Foundational Aspects of Syntax

Dale Miller and Catuscia Palamidessi
Department of Computer Science and Engineering
The Pennsylvania State University
220 Pond Laboratory
University Park, PA 16802-6106 USA

Phone: (814) 865-9505, FAX: (814) 865-3176
dale@cse.psu.edu, catuscia@cse.psu.edu
http://www.cse.psu.edu/~dale
http://www.cse.psu.edu/~catuscia

Introduction

A large variety of computing systems, such as compilers, interpreters, static analyzers, and theorem provers, need to manipulate syntactic objects like programs, types, formulas, and proofs. Choosing an appropriate computation representation for these syntactic objects is critical in building both formal theories and computational systems. We first examine the conventional approach to the representation of syntax and then argue that a richer abstract representation needs to be selected.

Expressions to be read by humans are often represented by strings. Strings, however, contain whitespaces, brackets, and syntactic sugar that aid in human readability but are not related to the intended semantics. On the other hand, important semantic information is not represented directly. For this reason, expressions are generally transformed prior to being manipulated into *parse trees*. For example, the first-order formula represented by the string

$$\text{“}\forall x \exists y (p(x, y) \supset \exists x q(f(x), a))\text{”}$$

would be parsed into a tree

$$(\forall\ x\ (\exists\ y\ (\supset\ \ (p\ x\ y)\ (\exists\ x\ (q\ (f\ x)\ a))))).$$

With such a representation of syntax, semantically important notions, like the function-argument relationship, are immediate, whereas in the string representation that information must be extracted carefully by counting parentheses and accounting for infix declarations.

As this example also illustrates, however, there are important aspects of the intended meaning that are not captured by the parse tree syntax, such as the concept of bound variable and the related notions of scope, α-conversion, and substitution. In the example, variables are represented by constructors of type *variable* (a subcategory of terms), and the various aspects of binding are derived

notions that need to be carefully defined and implemented. Fortunately, there have been a large number of advances in the theory of syntax, particularly resulting from proof theory of intuitionistic logic, higher-type quantification, and dependent λ-calculus, that suggest an alternative approach [MN87, PE88], which we call here the λ-tree syntax.

The λ-tree syntax

This approach enriches the parse tree representation in the following two ways.

(1) Bindings are denoted by λ-abstractions and the abstracted expressions are given new syntactic categories formed using the type arrow constructor. For example, $\lambda x.p(a,x) \supset q(f(x),a)$ represents an abstraction of a term over a formula and this expression has the type $\mathit{term} \to \mathit{formula}$. As Church argued in [Chu40], most binding operators can be encoded by using λ-abstraction. For instance, both the universal and existential quantifier in the example above would be modified to take one argument of type $\mathit{term} \to \mathit{formula}$ instead of the two arguments of type *variable* and *formula*. In this way, the category *variable* is not longer necessary (being subsumed by the corresponding notion in the λ-calculus) and the expression of type *formula* above would be represented as

$$(\forall\ \lambda x\ (\exists\ \lambda y\ (\supset\ \ (p\ x\ y)\ (\exists\ \lambda x\ (q\ (f\ x)\ a))))).$$

(2) α-conversion is part of this representation in the sense that two parse trees which differ only for the names of bound variables are identified as λ-trees. As a consequence, the names of bound variables are not accessible (just as memory locations are not available in high-level languages) and operations that, on parse trees, would require dealing with the many technical aspects of variable names, are treated by using higher-level mechanisms described below. Furthermore, the η-rule is also assumed since it is most natural in this simply typed setting to identify an expression t with $\lambda x.tx$ if x is not free in t.

For another example of using λ-tree syntax, consider the untyped λ-calculus. Let tm denote the syntactic category for the untyped λ-terms. Let application be represented by the constructor *app* of type $tm \to tm \to tm$ and let abstraction be represented by the constructor *abs* of type $(tm \to tm) \to tm$. It is easy to show that α-equivalence classes of closed, untyped λ-terms (the object-level terms) correspond to $\alpha\beta\eta$-equivalence classes of closed terms of type tm (the meta-level terms).

Computing on λ-trees

We list two central issues that arise when computing with λ-trees.

Determining the structure of λ-trees. One approach would be to use matching (or unification) modulo $\alpha\eta$-conversion. However, α and η are not enough. For example, consider the problem of recognizing exactly those expressions that represent universally quantified implications. One might consider the pattern $(\forall\lambda u(\supset\ P\ Q))$, where P and Q are the meta-variables to be instantiated by a successful match, but because substitution is not allowed (logically) to capture free variables, all instances of this pattern would be expressions in which the universally quantified variable is vacuous. The pattern $(\forall\lambda u(\supset\ (P\ u)\ (Q\ u)))$ however will work, if we use β-conversion to simplify the instantiate pattern (β-conversion is the rule that states that the expression $(\lambda x.t)s$ is equal to t with s substituted for x). For example, this pattern will match with the λ-tree $(\forall\lambda x(\supset\ (p\ x\ x)\ (q\ a\ a)))$ by instantiating P with $\lambda x(p\ x\ x)$ and Q with $\lambda x(q\ a\ a)$ and then using β-conversion.

Matching and unification of simply typed λ-terms modulo $\alpha\beta\eta$-conversion are complex operations. For example, matching at second order is NP-complete and at higher-types is undecidable. If we examine more closely the example above, however, we find that we do not need full β-conversion: we only need the weaker β_0-conversion rule that states that the expression $(\lambda x.t)y$ is equal to t with x replaced by y, if y is not free in $\lambda x.t$. If meta-variables are applied only to distinct bound variables, then β_0 is complete with respect to β, matching and unification are decidable and unitary [Mil91], and can actually be solved in linear time [Qia93].

