Abstract syntax for variable binders: An overview

Dale Miller

Department of Computer Science and Engineering
220 Pond Laboratory, The Pennsylvania State University
University Park, PA 16802-6106 USA dale@cse.psu.edu

Abstract. 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, formal parameters, and blocks. It is a common observation that representing such binders using only first-order expressions is problematic since the notions of bound variable names, free and bound occurrences, equality up to alpha-conversion, substitution, etc., are not addressed naturally by the structure of first-order terms (labeled trees). This overview describes a higher-level and more declarative approach to representing syntax within such computational systems. In particular, we shall focus on a representation of syntax called *higher-order abstract syntax* and on a more primitive version of that representation called λ*-tree syntax*.

1 How abstract is your syntax?

Consider writing programs in which the data objects to be computed are syntactic structures, such as programs, formulas, types, and proofs, all of which generally involve notions of abstractions, scope, bound and free variables, substitution instances, and equality up to renaming of bound variables. Although the data types available in most computer programming languages are rich enough to represent all these kinds of structures, such data types do not have direct support for these common characteristics. Instead, "packages" need to be implemented to support such data structures. For example, although it is trivial to represent first-order formulas in Lisp, it is a more complex matter to write Lisp code to test for the equality of formulas up to renaming of variables, to determine if a certain variable's occurrence is free or bound, and to correctly substitute a term into a formula (being careful not to capture bound variables). This situation is the same when structures like programs or (natural deduction) proofs are to be manipulated and if other programming languages, such as Pascal, Prolog, and ML, replace Lisp.

Generally, syntax is classified into *concrete* and *abstract* syntax. The first is the textual form of syntax that is readable and typable by a human. This representation of syntax is implemented using strings (arrays or lists of characters).

The advantages of this kind of syntax representation are that it can be easily read by humans and involves a simple computational model based on strings. The disadvantages of this style of representation are, however, numerous and serious. Concrete syntax contains too much information not important for many manipulations, such as white space, infix/prefix notation, and keywords; and important computational information is not represented explicitly, such as recursive structure, function–argument relationship, and the term–subterm relationship.

The costs of computing on concrete syntax can be overcome by parsing concrete syntax into *parse trees* (often also called *abstract syntax*). This representation of syntax is implemented using first-order terms, labeled trees, or linked lists, and it is processed using constructors and destructors (such as car/cdr/cons in Lisp) or using first-order unification (Prolog) or matching (ML). The advantages to this representation are clear: the recursive structure of syntax is immediate, recursion over syntax is easily accommodated by recursion in most programming languages, and the term-subterm relationship is identified with the tree-subtree relationship. Also, there are various semantics approaches, such as algebra, that provide mathematical models for many operations on syntax. One should realize, however, that there are costs associated with using this more abstract representation. For example, when moving to greater abstraction, some information is lost: for example, spacing and indenting of the concrete syntax is (generally) discarded in the parse tree syntax. Also, implementation support is needed to provide recursion and linked lists. These costs assocated with using parse tree synatx are generally accepted since one generally does not mind the loss of pagination in the original syntax and since a few decades of programming language research has yielded workable and effective runtime environments that support the required dynamic memory demands required to process parse trees.

When representing syntax containing bound variables, there are, however, significant costs involved in not using a representation that is even more abstract than parse trees since otherwise the constellation of concepts surrounding bindings needs to be implemented by the programmer. There are generally two approaches to providing such implementations. The first approach treats bound variables as global objects and programs are then written to determine which of these global objects are to be considered free (global) and which are to be considered scoped. This approach is quite natural and seems the simplest to deploy. It requires no special meta-level support (all support must be provided explicitly by the programmer) and is the approach commonly used in text books on logic. A second approach uses the nameless dummies of de Bruijn [2]. Here, first-order terms containing natural numbers are used to describe alpha-equivalence classes of λ-terms: syntax is abstracted by removing bound variable names entirely. There has been a lot of success in using nameless dummies in low-level compilation of automated deduction systems and type systems. Consider, for instance, the work on explicit substitutions of Nadathur [25, 28] and Abadi, Cardelli, Curien, and Lévy [1]. Nadathur, for example, has recently built a compiler and abstract machine that exploits this representation of syntax [27].

While successful at implementing bound variables in syntax, nameless dummies, however do not provide a high-level and declarative treatment of binding.

We will trace the development of the ideas behind a third, more abstract form of syntactic representation, called *λ-tree syntax* [23] and the closely related notion of *higher-order abstract syntax* [36].

