Abstract Syntax and Logic Programming

September 1991

Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA
dale@cis.upenn.edu

Abstract. When writing programs to manipulate structures such as algebraic expressions, logical formulas, proofs, and programs, it is highly desirable to take the linear, human-oriented, concrete syntax of these structures and parse them into a more computation-oriented syntax. For a wide variety of manipulations, concrete syntax contains too much useless information (*e.g.*, keywords and white space) while important information is not explicitly represented (*e.g.*, function-argument relations and the scope of operators). In parse trees, much of the semantically useless information is removed while other relationships, such as between function and argument, are made more explicit. Unfortunately, parse trees do not adequately address important notions of object-level syntax, such as bound and free object-variables, scopes, alphabetic changes of bound variables, and object-level substitution. I will argue here that the *abstract syntax* of such objects should be organized around α-equivalence classes of λ-terms instead of parse trees. Incorporating this notion of abstract syntax into programming languages is an interesting challenge. This paper briefly describes a logic programming language that directly supports this notion of syntax. An example specifications in this programming language is presented to illustrate its approach to handling object-level syntax. A model theoretic semantics for this logic programming language is also presented.

1. Introduction

Consider writing programs in which the data objects to be computed are syntactic structures such as programs, formulas, types, and proofs. A common characteristic of all these structures is that they involve notions of abstractions, scope, bound and free variables, substitution instances, and equality up to alphabetic changes of bound variables. Although the data types available in most computer programming languages are, of course, 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 programs that correctly substitute a term into a formula (being careful not to capture bound variables), to test for the equality of formulas up to alphabetic variation, and to determine if a certain variable's occurrence is free or bound. This situation is the same

To appear in the *Proceedings of the Second Russian Conference on Logic Programming, September 1991*, edited by A. Voronkov, Lecture Notes in Artificial Intelligence, Springer-Verlag.

Implementation	Strings, text (arrays or lists of characters)
Access	Parsers, editors
Good points	1. Readable, publishable. 2. Simple computational models for implementation (arrays, iteration).
Bad points	1. Contains too much information not important for many manipulations: white space, infix/prefix notation, keywords. 2. Important information is not represented explicitly: recursive structure, function–argument relationship, term–subterm relationship.

Figure 1: Characteristics of concrete syntax

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.

Before proposing an approach to dealing with representing such syntactic structures in a logic programming language, let us consider current practice in representing syntax in computer programs. Generally, syntax is divided into *concrete* and *abstract* syntax. The first is the linear form of syntax that is readable and typable by a human. Figure 1 characterizes some properties of concrete syntax. The bad points can be overcome by parsing concrete syntax into *parse trees*. Figure 2 characterizes some properties of parse trees. The bad points concerning concrete syntax are now properly addressed, although at significant costs. For example, higher levels of support are required for the programming language and runtime system that encode parse trees. Parse trees, however, are so much more convenient and natural to compute with than strings that these additional costs are outweighed by the advantages to the programmer who must write programs to manipulate syntax. The term *abstract syntax* is often identified with parse trees: we shall reserve the former term for the more "abstract" form of syntax described in the next section.

Parse trees are not without their bad points also. In particular, notions of abstraction within syntax are not supported directly. For example, we have the following unfortunate properties of parse trees for representing syntax containing bound variables.

- Bound variables are, like constants, treated as global objects.
- Concepts such as free and bound occurrences of variables are derivative notions, supported not by programming languages but by programs added on top of the data type for parse trees.
- Although alphabetic variants generally denote the same intended object, the correct choice of such variants is unfortunately very often important.
- Substitution is generally difficult to implement correctly.
- An implementation of substitution for one data structure, say first-order formulas, will not work for another, say functional programs.

There are various computer systems that use a different approach to syntax. They all make use of λ-terms modulo the equations of α, β, and η-conversions and implement

Implementation	first-order terms, linked lists
Access	car/cdr/cons in Lisp, first-order unification in Prolog, or matching in ML.
Good points	1. Recursive structure is immediate. 2. Recursion over syntax is easy to specify. 3. Term–subterm relationship is identified with tree-subtree relationship. 4. Algebra provides a model for many operations on syntax.
Bad points	1. Requires higher-level language support: pointers, linked lists, garbage collection, structure sharing. 2. Notions of scope, abstraction, substitution, and free and bound variables occurrences are not supported.

Figure 2: Characteristics of parse trees

various aspects of $\beta\eta$-unification (often called "higher-order" unification). One of the earliest was designed by Huet and Lang [13]: here, only second-order matching was used to decompose syntax. The generic theorem prover Isabelle uses a fragment of intuitionistic logic with quantification at higher-order types. The Isabelle implementation includes $\beta\eta$-unification at all finite types. The language λProlog [21] is an extension of Prolog that includes, among other things, $\beta\eta$-unification at all finite types. The Elf programming language [23] is a logic programming language implementation of the LF specification language [12] in a style similar to λProlog.

This short paper is organized as follows. In the next section, we shall motivate a notion of abstract syntax that is more "high-level" than parse trees. Section 3 presents the logic programming language $\mathcal{M}$ that incorporates such abstract syntax. In Section 4 an example specification in $\mathcal{M}$ is presented. Finally, a model theory for $\mathcal{M}$ is given in Section 5.

