In: Proceedings of the 5th Workshop on Inference in Computational Semantics (ICoS-5),
April 2006, Buxton, UK. (6 pages).

The ALLIGATOR Theorem Prover for Dependent Type Systems: Description and Proof Sample

Paul Piwek

Centre for Research in Computing
The Open University, Milton Keynes, UK
p.piwek@open.ac.uk

Abstract

This paper introduces the ALLIGATOR theorem prover for Dependent Type Systems (DTS). We start with highlighting a number of properties of DTS that make them specifically suited for computational semantics. We then briefly introduce DTS and our implementation. The paper concludes with an example of a DTS proof that illustrates the suitability of DTS for modelling anaphora resolution.

1 Introduction

Automated *symbolic* inference requires a formal language as the substratum for reasoning. Blackburn and Bos ([7]) make a good case for the use of First Order Predicate Logic (FOPL) in computational semantics, citing both practical (availability of high performance theorem provers and to a lesser extent model builders) and theoretical reasons (they discuss a range of interesting phenomena which can be dealt with in FOPL).

We agree with the idea that FOPL is a good starting point, but also think that for computational semantics to develop further as a field, extensions going beyond FOPL should be actively explored. In this paper, a research tool is described that takes such explorations in one particular direction. The tool – ALLIGATOR – is a theorem prover for Dependent Type Systems (DTS) [4,5]. The Sicstus Prolog source code of this prover is available, free of charge, for research purposes ([18]). DTS are an attractive option for computational semantics for a number of reasons:

(i) DYNAMIC POTENTIAL (cf. [15]): The notion of a *context* that is built up *incrementally* is inherent to DTS.

(ii) FLEXIBILITY: By varying a limited number of parameters, it is possible to switch from, for example, propositional to predicate logic, or first order to higher order logics. Additionally, although the basic underlying logic is constructive, DTS allows for the flexible use of axioms to regain full classical

In: Proceedings of the 5th Workshop on Inference in Computational Semantics (ICoS-5),
April 2006, Buxton, UK. (6 pages).

logic, or more fine-grained alternatives. For example, it is possible to specify for individual predicates whether they are bivalent.

(iii) EXTENSIBILITY: A DTS-context includes what is known as the signature in FOPL. Consequently, the signature can be extended incrementally, making it possible to model the acquisition of new concepts by language users.

(iv) PROOF-OBJECTS: In DTS, Gentzen-style natural deduction proofs are first-class citizens. This gives us the following advantages: (a) *Reliability*: It allows us to heed the *de Bruijn criterion* for reliable proof systems: "A proof assistant satisfies the de Bruijn criterion if it generates 'proof-objects' (of some form) that can be checked by an easy algorithm." (cited from [5]) (b) *Naturalness*: DTS proofs correspond with natural deduction proofs. This is of interest if one is concerned with models of human reasoning in natural language understanding. In psychology, some schools of thought argue that natural deduction is a good approximation of human reasoning (see, e.g., [21]). (c) *Relevance*: Proof objects can help to identify proofs which are valid but spurious in the sense that they do not really consume their premises (see [14]). (d) *Justification of behaviour*: Explicit proof objects provide direct access to the justifications that an agent has for the conclusions and the interpretations that it constructs. This is particularly useful for dialogue agents that need to respond to utterances of other agents. Such responses can themselves again be queried, for example, through clarificatory questions (*cf.* [22]) and why questions (A:p, B: no, $\neg p$, A: Why $\neg p$?). In order to respond appropriately, the agent needs to access its own background knowledge and how it was used to draw conclusions. DTS proof objects provide a compact representation of this information.

(v) APPLICATIONS: DTS-style analyses exist for a wide range of linguistic phenomena including donkey sentences ([23]), anaphoric expressions and temporal reference ([20]), belief revision ([8]), bridging anaphora ([19]), clarification ellipsis ([10]), metonymy ([9]), inter-agent communication, knowledge and observation ([1]), ontological reasoning for feedback dialogues ([6]), and human-machine dialogue ([2]). Additionally, there is research on relating DTS proof-theoretic natural language semantics to model-theoretic approaches ([12]), and there are studies employing the related formalism of labelled deduction to natural language semantics ([16]). In 2005, the 2^{nd} Workshop on Lambda-Calculus, Type Theory, and Natural Language took place at King's College London ([11]).

We concede that none of the properties we have listed is on its own unique to DTS. However, to the best of our knowledge, no extant logical calculus combines all these properties in a *single system* with well-understood meta-mathematical properties (DTS play a central role in theoretical computer science, see [4]).

2 Dependent Type Systems

DTS come in a wide variety of flavours and variations. All these systems share, however, two features: a *typing system* and a notion of *dependency*. Firstly, DTS are

In: Proceedings of the 5th Workshop on Inference in Computational Semantics (ICoS-5),
April 2006, Buxton, UK. (6 pages).

type systems. That is, given a set of assumptions Γ, also known as the *context*, they provide rules for determining whether a particular object, say a, belongs to a given type, say t. We write $\Gamma \vdash a : t$, if, given the context Γ, a is of type t, i.e., a *inhabits* type t. The objects that are classified using type systems are (normalizing) terms of the λ-calculus. Γ is a sequence of statements $x_1 : t_1, \ldots, x_n : t_n$ (with $n \geq 0$).