Logic embraces and explains elegantly the nature of bound variables and substitution. These are part of the very fabric of logic. So it is not surprising that our story starts and mostly stays within the area of logic.

2 Church's use of λ-terms within logic

In [3], Church presented a higher-order logic, called the *Simple Theory of Types* (STT), as a foundation for mathematics. In STT, the syntax of formulas and terms is built on simply typed λ-terms. The axioms for STT include those governing the logical connectives and quantifiers as well as the more mathematical axioms for infinity, choice, and extensionality. The λ-terms of STT are also equated using the following equations of α, β, and η-conversion.

(α)	$\lambda x.M = \lambda y.M[y/x]$,	provided y is not free in M
(β)	$(\lambda x.M)\ N = M\ [N/x]$	
(η)	$\lambda x.(M\ x) = M$,	provided x is not free in M

Here, the expression $M[t/x]$ denotes the substitution of t for the variable x in M in which bound variables are systematically changed to avoid variable capture.

Church made use of the single binding operation of λ-abstraction to encode all of the other binding operators present in STT: universal and existential quantification as well as the definite description and choice operators. This reuse of the λ-binder in these other situations allows the notions of bound and free variables occurrences and of substitution to be solved once with respect to λ-binding and then be used to solve the associated problems with these other binding operations. In recent years, this same economy has been employed in a number of logical and computational systems.

Church used the λ-binder to introduce a new syntactic type, that of an *abstraction* of one syntactic type over another. For example, Church encoded the universal quantifier using a constant Π that instead of taking two augments, say, the name of a bound variable and the body of the quantifier, took one argument, namely, the abstraction of the variable over the body. That is, instead of representing universal quantification as, say, $\Pi(x, B)$ where Π has the type $\tau * o \to o$ (here, o is the type of formulas and τ is a type variable), it is represented as $\Pi(\lambda x.B)$, where Π has the type $(\tau \to o) \to o$. (This latter expression can be abbreviated using more familiar syntax as $\forall x.B$.) The λ-binder is used to construct the arrow $(\to)$ type. Similarly, the existential quantifier used a constant Σ of type $(\tau \to o) \to o$ and the choice operator ι had type $(\tau \to o) \to \tau$: both take an abstraction as their argument.

Since Church was seeking to use this logic as a foundations for mathematics, λ-terms were intended to encode rich collections of mathematical functions that

could be defined recursively and which were extensional. By adding higher-order quantification and axioms for infinity, extensionality, and choice, the equality of λ-term was governed by much more than simply the equations for α, β, and η-conversion. Hence, λ-abstractions could no longer be taken for expressions denoting abstractions of one syntactic types over another. For example, the formula $\Pi(\lambda x.(p\ x) \wedge q)$ would be equivalent and equal to the formula $\Pi(\lambda x.q \wedge (p\ x))$: there is no way in STT to separate these two formulas. Thus, the domain o became associated to the *denotation* or *extension* of formulas and not with their *intension*.

3 Equality modulo $\alpha\beta\eta$-conversion

One way to maintain λ-abstraction as the builder of a syntactic type is to weaken the theory of STT significantly, so that λ-terms no longer represent general functional expressions. The resulting system may no longer be a general foundations for mathematics but it may be useful for specifying computational processes. The most common approach to doing this weakening is to drop the axioms of infinity, extensionality, and choice. In the remaining theory, λ-terms are governed only by the rules of α, β, and η-conversion. The simply typed λ-calculus with an equality theory of α, β, η is no longer a general framework for functional computation although its is still rather rich [39].

The presence of β-conversion in the equality theory means that object-level substitution can be specified simply by using the meta-level equality theory. For example, consider the problem of instantiating the universal quantifier $\forall x.B$ with the term t to get $B[t/x]$. Using Church's representation for universal quantification, this operation can be represented simply as taking the expression (ΠR) and the term t and returning the term $(R\ t)$. Here, R denotes the abstraction $\lambda x.B$, so $(R\ t)$ is a meta-level β-redex that is equal to $B[t/x]$. Thus, β-reduction can encode object-level substitution elegantly and simply. For example, consider the following signature for encoding terms and formulas in a small object-logic:

$$\begin{array}{ll} \forall, \exists : (\mathit{term} \to \mathit{formula}) \to \mathit{formula} & a, b : \mathit{term} \\ \supset : \mathit{formula} \to \mathit{formula} \to \mathit{formula} & f : \mathit{term} \to \mathit{term} \to \mathit{term} \\ r, s : \mathit{term} \to \mathit{formula} & t : \mathit{formula}. \end{array}$$