2. Motivating abstract syntax

Consider the recursive structure of first-order terms over the following signature.

$$\Sigma = \{a : i, \quad b : i, \quad f : i \to i, \quad g : i \to i \to i\}$$

Here, i is a primitive type (or sort). These four typed constants can be encoded as the following four inference rules for determining which first-order terms over Σ are correctly constructed.

$$\frac{\Sigma \vdash X : i}{\Sigma \vdash f\ X : i} \qquad \frac{\Sigma \vdash X : i \qquad \Sigma \vdash Y : i}{\Sigma \vdash g\ X\ Y : i}$$

$$\frac{}{\Sigma \vdash a : i} \qquad \frac{}{\Sigma \vdash b : i}$$

The following is a proof that the term $g\ (f\ a)\ b$ is a correctly formed first-order term (of type i).

$$\dfrac{\dfrac{\dfrac{}{\Sigma\ \vdash\ a : i}}{\Sigma\ \vdash\ f\ a : i} \qquad \dfrac{}{\Sigma\ \vdash\ b : i}}{\Sigma\ \vdash\ g\ (f\ a)\ b : i}$$

Notice that the signature Σ does not change in a proof: it is global and does not need to be written as part of each inference rule.

To consider the structure of λ-terms, let $\Sigma' = \Sigma \cup \{h : (i \to i) \to i\}$ be a signature with the constant h of second-order type. In order to incorporate this new constant into an inference rule, we actually need two rules: one to infer a term with h as its head and one to infer a term of an arrow type (here, $i \to i$). If Γ is a signature that contains Σ', then the two new inference rules are simply

$$\dfrac{\Gamma\ \vdash\ U : i \to i}{\Gamma\ \vdash\ h\ U : i} \qquad\qquad \dfrac{\Gamma,\ x : i\ \vdash\ V : i}{\Gamma\ \vdash\ \lambda x.V : i \to i}.$$

Here, x is not in Γ. The following is a proof that the term $f\ (h\ (\lambda x(g\ x\ (f\ x))))$ is correctly formed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma',\ x : i\ \vdash\ x : i}}{\Sigma',\ x : i\ \vdash\ x : i} \qquad \dfrac{\dfrac{}{\Sigma',\ x : i\ \vdash\ x : i}}{\Sigma',\ x : i\ \vdash\ f\ x : i}}{\Sigma',\ x : i\ \vdash\ g\ x\ (f\ x) : i}}{\Sigma'\ \vdash\ \lambda x(g\ x\ (f\ x)) : i \to i}}{\Sigma'\ \vdash\ h\ (\lambda x(g\ x\ (f\ x))) : i}}{\Sigma'\ \vdash\ f\ (h\ (\lambda x(g\ x\ (f\ x)))) : i}$$

(Also replace Σ by Γ in the four rules for a, b, f, and g.) Notice that now, the signatures do not remain constant within proofs: as one moves up through such a proof, signatures can get larger. This suggests that a good notion of bound variable is essentially "scoped constant": it acts like a constant that is not visible from the top of the term, but may become visible when a descent is made through the abstraction. Thus, we state the first of two principles that are needed to support our notion of abstract syntax.

Principle 1. Recursion through syntax containing bound variables requires signatures (contexts) to be dynamically augmented.

The second principle supporting our notion of abstract syntax is rather obvious but produces serious problems for integrating into a programming language.

Principle 2. The equality of syntax should be (at least) α-conversion.

If the equations of α-conversion are assumed then terms are not freely generated and simple destructuring is not a sensible operation. For example, the two terms $\lambda x(fxx)$ and $\lambda y(fyy)$ denote the same syntactic object. If, however, λ-abstraction is treated as

a two place constructor, then these equal terms can be decomposed into unequal parts: that is, into x and y and into fxx and fyy.

An approach to solving this problem is to try to decompose syntax using unification modulo α-conversion. For example, consider the following signature over two primitive types i (representing object-level terms) and b (representing object-level formulas):

$$\begin{array}{lll} \forall : (i \to b) \to b & \wedge : b \to b \to b & \supset : b \to b \to b \\ r : i \to b & s : i \to b & t : b. \end{array}$$

Consider attempting to decompose formula

$$\forall \lambda y((ry \supset sy) \wedge t)$$

by unifying it with the formula $\forall \lambda x(P \wedge Q)$, where P and Q are free variables. This pair has no unifiers (modulo α-conversion) since no substitution instance for P will be able to bind the variable x: we are assuming that substitution at the meta-level is the correct declarative substitution that avoids bound variable capture. This example illustrates that unification using purely α-conversion is not able to cope with decomposing syntax involving a bound variable. If we change this example by attempting to match the same formula with $\forall \lambda x(Px \wedge Q)$ we now find that there is exactly one unifier (up to α-conversion), namely,

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

This substitution is a unifier, however, when α *and* β-conversions are assumed since after substituting for P and Q, the resulting term $\forall \lambda x([\lambda w(rw \supset sw)x] \wedge t)$ requires a β-reduction and an α-conversion before it is equal to $\forall \lambda y((ry \supset sy) \wedge t)$.

For some additional matching examples of this kind, consider matching the following pair of open terms (free variables are capital letters) with closed λ-term over the signature Σ.

$$\begin{array}{ll} (1)\ \lambda x \lambda y(f\ (H\ x)) & \lambda u \lambda v(f\ (f\ u)) \\ (2)\ \lambda x \lambda y(f\ (H\ x)) & \lambda u \lambda v(f\ (f\ v)) \\ (3)\ \lambda x \lambda y(g\ (H\ y\ x)\ (f\ (L\ x))) & \lambda u \lambda v(g\ u\ (f\ u)) \\ (4)\ \lambda x \lambda y(g\ (H\ x)\ (L\ x)) & \lambda u \lambda v(g\ (g\ a\ u)\ (g\ u\ u)) \end{array}$$

The second pair cannot be matched for reasons similar to those described above. The other three cases yield unique matches, assuming α and η-conversion.

$$\begin{array}{lll} (1) & H \mapsto \lambda w(f\ w) & \\ (3) & H \mapsto \lambda y \lambda x.x & L \mapsto \lambda x.x \\ (4) & H \mapsto \lambda x(g\ a\ x) & L \mapsto \lambda x(g\ x\ x) \end{array}$$

All of these examples use a very weak form of β-conversion. In particular, they continue to work if β-conversion is replaced by β_0 conversion, which is defined by the equation $(\lambda x.B)x = B$.