Recursion over λ-trees. A second major aspect of computing with λ-trees is how to define recursion over them: namely, how does one compute on subexpressions of a λ-abstraction to determine a value for the full expression. In particular, how can one "descend" into the λ-abstraction $\lambda x.t$ in a way that is independent from the choice of the name x. One successful solution to this problem is to use the *generic* and *hypothetical* judgments that are found in intuitionistic logic and associated dependent typed λ-calculi. In such settings, computations are specified with relations (atomic judgments) and generic and hypothetical judgments employ universal quantification and implication, respectively.

Consider, for example, the judgment that an untyped λ-term has a certain simple type. We first introduce the category ty to denote the syntactic domain of simple types; provide constructors i of type ty and *arrow* of type $ty \to ty \to ty$; and introduce the atomic judgment (predicate) *typeof* that asserts that its first argument (a term of type tm) has its second argument (a term of type ty) as a simple type. The following two inference rules completely specify the *typeof* judgment.

$$\frac{typeof\ M\ (arrow\ A\ B) \quad typeof\ N\ A}{typeof\ (app\ M\ N)\ B} \qquad \frac{\forall x(typeof\ x\ A \supset typeof\ (R\ x)\ B)}{typeof\ (abs\ R)\ (arrow\ A\ B)}$$

The first of these inference rules is essentially a simple Horn clause while the

second has both a universal quantifier (for the generic judgment) and an implication (for the hypothetical judgment). Inference rules such as the second one are easily expressible in hereditary Harrop formulas [MNPS91], the logical foundations of Isabelle [Pau90] and λProlog [NM88], and in the dependent typed λ-calculus [HHP93], which has been mechanized by Pfenning in Elf [Pfe89].

The conventional approach to specifying such a typing judgment would involve an explicit context of typing assumptions and an explicit treatment of bound variables names, either as strings or deBruijn numbers. The hypothetical judgment (the meta-level implication) implicitly addresses the typing context, and the generic judgment (the universal quantifier) implicitly addresses the bound variable names by via the use of eigenvariables in a proof using these inference rules.

An additional example

We show here how to use λ-trees to encode the π-calculus [MPW92]. We need two syntactic categories, *name* for channels and *proc* for processes. The output prefix is the constructor *out* of type $name \to name \to proc \to proc$ and the input prefix is the constructor *in* of type $name \to (name \to proc) \to proc$: the π-calculus expressions $\bar{x}y.P$ and $x(y).P$ are represented as $(out\ x\ y\ P)$ and $(in\ x\ (\lambda y.P))$, respectively. We use $|$ (written as infix) of type $proc \to proc \to proc$ to denote parallel composition and ν of type $(name \to proc) \to proc$ to denote restriction. To encode the labeled transition system [MMP96] of the π-calculus, we introduce another type *action* with three constructors for it: τ of type *action* denotes the silent action and $\downarrow$ and $\uparrow$, both of type $name \to name \to action$, denote the input and output, respectively, on one named channel with a named value. The transition semantics uses two predicates: $\longrightarrow$, which takes three arguments of type *proc*, *action*, and *proc*; and $\longrightarrow\!\!\!\!\!\rightharpoonup$, which takes three arguments of type *proc*, $name \to action$, and $name \to proc$. The first of these predicates encodes transitions involving free values and the second encodes transitions involving bound values. Below we specify the main transition rules for the π-calculus.

$$\frac{}{out\ x\ y\ P \xrightarrow{\uparrow xy} P}\ \text{output} \qquad \frac{}{in\ x\ M \xrightarrow{\downarrow x} M}\ \text{input}$$

$$\frac{\forall y(My \xrightarrow{\uparrow xy} M'y)}{\nu M \xrightarrow{\uparrow x} \nu M'}\ \text{open} \qquad \frac{P \xrightarrow{\downarrow x} M \qquad Q \xrightarrow{\uparrow x} N}{P|Q \xrightarrow{\tau} \nu\lambda n((Mn)|(Nn))}\ \text{close}$$

$$\frac{P \xrightarrow{A} M}{P|Q \xrightarrow{A} \lambda n((Mn)|Q)}\ \text{par} \qquad \frac{P \xrightarrow{\downarrow x} M \qquad Q \xrightarrow{\uparrow xy} Q'}{P|Q \xrightarrow{\tau} (My)|Q'}\ \text{L-com}$$

One advantage of this style of specification over the traditional one [MPW92] is the absence of complicated side-conditions on variables: they are handled directly by the logical mechanisms described above.

Open Problems

There are many open questions, however, regarding λ-tree syntax. While some work on efficient implementation support for this style of representation has been prototyped, extensive use of those ideas on large scale applications have not yet been taken. Using induction to reason about computations specified using hypothetical and generic judgments is difficult, but some approaches have been proposed recently [MM97]. Logic programming languages and rewriting systems [Nip91] are the only ones programming language paradigms that have successfully supported λ-tree syntax: it would be quite interesting to see if other programming languages can encompass this approach to representation.

References

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *Proceedings of teh 1997 Symposium on Logic and Computer Science*, 1997.

[MMP96] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus: Preliminary report. *Electronic Notes in Theoretical Computer Science*, 3, 1996.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

[Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In J.-P. Jouannaud, editor, *Proc. 1993 Coll. Trees in Algebra and Programming*. Springer Verlag LNCS, 1993.