The λ-term $\forall\lambda x.\exists\lambda y.\ r\ (f\ x\ y) \supset s\ (f\ y\ x)$ is of type *formula* and is built by applying the constant $\forall$ to the a λ-term of type $\mathit{term} \to \mathit{formula}$, the syntactic type of a term abstracted over a formula. This universally quantified object-level formula can be instantiated with the term $(f\ a\ b)$ by first matching it with the expression $(\forall\ R)$ and then considering the term $(R\ (f\ a\ b))$. Since R will be bound to a λ-expression, this latter term will be a meta-level β-redex. If β is part of our equality theory, then this term is equal to

$$\exists\lambda y.r\ (f\ (f\ a\ b)\ y) \supset s\ (f\ y\ (f\ a\ b)),$$

which is the result of instantiating the universal quantifier.

Huet and Lang [13] were probably the first people to use a simply typed λ-calculus modulo α, β, η to express program analysis and program transformation steps. They used second-order variables to range over program abstractions and used second-order matching to bind such variables to such abstractions. The reliance on β-conversion also meant that the matching procedure was accounting for object-level substitution as well as abstractions. Second-order matching is NP-complete, in part, because reversing object-level substitution is complicated. There was no use of logic in this particular work, so its relationship to Church's system was rather minor.

In the mid-to-late 80's, two computational systems, Isabelle [32] and λProlog [26], were developed that both exploited the intuitionistic theory of implications, conjunctions, and universal quantification at all non-predicate types. In Isabelle, this logic was implemented in ML and search for proofs was governed by an ML implementation of tactics and tacticals. This system was intended to provide support for interactive and automatic theorem proving. λProlog implemented this logic (actually a extension of it called *higher-order hereditary Harrop formulas* [22]) by using a generalization of Prolog's depth-first search mechanism. Both systems implemented versions of unification for simply typed λ-terms modulo α, β, η conversion (often called *higher-order unification*). The general structuring of those unification procedures was fashioned on the unification search processes described by Huet in [12]. In λProlog, it was possible to generalize the work of Huet and Lang from simple template matching to more general analysis of program analysis and transformation [19–21].

The dependent typed λ-calculus LF [10] was developed to provide a high-level specification language for logics. This system contained quantification at higher-types and was based on an equality theory that incorporated α, β, and η-conversion (of dependent typed λ-calculus). Pfenning implemented LF as the Elf system [33] using a λProlog-style operational semantics. The earliest version of Elf implemented unification modulo α, β, η.

It was clear that these computer systems provided new ways to compute on the syntax of expressions with bound variables. The availability of unification and substitution in implementations of these meta-logics immediately allowed bound variable names to be ignored and substitutions for all data structures that contain bound variables to be provided directly.

Pfenning and Elliott in [36] coined the term *higher-order abstract syntax* for this new style of programming and specification. They also analyzed this style of syntactic specification and concluded that it should be based on an enrichment of the simply typed λ-calculus containing products and polymorphism, since they found that these two extensions were essential features for practical applications. To date, no computational system has been built to implement this particular notion of higher-order abstract syntax. It appears that in general, most practical applications can be accommodated in a type system without polymorphism or products. In practice, higher-order abstract syntax has generally come to refer to the encoding and manipulating of syntax using either simply or dependently typed λ-calculus modulo α, β, and η-conversion.

4 A weaker form of β-conversion

Unification modulo α, β, and η conversion of λ-terms, either simply typed or dependently typed, is undecidable, even when restricted to second-order. Complexity results for matching are not fully know, although restricting to second-order matching is known to be NP-complete. Thus, the equalities implemented by these computer systems (Isabelle, λProlog, and Elf) are quite complex. This complexity suggests that we should find a simpler approach to using the λ-binder as a constructor for a syntactic type of abstraction. The presence of bound variables in syntax is a complication, but it should not make computations on syntax overly costly. We had some progress towards this goal when we weaken Church's STT so that λ-abstractions are not general functions. But since the equality and unification remains complex, it seems that we have not weakened that theory enough.

