Foundational Aspects of Syntax [1]

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. A common characteristic of these syntactic objects is that they contain variable binders, such as quantifiers, scoping operators, and parameters. The presence of binders complicates formal specifications and symbolic processing.

Consider, for example, a function definition of the form

$$f(x) = \text{let } y = e \text{ in } x + y.$$

When analyzing or transforming a program containing the call $f(e')$, we might wish to replace $f(e')$ with the body of f in which x is substituted by e'. But we cannot simply apply the substitution $[x \mapsto e']$ because a free variable could be captured. For example, if e' is the expression y, naive substitution would yield the expression (let $y = e$ in $y + y$), which is incorrect.

Binders are often treated in traditional specifications by adding side conditions on variables. Consider, for example, the following (late semantics) rule of the π-calculus [MPW92], expressing how bound input propagates in a context of parallel processes:

$$\frac{P \xrightarrow{x(y)} P'}{P|Q \xrightarrow{x(y)} P'|Q} \quad y \notin \mathit{freevar}(Q).$$

Here, the scope of the binding for y is intended to be over P' only. The side condition is necessary to avoid capturing possibly free occurrences of y in Q.

[1] To appear in the *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, edited by Pierpaolo Degano, Roberto Gorrieri, Alberto Marchetti-Spaccamela, and Peter Wegner, March or June 1999.

Such side conditions on variables are a burden both for formal reasoning and implementations.

The problem with both the above examples is that in the representation the scoping nature of binders is lost: bound and free variables are represented in the same way. Choosing a representation which provides means to encode and operate directly on abstractions is highly desirable for building formal theories and computer systems involving such syntactic objects. Fortunately, there have been a large number of advances in the theory of syntax, particularly resulting from the proof theory of intuitionistic logic, higher-type quantification, and dependent λ-calculus, that suggests an approach [MN87, PE88], which we call here the *λ-tree syntax*. In the literature, this approach is also known as *higher-order abstract syntax* [PE88].

The conventional approach: parse trees

Expressions to be read by humans are often represented by strings. Strings, however, contain whitespaces, brackets, keywords, 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* (also called *abstract syntax 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 parse trees: the concepts of bound variable, scope, α-conversion, and substitution are not directly supported. In the above 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.

The λ-tree syntax

The λ-tree representation enriches parse trees in the following two ways.

First, bindings are encoded using λ-abstractions, following Church's technique for encoding universal and existential quantifiers [Chu40]. The abstracted expressions are given new syntactic categories formed using the type arrow constructor. For example, $\lambda x. \supset (p\ a\ x)\ (q\ (f\ x)\ a)$ represents an abstraction of a term over a formula and has the type $term \rightarrow formula$. The universal and existential quantifiers in the example above can then be modified to take one argument of type $term \rightarrow formula$ instead of the two arguments of type *variable* and *formula*. In this way, the category *variable* is no longer necessary (being subsumed by the corresponding notion in the λ-calculus) and the expression above would be represented by the following λ-term of type *formula*:

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

Second, α-conversion is part of this representation in the sense that two parse trees which differ only in 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 $\lambda x.tx$ whenever 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 untyped λ-terms, let application be denoted by the constructor *app* of type $tm \rightarrow tm \rightarrow tm$, and let abstraction be denoted by the constructor *abs* of type $(tm \rightarrow tm) \rightarrow tm$. For example, the untyped λ-term $\lambda x.xx$ would be encoded as the term $abs(\lambda x.\ app\ x\ x)$ of type *tm*. Two α-equivalent closed untyped λ-terms translate to α-equivalent terms of type *tm*. While this encoding is not surjective, it is the case that every close term of type *tm* is $\alpha\beta\eta$-equivalent to a term that is an encoding of an untyped λ-term.

