One application of iAL in computer science might be related to distributed protocols. One can develop efficient deployed algorithms from the natural constructive proofs of theorems about data structures expressible in iAL, and explore specific direct use of such proofs, e.g. in building distributed protocols and making them attack-tolerant. We further plan to explore the connections between iAL to new useful variations of KAT ($KAT + B!$ [17], Kleene Algebra with Equations [25], Kleene algebras in software defined networks [14]), mostly motivated by concrete applications for verification.

Exploring ways of strengthening iAL in a natural way are also needed. One standard way is to develop a method for handling constants and function symbols in a polymorphic way. Further work is required in order to explore some proof theoretical properties of the system iAL. For instance, we conjecture that the proof system for iAL satisfy some appropriate form of the subformula property, but it is clear that the usual definition of this notion should be revised.[15] Thus the induction rule of the system satisfies the subformula property only if we take a formula to be a subformula of every substitution instance of it. We also plan to determine and explore fragments of the system iAL that are more convenient to work with (e.g. they admit full cut-elimination), but are still sufficient for at least some concrete applications. An example of such a fragment may be the one which corresponds to the use of the *deterministic* transitive closure operator (see, e.g., [21]).

[15] Exactly as the straightforward notion of subformula used in propositional languages is changed on the first-order level, where for example a formula of the form $\psi\{\frac{t}{x}\}$ is considered to be a subformula of $\forall x\psi$, even though it might be much longer than the latter.

Acknowledgment

This research was partially supported by the Ministry of Science, Technology and Space, Israel, and the Cornell University PRL Group.

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In Angus Macintyre, Leszek Pacholski, and Jeff Paris, editors, *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. Elsevier, 1978.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier, 1986.

[3] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[4] Arnon Avron. Transitive closure and the mechanization of mathematics. In *Thirty Five Years of Automating Mathematics*, pages 149–171. Springer, 2003.

[5] Bruno Barras. Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.

[6] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Program. Lang. Syst.*, 7(1):113–136, 1985.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[8] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[9] Adam Chlipala. A Verified Compiler for an Impure Functional Language. In *POPL*, pages 93–106, 2010.

[10] Liron Cohen and Arnon Avron. Ancestral logic: A proof theoretical study. In Ulrich Kohlenbach et al., editor, *Logic, Language, Information, and Computation*, volume 8652 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2014.

[11] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. *North-Holland Mathematics Studies*, 102:21–37, 1985.

[12] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, James F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[13] Robert L. Constable and Mark Bickford. Intuitionistic Completeness of First-Order Logic. *Annals of Pure and Applied Logic*, 165(1):164–198, 2014.

[14] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–355. ACM, 2015.

[15] Harvey Friedman. The consistency of classical set theory relative to a set theory with intuitionistic logic. *The Journal of Symbolic Logic*, 38(2):pp. 315–319, 1973.

[16] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78. Springer, 1979.

[17] Niels B. B. Grathwohl, Dexter Kozen, and Konstantinos Mamouras. KAT+ B! In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 44. ACM, 2014.

[18] Joseph Y. Halpern, Robert W. Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(02):213–236, 2001.

[19] Robert W. Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[20] Jason J. Hickey. *The MetaPRL logical programming environment*. PhD thesis, Cornell University, 2001.

[21] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[22] Stephen C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*. Memorandum (Rand Corporation). Rand Corporation, 1951.

[23] Dexter Kozen and Allegra Angus. Kleene algebra with tests and program schematology. Technical report, Cornell University, 2001.

[24] Dexter Kozen and Adam Barth. Equational verification of cache blocking in lu decomposition using kleene algebra with tests. Technical report, Cornell University, 2002.

[25] Dexter Kozen and Konstantinos Mamouras. Kleene algebra with equations. In *Automata, Languages, and Programming*, pages 280–292. Springer, 2014.

[26] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. volume 41, pages 42–54. ACM, 2006.

[27] Tal Lev-Ami, Neil Immerman, Tom Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Automated Deduction–CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2005.

[28] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, Inc., 1974.

[29] Zohar Manna and Richard Waldinger. *The logical basis for computer programming*, volume 1. Addison-Wesley Reading, 1985.

[30] Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104:153–175, 1982.

[31] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory.* Studies in proof theory. Bibliopolis, 1984.

[32] John McCarthy. A formal description of a subset of algol. Technical report, DTIC Document, 1964.

[33] James D. Monk. *Mathematical Logic.* Number 1-243 in Graduate Texts in Mathematics. Springer, 1976.

[34] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[35] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction.* International series of monographs on computer science. Clarendon Press, 1990.

[36] Vincent Rahli, Nicolas Schiper, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. A diversified and correct-by-construction broadcast service. In *The 2nd International Workshop on Rigorous Protocol Engineering*, 2012.

[37] Morten H. Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomoprhism*. Elsevier, 2006.

[38] Anne S. Troelstra and Dirk Van Dalen. *Constructivism in Mathematics: An Introduction*. Number 1 in Constructivism in Mathematics. North-Holland, 1988.