We can consider, for example, getting rid of β-conversion entirely and only consider equality modulo α, η-conversion. However, this seems to leave the equality system too weak. To illustrate that weakness, consider solving the following match, where capital letters denote the match variables:

$$\forall\lambda x(P \wedge Q) = \forall\lambda y((ry \supset sy) \wedge t).$$

There is no substitution for P and Q that will make these two expressions equal modulo α and η-conversion: recall that we intend our meta-level to be a logic and that the proper logical reading of substitution does not permit variable capturing substitutions. Hence, the substitution

$$\{P \mapsto (rx \supset sx), Q \mapsto t\}$$

does not equate these two expressions: substituting into the first of these two terms produces a term equal to $\forall\lambda z((rx \supset sx) \wedge t)$ and not equal to the intended term $\forall\lambda y((ry \supset sy) \wedge t)$. If we leave things here, it seems impossible to do interesting pattern matching that can explore structure underneath a λ-abstraction.

If we change this match problem, however, by raising the type of P from *formula* to *term* $\rightarrow$ *formula* and consider the following matching problem instead,

$$\forall\lambda x(Px \wedge Q) = \forall\lambda y((ry \supset sy) \wedge t)$$

then this match problem does, in fact, have one unifier, namely,

$$\{P \mapsto \lambda w(rw \supset sw), Q \mapsto t\}.$$

For this to be a unifier, however, the equality theory we use must allow $((\lambda w.rw \supset sw)\ x)$ to be rewritten to $(rx \supset sx)$. Clearly β will allow this, but we have really only motivated a much weaker version of β-conversion, in particular, the case when a λ-abstraction is applied to a bound variable that is not free in the abstraction. The restriction of β to the rule $(\lambda x.B)y = B[y/x]$, provided y is not free in $(\lambda x.B)$, is called β_0-conversion [15]. In the presence of α-conversion, this

rule can be written more simply and without a proviso as $(\lambda x.B)x = B$. Our example can now be completed by allowing the equality theory to be based on α, β_0, and η-conversion. In such a theory, we have

$$\forall\lambda x((\lambda w(rw \supset sw)x) \wedge t) = \forall\lambda y((ry \supset sy) \wedge t).$$

If the λ-binder can be viewed as introducing a syntactic domain representing the abstraction of a bound variable from a term, then we can view β_0 as the rule that allows destructing a λ-binder by replacing it with a bound variable.

It is easy to imagine generalizing the example above to cases where match variables have occurrences in the scope of more than one abstraction, where different syntax is being represented, and where unification and not matching is considered. In fact, when examining typical λProlog programs, it is clear that most instances of β-conversion performed by the interpreter are, in fact, instances of β_0-conversion. Consider, for example, a term with a free occurrence of M of the form

$$\lambda x \ldots \lambda y \ldots (M\ y\ x) \ldots$$

Any substitution for M applied to such a term introduces β_0 redexes only. For example, if M above is instantiated with a λ-term, say $\lambda u\lambda v.t$, then the only new β-redex formed is $((\lambda u\lambda v.t)\ y\ x)$. This term is reduced to normal form by simply renaming in t the variables u and v to y and x — a very simple computation. Notice that replacing a β_0-redex $(\lambda x.B)y$ with $B[y/x]$ makes the term strictly smaller, which stands in striking contrast to β-reduction, where the size of terms can grow explosively.

5 L_λ-Unification

In [15], Miller introduced a subset of hereditary Harrop formulas, called L_λ, such that the equality theory of α, β, η only involved α, β_0, η rewritings. In that setting, Miller showed that unification of λ-terms is decidable and unary (most general unifiers exist when unifiers exist).

When L_λ is restricted to simply comparing two atomic formula or two terms, it is generally referred to as *L_λ-unification* or as *higher-order pattern unification*. More precisely, in this setting a *unification problem* a set of ordered pairs

$$\{(t_1, s_1), \ldots, (t_n, s_n)\},$$

where for $i = 1, \ldots, n$ and where t_i and s_i are simply typed λ-terms of the same type. Such a unification problem is an L_λ-unification problem if every free variable occurrence in that problem is applied to at most distinct bound variables. This severe restriction on the applications of variables of higher-type is the key restriction of L_λ.