In the next section we present a meta-logic $\mathcal{M}$ that supports both of the principles of abstract syntax that we have described above. The language has as its equality theory α, β, and η-conversion for the simply typed λ-terms. It is possible to significantly weaken the logic $\mathcal{M}$ to a logic called L_λ where the equality theory only needs to be a restricted

form of α, β_0, and η-conversion. This equality theory is weak enough so that unification in it is decidable and most general unifiers exist when unifiers exist. It is also strong enough to support the two principles of abstract syntax presented above. The logic L_λ is describe in the papers [15, 16]. We shall not be concerned with it further here.

Abstract syntax is characterized in Figure 3: in the rest of this paper, we shall discuss the logic $\mathcal{M}$ and how it supports this notion of syntax. The paper [14] describes an approach to incorporating abstract syntax into the ML programming language [19]. What we are calling abstract syntax in this paper has also been called "higher-order abstract syntax" in [24].

Implementation	α-equivalence classes of $\beta\eta$-normal λ-terms of simple types
Access	$\beta\eta$-unification or a restriction of $\beta_0\eta$-unification (as in L_λ)
Good points	1. Bound variable names are inaccessible so many technical problems regarding them disappear. 2. Substitution is easy to support for every data structure containing abstracted variables. 3. Semantics is provided by proof theory, logical relations, and Kripke models.
Bad points	1. Requires higher-level support: dynamic contexts, extensions to first-order unification, and a richer notion of equality. 2. No robust, well-defined, and generally available programming language supports this notion of syntax (yet).

Figure 3: Characteristics of abstract syntax.

3. A Logic programming language that incorporates abstract syntax

Let S be a set of primitive types. Type expressions are all first-order expressions built from primitive types and the infix, function type constructor $\rightarrow$. This constructor associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Let $\mathcal{S}$ be a finite set of predicate symbols that are sorted using expressions of the form $\langle \tau_1, \ldots, \tau_n \rangle$ for $n \geq 0$, where $\tau_1, \ldots, \tau_n$ are types. Using a primitive type for propositions, say o as in [2], then the sort for predicates could be considered as a type of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow o$. We shall not, however, give predicates functional types: the expression $\langle \tau_1, \ldots, \tau_n \rangle$ is not a type expression.

Signatures are sets of associations of types to tokens such as

$$\Sigma = \{c_1 : \tau_1, \ldots, c_n : \tau_n, \ldots\}.$$

Signatures can be finite or infinite and are sometimes called *type assignments*. The usual functional requirement holds: if $c : \tau$ and $c : \sigma$ are members of Σ, then τ and σ

are the same type. The expression $\Sigma + c : \tau$ is legal only if c is not assigned by Σ, in which case that expression is equal to

$$\{c : \tau, c_1 : \tau_1, \ldots, c_n : \tau_n, \ldots\}.$$

A Σ-*term of type* τ is a closed λ-term all of whose constants are in Σ and which has type τ. Notice that a given λ-term may be a Σ-term at different types; for example, consider the term $\lambda x.x$. Σ-*formulas* are defined in the following fashion.

- If Q is a predicate in $\mathcal{S}$ that is sorted with $\langle \tau_1, \ldots, \tau_n \rangle$ and t_i is a Σ-term of type τ_i (for $i = 1, \ldots, n$), then $Qt_1 \cdots t_n$ is a Σ-formula. In particular, it is an *atomic* Σ-formula.
- If B and C are Σ-formulas then $B \wedge C$ and $B \supset C$ are Σ-formulas.
- If B is a $\Sigma + x : \tau$-formula then $\forall_\tau x.B$ is a Σ-formula.

Equality of terms and formulas is determined using the usual rules of $\beta\eta$-conversion.

The collection of Σ-formulas over the primitive types in S and the predicates in $\mathcal{S}$ is denoted by $\mathcal{M}(S, \mathcal{S})$, which is written as simply $\mathcal{M}$ if S and $\mathcal{S}$ can be determined from context. A proof system for $\mathcal{M}(S, \mathcal{S})$ is given by the sequent rules in Figure 4. The triple $\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B$ is a sequent if $\mathcal{P} \cup \{B\}$ is a set of Σ-formulas. We shall assume that the rules of $\beta\eta$-conversion are used whenever needed to join two inference rules together. The syntax $\mathcal{P}, B$ is short for $\mathcal{P} \cup \{B\}$. The expression $\Sigma; \mathcal{P} \vdash B$ means that the sequent $\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B$ is provable (without cut). If there is a Σ-term for all primitive types, then this proof system coincides with the more common notion of intuitionistic sequent calculus. Since $\mathcal{P}$ in the sequent $\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B$ is a set, the usual structural rules of thining, contraction, and exchange are not needed.