Computing on λ-trees

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

Determining the structure of λ-trees. Matching or unification modulo $\alpha\eta$-conversion, as we shall see, is not enough to decompose λ-trees adequately. Consider, for example, 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. Because substitution does not allow the capturing of free variables, all instances of this pattern would be expressions in which the universally quantified variable is vacuous. The pattern $(\forall\ (\lambda u. \supset\ (P\ u)\ (Q\ u)))$

will work, however, if we admit 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 α and 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 unification 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 $(\lambda x.t)x$ is equal to t. If meta-variables are applied only to distinct bound variables, then α and β_0 are complete with respect to β, and matching and unification are decidable and unitary [Mil91], and can be solved in linear time [Qia93].

Recursion over λ-trees. In order to compute with λ-trees, it must be possible to define recursion over them. This requires understanding how one "descends" 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 logic 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 *arr* 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 specify the *typeof* judgment.

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

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 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 de Bruijn numbers. The hypothetical judgment (the meta-level implication) implicitly handles the typing context, and the generic judgment (the universal quantifier) implicitly handles the bound variable names by via the use of meta-level eigenvariables.

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* $\rightarrow$ *name* $\rightarrow$ *proc* $\rightarrow$ *proc* and the input prefix is the constructor *in* of type *name* $\rightarrow$ (*name* $\rightarrow$ *proc*) $\rightarrow$ *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* $\rightarrow$ *proc* $\rightarrow$ *proc* to denote parallel composition and ν of type (*name* $\rightarrow$ *proc*) $\rightarrow$ *proc* to denote restriction. To encode the labeled transition system of the π-calculus [MPW92], we introduce another type *action* with three constructors for it: τ denotes the silent action and $\downarrow$ and $\uparrow$, both of type *name* $\rightarrow$ *name* $\rightarrow$ *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 $\xrightharpoonup{}$, which takes three arguments of type *proc*, *name* $\rightarrow$ *action*, and *name* $\rightarrow$ *proc*. The first of these predicates encodes transitions involving free values and the second encodes transitions involving bound values. Below we specify a few transition rules for the π-calculus.

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

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

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

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. In particular, the par rule above implements the rule displayed in the introduction but without the need of an explicit side condition. When examining the specification of the full π-calculus, most inference rules require only meta-level β_0 and not full β-conversion. If one considers the subset of the π-calculus that arises from dropping those the rules requiring β-conversion (L-com in the inference rules above), the resulting calculus happens to be a natural subset of the π-calculus, independently investigated in the literature under the name π_I [San96].

Future work

There are several challenging topics related to λ-tree syntax. We list a few of them here.

Implementation. While some work on designing and implementing support for this style of representation has been completed [NW98], it is still unknown how well these ideas will work in large scale applications. Logic programming languages [MN87, Pfe89] and rewriting systems [Nip91] are the only programming language paradigms that have successfully supported λ-tree syntax: it would be interesting to see if other programming languages can encompass this approach to representation.

Semantics. For conventional specifications using parse trees syntax, well understood semantic tools are available, such as those of initial algebras and models for equality. Similar tools have not yet been developed to handle λ-tree syntax. Since the logic that surrounds λ-tree syntax is that of intuitionistic logic, Kripke models are likely to be useful.

Techniques. Good techniques for reasoning inductively have still to be formulated for λ-tree syntax. A starting point could be the work of McDowell [McD97, MM97], which proposes a logic that includes λ-tree syntax and natural number induction.

Properties. When specifications comply with certain restrictions, interesting properties for the specified formalism can be inferred. For instance, if a transition system is in the tyft/tyxt format then bisimulation is a congruence [GV92]. It would be interesting to study analogous results for specifications involving λ-trees. Note that restrictions like the ones imposed by the tyft/tyxt format are probably too restrictive for λ-tree specifications since they are tailored for term-decomposition in parse trees only.

References

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

[GV92] J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[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.

[McD97] Raymond McDowell. *Reasoning in a Logic with Definitions and Induction.* PhD thesis, University of Pennsylvania, December 1997.

[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.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, 379–388, San Francisco, 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 II. *Information and Computation*, pages 41–77, 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.

[NW98] Gopalan Nadathur and Debra Sue Wilson. A Notation for Lambda Terms: A Generalization of Environments. *Theoretical Computer Science* 198(1-2): 49-98 (1998).

[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*,199–208. ACM Press, 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.

[San96] Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.