This kind of unification can be seen both as a generalization of first-order unification and as a simplification of the unification process of Huet [12]. Any β-normal λ-term has the top-level structure $\lambda x_1 \ldots \lambda x_p(h\ t_1 \ldots\ t_q)$ where $p, q \geq 0$, the *binder* $x_1, \ldots, x_p$ is a list of distinct bound variables, the *arguments* $t_1, \ldots, t_q$

are β-normal terms, and the *head* h is either a constant, a bound variable (*i.e.*, a member of $\{x_1, \ldots, x_p\}$), or a free variable. (We shall sometimes write $\bar{x}$ to denote a list of variables $x_1, \ldots, x_n$, for some n.) If the head is a free variable, the term is called *flexible*; otherwise, it is called *rigid*. Notice that if a term in L_λ is flexible, then it is of the form $\lambda x_1 \ldots \lambda x_n . V\ y_1\ \ldots\ y_p$ where each list $x_1, \ldots, x_n$ and $y_1, \ldots, y_p$ contain distinct occurrences of variables and where the the set $\{y_1, \ldots, y_p\}$ is a subset of $\{x_1, \ldots, x_n\}$. Pairs in unification problems will be classified as either rigid-rigid, rigid-flexible, flexible-rigid, or flexible-flexible depending on the status of the two terms forming that pair. We can always assume that the two terms in a pair have the same binder: if not, use η to make the shorter binder longer and α to get them to have the same names.

We present the main steps of the unification algorithm (see for [15] for a fuller description). Select a pair in the given unification and choose the appropriate steps from the following steps.

Rigid-rigid step. If the pair is rigid-rigid and both terms have the same head symbol, say, $\langle \lambda \bar{x}.ht_1 \ldots t_n, \lambda \bar{x}.hs_1 \ldots s_n \rangle$, then replace that pair with the pairs $\langle \lambda \bar{x}.t_1, \lambda \bar{x}.s_1 \rangle, \ldots, \langle \lambda \bar{x}.t_n, \lambda \bar{x}.s_n \rangle$ and continue processing pairs. If the pair has different heads, then there is no unifier for this unification problem.

Flexible-flexible step. If the pair is flexible-flexible, then it is of the form $\langle \lambda \bar{x}.Vy_1 \ldots y_n, \lambda \bar{x}.Uz_1 \ldots z_p \rangle$ where $n, p \geq 0$ and where the lists $y_1, \ldots, y_n$ and $z_1 \ldots z_p$ are both lists of distinct variables and are both subsets of the binder $\bar{x}$. There are two cases to consider.

Case 1. If V and U are different, then this pair is solved by the substitution $[V \mapsto \lambda \bar{y}.W\bar{w}, U \mapsto \lambda \bar{z}.W\bar{w}]$, where W is a new free variable and $\bar{w}$ is a list enumerating the variables that are in both the list $\bar{y}$ and the list $\bar{z}$.

Case 2. If V and U are equal, then, given the typing of λ-terms, p and n must also be equal. Let $\bar{w}$ be an enumeration of the set $\{y_i \mid y_i = z_i, i \in \{1, \ldots, n\}\}$. We solve this pair with the substitution $[V \mapsto \lambda \bar{y}.W\bar{w}]$ (notice that this is the same via α-conversion to $[V \mapsto \lambda \bar{z}.W\bar{w}]$), where W is a new free variable.

Flexible-rigid step. If the pair is flexible-rigid, then that pair is of the form $\langle \lambda \bar{x}.Vy_1 \ldots y_n, r \rangle$. If V has a free occurrence in r then this unification has no solution. Otherwise, this pair is solved using the substitution $[V \mapsto \lambda y_1 \ldots \lambda y_n.r]$.

Rigid-flexible step. If the pair is rigid-flexible, then switch the order of the pair and do the flexible-rigid step.

Huet's process [12], when applied to such unification problems, produces the same reduction except for the flexible-flexible steps. Huet's procedure actually does pre-unification, leaving flexible-flexible pairs as constraints for future unifications since general (non-L_λ) flexible-flexible pairs have too many solutions to actually enumerate effectively. Given the restrictions in L_λ, flexible-flexible pairs can be solved simply and do not need to be suspended.

Qian has shown that L_λ-unification can be done in linear time and space [38] (using a much more sophisticated algorithm than the one hinted at above). Nipkow has written a simple functional implementation of L_λ-unification [30] and has also showed that results concerning first-order critical pairs lift naturally to the L_λ setting [29].

It was also shown in [15] that L_λ-unification can be modified to work with untyped λ-terms. This observation means, for example, that the results about L_λ can be lifted to other type systems, not just the simple theory of types. Pfenning has done such a generalization to a dependent typed system [34]. Pfenning has also modified Elf so that pre-unification essentially corresponds to L_λ-unification: unification constraints that do not satisfy the L_λ restriction on free variables are delayed. The equality theory of Elf, however, is still based on full β-conversion.