$$\frac{\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B \qquad \Sigma \ ; \ \mathcal{P} \ \longrightarrow \ C}{\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B \wedge C} \wedge\text{-R}$$

$$\frac{\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B \qquad \Sigma \ ; \ C, \mathcal{P} \ \longrightarrow \ A}{\Sigma \ ; \ B \supset C, \mathcal{P} \ \longrightarrow \ A} \supset\text{-L}$$

$$\frac{\Sigma \ ; \ B, C, \Delta \ \longrightarrow \ A}{\Sigma \ ; \ B \wedge C, \Delta \ \longrightarrow \ A} \wedge\text{-L} \qquad \frac{\Sigma \ ; \ B, \mathcal{P} \ \longrightarrow \ C}{\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ B \supset C} \supset\text{-R}$$

$$\frac{t \text{ is a } \Sigma\text{-term of type } \tau \qquad \Sigma \ ; \ \mathcal{P}, B[t/x] \ \longrightarrow \ C}{\Sigma \ ; \ \mathcal{P}, \forall_\tau x \ B \ \longrightarrow \ C} \forall\text{-L}$$

$$\frac{\Sigma + c{:}\tau \ ; \ \mathcal{P} \ \longrightarrow \ B[c/x]}{\Sigma \ ; \ \mathcal{P} \ \longrightarrow \ \forall_\tau x \ B} \forall\text{-R} \qquad \frac{}{\Sigma \ ; \ \mathcal{P}, B \ \longrightarrow \ B} \text{ initial}$$

Figure 4: Proof rules for $\mathcal{M}$.

There are two forms of cut rules for this sequent calculus: one works with the signature of the antecedent (called the *subst* rule) and one works with the formulas of the antecedent (called simply the *cut* rule). Both rules are displayed in Figure 5. The following theorem is know as the *cut elimination* theorem.

Theorem 3.1. *A sequent is provable with the two rules of cut and subst (Figure 5) if and only if it is provable without these two inference rules.*

The proof of this fact follows from Gentzen's original result augmented with elementary facts about the meta-theory of the $\beta\eta$-theory of simply typed λ-terms. Notice that since $\mathcal{M}$ does not admit predicate quantification, the cut-elimination result follows the usual line for first-order logics. Sequent proofs in this paper will not contain instances of cut or subst.

$$\frac{\Sigma\ ;\ \mathcal{P}, B\ \longrightarrow\ C \qquad \Sigma'\ ;\ \mathcal{P}'\ \longrightarrow\ B}{\Sigma'\ ;\ \mathcal{P}'\ \longrightarrow\ C}\ \text{cut}$$

$$\frac{\Sigma + x:\tau\ ;\ \mathcal{P}\ \longrightarrow\ B \qquad t \text{ is a } \Sigma'\text{-term of type } \tau}{\Sigma'\ ;\ \mathcal{P}'\ \longrightarrow\ B[t/x]}\ \text{subst}$$

Figure 5: Cut and subst rules for $\mathcal{M}$. Here, $\Sigma \subseteq \Sigma'$ and $\mathcal{P} \subseteq \mathcal{P}'$.

The following theorem provides an abstract justification for referring to $\mathcal{M}$ as a logic programming language. This theorem says that *goal-directed* search for proofs in $\mathcal{M}$ is a complete search method.

Theorem 3.2. *A sequent proof is uniform if every occurrence of a sequent in that proof with a non-atomic right-hand side is the conclusion of a right-introduction rule. Then, a sequent is provable in $\mathcal{M}$ if and only if it has a uniform proof.*

This is easily proved by using permutations of inference rules to convert any cut-free proof into a uniform proof. The proof of this result can be found in [18] where a richer logic than $\mathcal{M}$ is considered. A similar proof is given in [3] for a strictly first-order logic (quantification at primitive types only). Both of these papers also motivate why uniform proofs and goal-directed search are useful for characterizing logic programs.

Many examples of using $\mathcal{M}$ as a specification language and using λProlog to implement them have been considered. For example, in the area of theorem proving see the papers [3, 4, 5, 6, 22]; in the area of meta-programming of functional programs see the papers [7, 8, 9, 10, 11, 17]. The logic $\mathcal{M}$ is very similar to the logic hh^{ω} in [6]. The next section presents one of the example specifications described in [8].

4. Functional program as objects

Let $S_0 = \{i\}$ and let $\mathcal{S} = \{eval : \langle i, i\rangle\}$. Let Σ_0 be the signature containing the following constants:

$$true : i,\ false : i,\ \ 0 : i,\ 1 : i,\ 2 : i, \ldots$$
$$=: i \to i \to i,\ +: i \to i \to i,\ if : i \to i \to i \to i$$
$$app : i \to i \to i,\ abs : (i \to i) \to i,\ fix : (i \to i) \to i$$

The type i denotes object-level functional programs. Obviously, it is possible to add more constants to this signature so that Σ_0-terms of type i denote richer functional programs. The functional program

$$fun\ g\ x\ y = if\ x = y\ then\ x\ else\ g\ y\ y$$

can be represented by the Σ_0-term of type i:

$$(fix\ \lambda g(abs\ \lambda x(abs\ \lambda y(if\ (=\ x\ y)\ x\ (app\ (app\ g\ y)\ y))))).$$

Notice that abstractions in the object-level, functional program are mapped to abstractions in the meta-level term. Using this kind of encoding of functional programs, it is impossible to specify predicates in $\mathcal{M}$ that can make distinctions between two α-convertible functional programs. Thus, this encoding obeys Principle 2.

An evaluator for this object language can be described in $\mathcal{M}(S_0, \mathcal{S}_0)$ using some simple formulas: for several examples of such evaluators see [9, 11]. Here, we shall reduce this functional programming language down to it smallest, interesting core. Let Σ_1 be just the signature for application and abstraction, namely, $\{app : i \to i \to i,\ abs : (i \to i) \to i\}$. This kind of representation is derived from the mapping of the untyped λ-terms into simply typed λ-terms. In particular, the pure, untyped λ-terms modulo α-conversion can be identified with Σ_1-terms of type i modulo $\beta\eta$-conversion.