Dependency is the second feature of DTS, and it comes in two forms. First, there is dependency between statements in the context: in order to use a type t_k to classify an object x_k, this type t_k needs to have been introduced in that part of the context that *precedes* it or t_k has to be a sort. In other words, t_k can only be used if (1) it itself inhabits a type or can be constructed from other types that are available in the context preceding it, or (2) it belongs to a fixed and usually small set of designated types that are called *sorts*. Because sorts need no preceding context, they make it possible to keep contexts finite.

Second, there is a variety of dependency that occurs *inside* types. Since type systems are used to classify terms of the λ-calculus, they can also deal with functions. A function f from objects of type t_1 to objects of type t_2 inhabits the function type $t_1 \rightarrow t_2$. *Dependent* function types are a generalization of function types: a dependent function type is a function type where the range of the function changes depending on the object to which the function is applied. The notation for dependent function types is $\Pi x : A.B$ (we also use our own alternative 'arrow notation': $[x : A] \Rightarrow B$). If we apply an inhabitant of this function type, say f, to an object of type A, then the resulting object fa (f applied to a) is of type B, but with all free occurrences of x in B substituted with a (that is, the type of fa is $B[x := a]$).

One way to make the leap from type systems to logic is as follows. From a logical point of view, we are interested in propositions as the constituents of deductive arguments. In classical logic, one focuses on judgements of the following form: the truth of proposition q follows/can be derived from the truth of the propositions $p_1, \ldots, p_n$. We reason from the truth of the premises to the truth of the conclusion. To do logic in a DTS, we move from *truth* to *proof*: we, now, reason from the proofs that we (assume to) have for the premises to a proof for the conclusion. In other words, we are interested in judgements of the following form: a is proof of proposition q follows/can be derived assuming that a_1 is a proof of p_1, a_2 is a proof of p_2, ..., and a_n is a proof p_n. Such a judgement can be formalized in a DTS as $a_1 : p_1, \ldots, a_n : p_n \vdash a : p$. Thus, we read $a : p$ as 'a is a proof for p'. Thus, we model proofs as (λ-calculus) terms and propositions as (a certain class of) types in DTS. This is known as the Curry-Howard-de Bruijn embedding.

The embedding is grounded in the Brouwer-Heyting-Kolmogorov interpretation of proofs as *constructions*; e.g., a proof for a conditional $p \rightarrow q$ is identified with a method that transforms a proof of p into a proof for q. In a DTS, this is formalized by modelling the proof f for a type $p \rightarrow q$ as a function from objects of type p to objects of type q, such that if a is a proof of p, then f applied to a is a proof of q (i.e., $fa : q$). Universal quantification is dealt with along the same lines. In a DTS, the counterpart for universal quantification is the dependent function type. In particular, $\forall x \in A : P(x)$ becomes $(\Pi x : A.Px)$. A proof for this type is a

In: Proceedings of the 5th Workshop on Inference in Computational Semantics (ICoS-5),
April 2006, Buxton, UK. (6 pages).

function f which, given any object $a : A$, returns the proof fa for Pa.

PURE TYPE SYSTEMS (PTS; [4]) are of particular interest, because of their generality. With a small number of parameters, PTS can be tailored to match a wide variety of DTS. ALLIGATOR implements an extension of PTS with Σ types. Σ types are also known as dependent product types and can be used to model $\wedge$ and $\exists$.

3 System Architecture, Implementation and Proof Sample

There is no room for a detailed description of the system here, for that we refer to the documentation and code available at [18]. What we can offer is, firstly, a list of differences between ALLIGATOR and other DTS provers: (a) ALLIGATOR directly constructs proof objects for natural deduction proofs. Other provers for DTS typically work with internal representations that are only at the end of the reasoning process *translated* to natural deduction proof objects. For example, COCKTAIL ([13]) uses tableaux and translates these, whereas TPS ([3]) is based on the mating method. The handbook chapter by Barendregt and Geuvers on proof assistants for DTS ([5]) lists a number of further automated theorem provers, none of which works directly with proof objects. (b) ALLIGATOR was not developed with mathematical/program specification reasoning in mind, but rather for inferences in language interpretation. As a consequence, it has been streamlined to link up with notation and functionality relevant to computational semantics (specifically, allowing for notation which is close to [15] and omission of inductive types). (3) To the best of our knowledge, ALLIGATOR is the only automated theorem prover which directly conforms to the specification of Pure Type Systems ([4]), the most general and flexible kind of DTS (most DTS can be emulated in PTS; see [17] for an overview of DTS and their counterparts in PTS).

ALLIGATOR 1.0 has been implemented in Sicstus PROLOG and been tested with version 3.12.2 of Sicstus. An overview of the architecture is presented in Figure 1.a. Note that the system applies both forward and backward inferencing. Most of the forward inferencing takes place before backward inferencing (though some backward inferencing rules do also have forward inferencing component). Reduction of terms is also carried out mainly before backward inferencing. Inferencing is done with a flattened representations of DTS terms (the arrow notation). Proofs are checked at the end of the inferencing process for their correctness (the code for proof checking is separate from the theorem proving code).

Currently, ALLIGATOR has the status of an experimental research tool. It is intended for testing computational solutions to theoretically challenging problems in computational semantics. Scalability has, so far, not been given much attention, though it will obviously need to be addressed if the system is to be used in large-scale practical applications. Currently, the system is merely intended as a *baseline* and *starting point* for implementing efficient and effective proof search heuristics. We now conclude with an example of the use of ALLIGATOR.

The discourse ‘The barn contains a chain saw or a power drill. It . . .’ (p. 205 of [15]) poses a problem for the structural approach to anaphora resolution proposed in