Notice that unification in L_λ is unification modulo α, β_0, and η but unification modulo α, β_0, and η on unrestricted terms is a more general problem. For example, if g is a constant of type $i \to i$ and F is a variable of type $i \to i \to i$, the equation $\lambda x.F\ x\ x = \lambda y.g\ y$ has two solutions modulo α, β_0, η, namely, $F \mapsto \lambda u \lambda v.g\ u$ and $F \mapsto \lambda u \lambda v.g\ v$. Notice that this unification problem is not in L_λ since the variable F is applied to the bound variable x twice. As this example shows, unification modulo α, β_0, η is not necessarily unary.

6 Logic programming in L_λ

Successful manipulation of syntax containing bound variables is not completely achieved by picking a suitable unification and equality theory for terms. In order to compute with λ-trees, it must be possible to define recursion over them. This requires understanding how one "descends" into a λ-abstraction $\lambda x.t$ in a way that is independent from the choice of the name x. A key observation made with respect to the design of such systems as Isabelle, λProlog, and Elf is that such a declarative treatment of bound variables requires the *generic* and *hypothetical* judgments that are found in intuitionistic logic (via implication and universal quantification) and associated dependent typed λ-calculi. The need to support universal quantification explicitly forces one to consider unification with both free (existentially quantified) variables and universally quantified variables. To handle unification with both kinds of variables present, Paulson developed $\forall$-*lifting* [31] and Miller developed *raising* [18] ($\forall$-lifting can be seen as backchaining followed by raising).

The name L_λ is actually the name of a subset of the hereditary Harrop formula used as a logical foundation for λProlog, except for restrictions on quantified variables made to ensure that only L_λ-unification occurs in interpreting the language. (L_λ is generally also restricted so as not to have the predicate quantification that is allowed in λProlog.) While we do not have adequate space here to present the full definition of the L_λ logic programming language (for that, see [15]) we shall illustrate the logic via a couple of examples.

We shall use inference figures to denote logic programming clauses in such a way that the conclusion and the premise of a rule corresponds to the head and body of the clause, respectively. For example, if A_0, A_1, and A_2 are syntactic variables for atomic formulas, then the two inference figures

$$\frac{A_1 \qquad A_2}{A_0} \qquad\qquad \frac{\forall x(A_1 \supset A_2)}{A_0},$$

denote the two formulas

$$\forall \bar{y}(A_1 \wedge A_2 \supset A_0) \quad \text{and} \quad \forall \bar{y}(\forall x(A_1 \supset A_2) \supset A_0)$$

The list of variables $\bar{y}$ is generally determined by collecting together the free variables of the premise and conclusion. In the inference figures, the corresponding free variables will be denoted by capital letters. The first of these inference rules denotes a simple Horn clause while the second inference rule is an example of a hereditary Harrop formula. The theory of higher-order hereditary Harrop formulas [22] provides an adequate operational and proof theoretical semantics for these kinds of clauses. The central restriction taken from L_λ-unification must be generalized to this setting. Note that in our examples this restriction implies that a variable in the list $\bar{y}$ can be applied to at most distinct variables that are either λ-bound or universally bound in the body of the clause.

Consider, for example, representing untyped λ-terms and simple types. Let tm and ty be two types for these two domains, respectively. The following four constants can be used to build objects in these two domains.

$$\begin{array}{ll} app : tm \to tm \to tm & arr : ty \to ty \to ty \\ abs : (tm \to tm) \to tm & i : ty \end{array}$$

The constants app and abs are constructors for applications and abstractions, while the constants arr and i are used to denote functional (arrow) types and a primitive type.

To capture the judgment that an untyped λ-term has a certain simple type, we 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.

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

Notice that the variable R is used in a higher-order fashion since it has an occurrence where it is an argument and an occurrence where it has an argument.

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 names or as de Bruijn numbers. In this specification of the *typeof* judgment, the hypothetical judgment (the intuitionistic implication) implicitly handles the typing context, and the generic judgment (the universal quantifier) implicitly handles the bound variable names via the use of eigenvariables.