A specification of a call-by-name evaluator for Σ_1-terms in $\mathcal{M}$ is given by the following two formulas.

$$\forall_{i\to i} R\ (eval\ (abs\ R)\ (abs\ R))$$
$$\forall_{i\to i} R \forall_i M, N, V\ (eval\ M\ (abs\ R) \wedge eval\ (R\ N)\ V \supset eval\ (app\ M\ N)\ V)$$

Notice there that meta-level β-reduction, in the expression $(R\ N)$, is used to do object-level substitution. Call-by-value evaluation can also be axiomized using the following two formulas.

$$\forall_{i\to i} R\ (eval\ (abs\ R)\ (abs\ R))$$
$$\forall_{i\to i} R \forall_i M, N, V, P\ (eval\ M\ (abs\ R) \wedge eval\ N\ P \wedge eval\ (R\ P)\ V \supset eval\ (app\ M\ N)\ V)$$

Let $\mathcal{P}_1$ be the set of two formulas specifying call-by-name evaluation. If $\Sigma_1; \mathcal{P}_1 \vdash eval\ t\ s$ then we say that t evaluates to s.

It is natural to try and extend evaluation so that it can evaluate under abstractions. That is, evaluation could be extended (over the Σ_0 signature) to relate the term

$$(fix\ \lambda f\ (abs\ \lambda x\ (if\ true\ (+\ x\ 1)\ (app\ (app\ f\ x)\ x))))$$

and the term

$$(fix\ \lambda f\ (abs\ \lambda x\ (+\ x\ 1))).$$

That is, evaluation can be pushed through abstractions to reduce redexes that are not at the top-level. Over the signature Σ_1 and formulas $\mathcal{P}_1$ the evaluation predicate only relates the term $(abs\ \lambda x(app\ (abs\ (\lambda y\ y))\ x))$ to itself. It should be possible to "lift" evaluation so that the internal redex $(app\ (abs\ (\lambda y\ y))\ x)$ can be reduced. This is problematic since this internal redex is not a Σ_1-term since it has x free in it. Thus, we need to understand how to evaluate expressions over "mixed" values. This problem is solved by dynamically adding x to the signature. Let $\mathcal{C}_1$ be the set of the following two formulas.

$\forall_{i\to i} R, S\ (\forall_i x, y\ (eval\ x\ y \supset eval\ (Rx)\ (Sy)) \supset eval\ (abs\ R)\ (abs\ S))$
$\forall_i M, N, P, Q\ (eval\ M\ P \wedge eval\ N\ Q \supset eval\ (app\ M\ N)\ (app\ P\ Q))$

The first of these formulas lifts evaluation over object-level abstractions. It can be read operationally as follows: To prove the atomic formula

$$eval\ (abs\ \lambda u.t)\ (abs\ \lambda v.s),$$

try the following steps:

- Introduce two new constants, say $c : i$ and $d : i$, not mentioned in the current signature (corresponds to using $\forall$-R).
- Add the atomic formula $eval\ c\ d$ to the current program (corresponds to using $\supset$-R).
- Attempt to prove $eval\ (t[c/u])\ (s[d/v])$ in the augmented signature and program (corresponds to using β-reduction). Here, c plays the role of the bound variable name when we descend into $\lambda u.t$.

This is an illustration of how Principle 1 is supported in $\mathcal{M}$.

Notice that given $\mathcal{P}_1 \cup \mathcal{C}_1$, the proof rules for *eval* are now more nondeterministic. For every Σ_1-term t, $\Sigma_1; \mathcal{P}_1 \cup \mathcal{C}_1 \vdash\ eval\ t\ t$; that is, the extension of *eval* is reflexive. Given the same context, the atomic formula

$$eval\ (abs\ \lambda x(app\ (abs\ (\lambda y\ y))\ x))\ (abs\ \lambda x\ x)$$

is also provable. See [8] for a discussion about how such syntactic lifting of evaluation is related to the notion of "mixed" or "symbolic" evaluation.

5. A Kripke model semantics

A model theory for $\mathcal{M}(S, \mathcal{S})$ can be based on the following kind of Kripke models.

Definition 5.1. A *dependent pair* is a pair $\langle \Sigma, \mathcal{P} \rangle$ where Σ is a signature and $\mathcal{P}$ is a set of Σ-formulas. Define $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ whenever $\Sigma \subseteq \Sigma'$ and $\mathcal{P} \subseteq \mathcal{P}'$. A *Kripke model*, $[\mathcal{W}, I]$, is the specification of a set of *worlds* $\mathcal{W}$, which is a set of dependent pairs, and a function I, called an *interpretation*, that maps pairs in $\mathcal{W}$ to sets of atomic formulas. The mapping I must satisfy the two conditions:

- $I(\langle \Sigma, \mathcal{P} \rangle)$ is a set of λ-normal, atomic Σ-formulas, and
- for all $w, w' \in \mathcal{W}$ such that $w \preceq w'$, $I(w) \subseteq I(w')$ (*i.e.*, I is order preserving). ∎

Satisfiability (also called *forcing*) in a Kripke model is defined as follows.

Definition 5.2. Let $[\mathcal{W}, I]$ be a Kripke model, let $\langle \Sigma, \mathcal{P} \rangle \in \mathcal{W}$, and let B be a Σ-formula. The three place *satisfaction* relation $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$ is defined by induction on the structure of B.

- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$ if B is atomic and the λ-normal form of B is in $I(\langle \Sigma, \mathcal{P} \rangle)$.
- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B \wedge B'$ if $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$ and $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B'$.
- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B \supset B'$ if for every $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B$ then $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B'$.
- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x.B$ if for every $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and for every Σ'-terms t of type τ, the relation $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B[t/x]$ holds.