Since the application of variables is restricted greatly in L_λ, object-level substitution cannot be handled simply by the equality theory of L_λ. For example, the clause

$$\frac{}{bredex\ (app\ (abs\ R)\ N)\ (R\ N)}$$

defines a predicate that relates the encoding of an untyped λ-term that represents a top-level β-redex to the result of reducing that redex. The formula that encodes

this inference rule does not satisfy the L_λ restriction since the variable R is not applied to a λ-bound variable: notice that instances of $(R\ N)$ might produce (meta-level) β-redexes that are not β_0-redexes. Instead, object-level substitution can be implemented as a simple logic program. To illustrate this, consider the following two classes for specifying equality for untyped λ-terms.

$$\frac{copy\ M\ M' \qquad copy\ N\ N'}{copy\ (app\ M\ N)\ (app\ M'\ N')} \qquad \frac{\forall x\ \forall y\ (copy\ x\ y \supset copy\ (R\ x)\ (S\ y))}{copy\ (abs\ R)\ (abs\ S)}$$

Clearly, the atom $copy\ t\ t'$ is provable from these two clauses if and only if t and t' denote the same untyped λ-term. Given this specification of equality, we can now specify object-level substitution with the following simple clause:

$$\frac{\forall x\ (copy\ x\ N \supset copy\ (R\ x)\ M)}{subst\ R\ N\ M}$$

which axiomatizes a three place relation, where the type of the first argument is $i \to i$ and the type of the other two arguments is i. We can now finally re-implement *bredex* so that it is now an L_λ program:

$$\frac{subst\ R\ N\ M}{bredex\ (app\ (abs\ R)\ N)\ M}$$

The entire specification *bredex* is now an L_λ logic program. For a general approach to accounting for object-level substitution in L_λ, see [16].

For a specific illustration that classical logic does not support the notion of syntax when higher-orders are involved, consider the following signature.

$$p, q, r : term \to o \qquad g : (term \to term) \to term \qquad f : term \to term$$

and the two clauses

$$\frac{p\ X}{r\ (f\ X)} \qquad \frac{\forall x\ (p\ x \supset q\ (U\ x))}{r\ (g\ U)}$$

Using the familiar "propositions-as-types" paradigm, the three atomic formulas $p\ t_1, q\ t_2$, and $r\ t_3$ can be seen as specifying subtypes of the type *term*, that is, they can be read as $t_1 : p, t_2 : q$, and $t_3 : r$. Using this analogy, these two clauses would then read as the type declarations $f : p \to r$ and $g : (p \to q) \to r$. Now consider the question of whether or not there is a term of type r. Simple inspection reveals that there is no term of type r built from these two constants. Similarly, there is no intuitionistic proof of $\exists X.r\ X$ from the two displayed clauses. On the contrary, there is a classical logic proof of $\exists X.r\ X$ from these formulas. We leave it to the reader to ponder how classical logic can be so liberal to allow such a conclusion. (Hint: consider the classical logic theorem $(\exists w.p\ w) \vee (\forall w.\neg p\ w)$.)

7 λ-tree syntax

In contrast to *concrete syntax* and *parse tree syntax*, a third level of syntax representation, named *λ-tree syntax* was introduced in [23]. This approach to syntactic representation uses λ-terms to encode data and L_λ-unification and equality modulo α, β_0, and η to construct and deconstruct syntax. There is no commitment to any particular type discipline for terms nor is typing necessary.

As we have observed, a programming language or specification language that incorporates λ-tree syntax must also provide an abstraction mechanism that can be used to support recursion under term level abstractions. In logic or typed languages, this is achieved using eigenvariables (a notion of bound variable within a proof). Such a mechanism can be described in a logic programming setting, like λProlog, as one where new, scoped constants are introduced to play the role of bound variables.

While supporting λ-tree syntax is more demanding on the languages that implements it, there has been a lot of work in making such implementations feasible. Consider for example the work on explicit substitutions [1, 7, 25, 28] and the abstract machine and compiler Teyjus [27] for λProlog. The Isabelle theorem prover [32] implements L_λ and the Elf system [33] provides an effective implementation of L_λ within a dependently typed λ-calculus.

Support for λ-term syntax does not necessarily need to reside only in logic programming-like systems. In [14] Miller proposed an extension to ML in which pattern matching supported L_λ matching and where data types allowed for the scoped introduction of new constants (locally bound variables). A second type, written `a' => b'`, was introduced to represent the type of syntactically abstracted variables: the usual function type, written `a' -> b'`, was not used for that purpose. It is possible, following the techniques we described for L_λ, to implement in the resulting ML extension, a function `subst` that maps the first domain into the second, that is, `subst` has type `(a' => b') -> (a' -> b')`. To our knowledge, this language has not been implemented.

The need for the new term *λ-tree syntax* instead of the more common term *higher-order abstract syntax* can be justified for a couple of reasons. First, since types are not necessary in this style of representation, the adjective "higher-order", which refers to the order of types for variables and constants, seems inappropriate. Second, higher-order abstract syntax generally denotes the stronger notion of equality and unification that is based on full β-conversion. For example, Pfenning in [35] states that "higher-order abstract syntax supports substitution through λ-reduction in the meta-language". Thus, the term higher-order abstract syntax would not be appropriate for describing projects, such as L_λ and the proposal mentioned above for extending ML, in which β-reduction is not part of the meta-language.

8 Related work

As we have mentioned, Church intended the function space constructor to be strong enough to model mathematical functions and not to support the weaker

notion of representing an abstraction over syntactic types. As a result, we argued that Church's system should be weakened by removing not only the axioms of infinity, choice, and extensionality but also full β-conversion. On the other hand, there has been work in trying to recover higher-order abstract syntax from rich function spaces such as those found in Coq: the main issue there is to restrict the function space constructor to exclude "exotic" terms, like those inhabiting function spaces but which do not denote syntactic abstractions [4–6].

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 might be useful: a simple step in this direction was taken in [17] by recasting the cut-elimination theorem for intuitionistic logic as a kind of initial model. Similarly, the notion of Kripke λ-models due to Mitchell and Moggi [24] could also be quite useful. The LICS 1999 proceedings contained three papers [9, 8, 11] that proposed semantics for abstract syntax containing bound variables that were based (roughly) on using initial models based on certain categories of sheaves. Pitts and Gabbay have used their semantics to develop an extension to ML that supports a notion of syntax somewhat similar to λ-tree syntax [37].

9 Conclusions

One might have some impatience with the idea of introducing a more high-level form of abstract syntax: just implement substitution and the associated support for bound variables and move on! But what we are discussing here is the foundations of syntax. The choices made here can impact much of what is built on top.

There is also the simple observation that with, say, the parse tree representation of syntax, it is natural to use meta-level application to encode object-level application. But application and abstraction are not two features that accidentally appear in the same logic: they are two sides of the same phenomenon, just as introduction and elimination rules in proof theory are two sides of a connective, and they need to be treated together. It should be just as natural to use meta-level abstractions to encode object-level abstractions, and indeed, this is what λ-tree syntax attempts to make possible.

Acknowledgments I would like to thank Catuscia Palamidessi and the reviewers of this paper for their many useful comments that helped to improve this paper's presentation. This work was supported in part by NSF Grants INT-9815645 and CCR-9803971.

References

1. Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

2. N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.
3. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. Joelle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.
5. Joelle Despeyroux and Andre Hirschowitz. Higher-order abstract syntax with induction in Coq. In *Fifth International Conference on Logic Programming and Automated Reasoning*, pages 159–173, June 1994.
6. Joelle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, April 1997.
7. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Logic in Computer Science*, pages 366–374, 1995.
8. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, 1999.
9. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.
10. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
11. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.
12. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
13. Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
14. Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.
15. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.
16. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
17. Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in LNAI, pages 322–337. Springer-Verlag, 1992.
18. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.
19. Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.
20. Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.
21. 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.

22. 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.
23. Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. In Pierpaolo Degano, Roberto Gorrieri, Alberto Marchetti-Spaccamela, and Peter Wegner, editors, *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, volume 31. ACM, Sep 1999. Article number 10.
24. John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51, 1991.
25. Gopalan Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.
26. Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
27. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, CADE-16, pages 287–291, Trento, Italy, July 1999. Springer LNCS.
28. Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
29. Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Sym. on Logic in Computer Science*, pages 342–349. IEEE, July 1991.
30. Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Sym. on Logic in Computer Science*, pages 64–74. IEEE, June 1993.
31. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.
32. Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
33. 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.
34. Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE, July 1991.
35. Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, volume LNCS 1059, pages 119–134. Springer-Verlag, 1996.
36. 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.
37. Andrew M. Pitts and Murdoch J. Gabbay. A meta language for programming with bound names modulo renaming (preliminary report). Draft January 2000.
38. Zhenyu Qian. Unification of higher-order patterns in linear time and space. *J. Logic and Computation*, 6(3):315–341, 1996.
39. Richard Statman. The typed λ calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.