The *signature of an interpretation* I is the largest signature that is contained in all worlds of the partial order underlying I. If Σ_0 is the signature of the interpretation I and B is a Σ_0-formula, then we write $I \Vdash B$ if $I, w \Vdash B$ for all $w \in \mathcal{W}$. ∎

This notion of model is similar to that of *Kripke λ-models* described in [20].

Definition 5.3. Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair. The *canonical model* for $\langle \Sigma, \mathcal{P} \rangle$ is defined as the model with the set of worlds $\{\langle \Sigma', \mathcal{P}' \rangle \mid \langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle\}$ and where I is defined so that $I(\langle \Sigma', \mathcal{P}' \rangle)$ is the set of all λ-normal, atomic formulas A so that $\Sigma'; \mathcal{P}' \vdash A$. ∎

Lemma 5.4. *Cut-elimination (Theorem 3.1) holds for $\mathcal{M}$ if and only if the following holds: for every dependent pair $\langle \Sigma, \mathcal{P} \rangle$ and every Σ-formula B, $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$, where I is the canonical model for $\langle \Sigma, \mathcal{P} \rangle$.*

Proof. Assume first that cut-elimination holds for $\mathcal{M}$. We now prove by induction on the structure of B that $\Sigma; \mathcal{P} \vdash B$ if and only if $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$.
Case: B is atomic. The equivalence is trivial.
Case: B is $B_1 \wedge B_2$. This case is simple and immediate.
Case: B is $B_1 \supset B_2$. Assume first that $\Sigma; \mathcal{P} \vdash B_1 \supset B_2$. By Theorem 3.2, $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_2$. To show $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$, let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ be such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_1$. By the inductive hypothesis, $\Sigma'; \mathcal{P}' \vdash B_1$ and by cut-elimination, $\Sigma'; \mathcal{P}' \vdash B_2$. By induction again, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_2$. Thus, $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$. For the converse, assume $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$. Since $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_1$, the inductive hypothesis yields $I, \langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \Vdash B_1$. By the definition of satisfaction of implication we must have $I, \langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \Vdash B_2$. But by the inductive hypothesis again, $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_2$, and $\Sigma; \mathcal{P} \vdash B_1 \supset B_2$.
Case: B is $\forall_\tau x B_1$. Assume first that $\Sigma; \mathcal{P} \vdash \forall_\tau x B_1$. By Theorem 3.2, $\Sigma \cup \{d\}; \mathcal{P} \vdash B_1[d/x]$ for any constant d not in Σ. To show $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$, let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ be such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and t is a Σ'-term of type τ. By cut-elimination on signatures (the subst rule), we have $\Sigma'; \mathcal{P}' \vdash B_1[t/x]$. By induction we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_1[t/x]$. Thus, $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$. For the converse, assume $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$. Let d be a constant not a member of Σ. Since d is a $\Sigma \cup \{d\}$-term, $I, \langle \Sigma \cup \{d\}, \mathcal{P} \rangle \Vdash B_1[d/x]$ by the definition of satisfaction of universal quantification. But by the inductive hypothesis again, $\Sigma \cup \{d\}; \mathcal{P} \vdash B_1[d/x]$ and $\Sigma; \mathcal{P} \vdash \forall_\tau x B_1$.

Now assume the equivalence: for every dependent pair $\langle \Sigma, \mathcal{P} \rangle$ and every Σ-formula B, $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$, where I is the canonical model for $\langle \Sigma, \mathcal{P} \rangle$. We now show that any sequent that can be proved using occurrences of the cut and subst rules

can be proved without such rules. In particular, we show that if $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ then each of the following holds.

(1) If $\Sigma'; \mathcal{P}' \vdash B$ and $\Sigma; \mathcal{P}, B \vdash C$ then $\Sigma'; \mathcal{P}' \vdash C$.

(2) If t is a Σ'-term of type τ and $\Sigma + x : \tau; \mathcal{P} \vdash B$ then $\Sigma'; \mathcal{P}' \vdash B[t/x]$ (of course, x does not occur in Σ).

From these facts, any number of occurrences of the cut and subst rules can be eliminated from a proof containing them.

To prove (1), assume that $\Sigma'; \mathcal{P}' \vdash B$ and $\Sigma; \mathcal{P}, B \vdash C$. Thus, $\Sigma; \mathcal{P} \vdash B \supset C$. By the assumed equivalence, $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B$ and $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B \supset C$. By the definition of satisfaction for implication, $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash C$. By the assumed equivalence again, this yields $\Sigma'; \mathcal{P}' \vdash C$.

To prove (2), assume that t is a Σ'-term of type τ and that $\Sigma + x : \tau; \mathcal{P} \vdash C$. Thus, $\Sigma; \mathcal{P} \vdash \forall_\tau x.B$. By the assumed equivalence, $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x.B$. By the definition of satisfaction for universal quantification, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B[t/x]$. By the assumed equivalence again, this yields $\Sigma'; \mathcal{P}' \vdash B[t/x]$. QED

Given Theorem 3.1, this lemma provides an immediate proof of the following theorem.

Theorem 5.5. *Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair and let I be the canonical model for $\langle \Sigma, \mathcal{P} \rangle$. For all Σ-formulas B, $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$. In particular, for every $B \in \mathcal{P}$, $I \Vdash B$.*

This theorem can be sharpened using the following definition of *order* for types and for formulas.

Definition 5.6. The *order of type* τ, written $ord(\tau)$, is 0 if τ is primitive; otherwise τ is of the form $\tau_1 \to \tau_2$, in which case, the order of τ is $\max(1 + ord(\tau_1), ord(\tau_2))$.

The *order of formula* B, written $ord(B)$, is 0 if B is atomic; is $\max(ord(B_1), ord(B_2))$ if B is $B_1 \wedge B_2$; is $\max(1 + ord(B_1), ord(B_2))$ if B is $B_1 \supset B_2$; and is $\max(1 + ord(\tau), ord(B_1))$ if B is $\forall_\tau x.B_1$. ∎

Notice that if B has order 1 then B is (modulo weak equivalences) a first-order Horn clause theory.

Next we define the notion of the canonical model at a given order. Such models contain, in a sense, fewer worlds than the canonical models introduced in Definition 5.3.

Definition 5.7. A dependent pair $\langle \Sigma, \mathcal{P} \rangle$ is of order n if all the types in Σ are of order n or less and all the formulas in $\mathcal{P}$ are of order n or less. Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair of order n. The *canonical model of order n* for $\langle \Sigma, \mathcal{P} \rangle$ is $[\mathcal{W}, I]$ where $\mathcal{W}$ is the set of all dependent pairs $\langle \Sigma', \mathcal{P}' \rangle$ of order n such that (i) Σ' extends Σ with constants of order at most $n-2$, and (ii) $\mathcal{P}'$ extends $\mathcal{P}$ with Σ'-formulas of order at most $n-2$. The mapping I is defined as before, namely, for all $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$, the set $I(\langle \Sigma', \mathcal{P}' \rangle)$ contains all atomic A so that $\Sigma'; \mathcal{P}' \vdash A$. ∎

Notice that if $\langle \Sigma, \mathcal{P} \rangle$ is of order 1 then Σ is a first-order signature (all constants are of order 0 or 1) and $\mathcal{P}$ is a set of Horn clauses. The canonical model for such a dependent pair contains just one world, namely, the pair $\langle \Sigma, \mathcal{P} \rangle$.

Lemma 5.8. *Cut-elimination (Theorem 3.1) holds for $\mathcal{M}$ if and only if the following holds: Let $n \geq 1$, let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair of order n, let I be the canonical model of order n for $\langle \Sigma, \mathcal{P} \rangle$, and let B be a Σ-formula of order $n-1$. Then $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$.*

Proof. Assume first that cut-elimination holds for $\mathcal{M}$. We now prove by induction on the structure of B that $\Sigma; \mathcal{P} \vdash B$ if and only if $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$. The forward part of this equivalence is the same as in the proof of Lemma 5.4. Thus we only show details of the reverse implication for the two interesting cases.
Case: B is $B_1 \supset B_2$. Thus the order of B_1 is $n-2$ or less. Assume $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$. Since $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_1$ and $\langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \in \mathcal{W}$, the inductive hypothesis yields $I, \langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \Vdash B_1$. By the definition of satisfaction of implication we must have $I, \langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \Vdash B_2$. But by the inductive hypothesis again, $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_2$ and $\Sigma; \mathcal{P} \vdash B_1 \supset B_2$.
Case: B is $\forall_\tau x B_1$. Thus the order of τ is $n-2$ or less. Assume $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$. Let d be a constant not a member of Σ. Since d is a $\Sigma \cup \{d\}$-term and since $\langle \Sigma \cup \{d\} \rangle$ is a member of $\mathcal{W}$, then we have $I, \langle \Sigma \cup \{d\}, \mathcal{P} \rangle \Vdash B_1[d/x]$ by the definition of satisfaction of universal quantification. But by the inductive hypothesis again, we have $\Sigma \cup \{d\}; \mathcal{P} \vdash B_1[d/x]$ and $\Sigma; \mathcal{P} \vdash \forall_\tau x B_1$.

The fact that cut-elimination holds follows just as in the proof of Lemma 5.4, except here we need to use the equivalence at various different orders. QED

We shall need the following technical result.

Lemma 5.9. *Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair of order $n \geq 1$, and let $[\mathcal{W}, I]$ be the canonical model of order n for $\langle \Sigma, \mathcal{P} \rangle$. Let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$, and let $[\mathcal{W}', I']$ be the canonical model of order n for $\langle \Sigma', \mathcal{P}' \rangle$. For all Σ'-formulas B of order n, $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B$ if and only if $I' \Vdash B$.*

Proof. Simple induction on the structure of B. QED

The next theorem shows that if $\langle \Sigma, \mathcal{P} \rangle$ is a dependent pair of order n then the canonical model for $\langle \Sigma, \mathcal{P} \rangle$ of order n is, in fact, a model for $\mathcal{P}$.

Theorem 5.10. *Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair of order n and let $[\mathcal{W}, I]$ be the canonical model of order n for $\langle \Sigma, \mathcal{P} \rangle$. If B is of order n or less, then $\Sigma; \mathcal{P} \vdash B$ implies $I \Vdash B$.*

Proof. We prove the following by induction on the structure of B: for every $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$, if $\Sigma'; \mathcal{P}' \vdash B$ then $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B$.
Cases: B is atomic or B is conjunctive. These cases are simple.
Case: B is $B_1 \supset B_2$ where B_1 is of order $n-1$ or less. Let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ and let $\langle \Sigma'', \mathcal{P}'' \rangle \in \mathcal{W}$ be such that $\langle \Sigma', \mathcal{P}' \rangle \preceq \langle \Sigma'', \mathcal{P}'' \rangle$ and $I, \langle \Sigma'', \mathcal{P}'' \rangle \Vdash B_1$. Let $[\mathcal{W}'', I'']$ be the canonical model of order n for $\langle \Sigma'', \mathcal{P}'' \rangle$. By Lemma 5.9, $I'' \Vdash B_1$. By Lemma 5.8, $\Sigma''; \mathcal{P}'' \vdash B_1$. By cut-elimination, $\Sigma''; \mathcal{P}'' \vdash B_2$. By the inductive hypothesis, we have $I, \langle \Sigma'', \mathcal{P}'' \rangle \Vdash B_2$. By the definition of satisfaction, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_1 \supset B_2$.
Case: B is $\forall_\tau x.B_1$ where τ is of order $n-1$ or less. Let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ and let $\langle \Sigma'', \mathcal{P}'' \rangle \in \mathcal{W}$ be such that $\langle \Sigma', \mathcal{P}' \rangle \preceq \langle \Sigma'', \mathcal{P}'' \rangle$ and let t be a Σ''-term of type τ. By cut-elimination, $\Sigma''; \mathcal{P}'' \vdash B_1[t/x]$. By the inductive hypothesis, we have $I, \langle \Sigma'', \mathcal{P}'' \rangle \Vdash B_1[t/x]$. By the definition of satisfaction, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash \forall_\tau x.B_1$. QED

If Theorem 5.10 is specialized to just the case for order 1, it provides the familiar "minimal model" construction for first-order Horn clause theories [1]. Thus, Theorem 5.10 can be seen as a generalization of that model construction to arbitrary orders.

Notice that the converse to Theorem 5.10 is not generally true if the formula B is of order n. For example, let i be the only primitive type, let p and q be the only predicates, each of sort $\langle i \rangle$, let Σ be the signature $\{a : i\}$ and let $\mathcal{P}$ be the set of Σ-formulas

$$\{p\ a,\ \forall_i x\ (p\ x \supset q\ x)\}.$$

Then, the formula of order 1, $\forall_i x\ (q\ x \supset p\ x)$ is valid in the canonical model of order 1 for $\langle \Sigma, \mathcal{P} \rangle$ but it is not provable from Σ and $\mathcal{P}$.

Consider again the problems of evaluation under abstractions within a functional program (Section 4). The pair $\langle \Sigma_1, \mathcal{P}_1 \cup \mathcal{C}_1 \rangle$ is of order 2. The canonical model of order 2 for this pair is built by considering all those pairs $\langle \Sigma', \mathcal{P}' \rangle$ so that Σ' extends Σ_1 with some number of constants of type i and where $\mathcal{P}'$ extends $\mathcal{P}_1 \cup \mathcal{C}_1$ with some number of formulas of the form *eval* t s where t and s are Σ'-terms (conjunctions of such atoms are also allowed). The interpretation mapping is built by the usual provability construction. Thus, an alternative way to view "lifted" evaluation is: t evaluates to s if and only if the atomic formula *eval* t s is true in this model.

It is worth making the following simple observation about how canonical models can be considered minimal. We shall say that a Kripke model $\mathcal{N}$ satisfies $\langle \Sigma, \mathcal{P} \rangle$ if Σ is contained in the signature of $\mathcal{N}$ and if for every $B \in \mathcal{P}$, $\mathcal{N} \Vdash B$.

Theorem 5.11. *Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair, and let $\mathcal{K}$ be the canonical model for $\langle \Sigma, \mathcal{P} \rangle$. If $\mathcal{N}$ is a model of $\langle \Sigma, \mathcal{P} \rangle$ then $\mathcal{K} \Vdash B$ implies $\mathcal{N} \Vdash B$.*

Proof. Since $\mathcal{K} \Vdash B$ then $\Sigma; \mathcal{P} \vdash B$. By the soundness of Kripke models and the fact that $\mathcal{N}$ models $\langle \Sigma, \mathcal{P} \rangle$, we have $\mathcal{N} \Vdash B$. QED

6. Conclusions

Just as concrete syntax inadequately represents the structure of most syntactic objects, parse trees also inadequately represent the structure of syntactic objects containing bound variables. Thus a more high-level notion of syntax, called here *abstract syntax*, is desirable. A logic $\mathcal{M}$ makes it possible to specify computations that support this notion of abstract syntax. The logic programming language λProlog can be used to provide implementations of such specifications made in $\mathcal{M}$. The semantics of specifications written in this meta-logic can be described using Kripke models.

7. Acknowledgements

During the academic year 1990 – 1991, I was visiting the Universities of Edinburgh and Glasgow. At the University of Glasgow, this work has been supported by a British Science and Engineering Research Council visiting fellowship research grant. At the University of Edinburgh, this work has been supported by SERC Grant No. GR/E 78487 "The Logical Framework" and ESPRIT Basic Research Action No. 3245 "Logical Frameworks: Design, Implementation, and Experiment." At the University of Pennsylvania, this work has been supported by ONR N00014-88-K-0633 and NSF CCR-87-05596. Thanks to Eva Ma for her proofreading help.

8. References

[1] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841 – 862, 1982.

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

[3] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. Ph.D. thesis, University of Pennsylvania, August 1989.

[4] Amy Felty. A logic program for transforming sequent proofs to natural deduction proofs. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, volume 475 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1991.

[5] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61 – 80, Argonne, IL, May 1988. Springer-Verlag.

[6] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449, pages 221–235. Springer Lecture Notes in Artificial Intelligence, 1990.

[7] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic Programming Conference*, pages 942–959, Seattle, Washington, August 1988. MIT Press.

[8] John Hannan and Dale Miller. Deriving mixed evaluation from standard evaluation for a simple functional programming language. In Jan L. A. van de Snepscheut, editor, *1989 International Conference on Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 1989.

[9] John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.

[10] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332. ACM Press, 1990.

[11] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. Ph.D. thesis, University of Pennsylvania, August 1990.

[12] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[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. *Journal of Logic and Computation*, 1991.

[16] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eigth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

[17] 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.

[18] 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.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[20] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. In *Second Annual Symposium on Logic in Computer Science*, pages 303–314, Ithaca, NY, June 1987.

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

[22] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, 1988.

[23] 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.

[24] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.