Concrete Syntax

Implementation

Strings, text (arrays or lists of characters)

Access

Parsers, editors

Good points

Readable, publishable

Simple computational models for implementation (arrays, iteration)

Bad points

Contains too much information not important for many manipulations:

- white space, infix/prefix notation, key words

Important information is not represented explicitly

- recursive structure
- function–argument relationship
- term–subterm relationship

Parse Trees

Implementation

first-order terms, linked lists

Access

car/cdr/cons (Lisp)

first-order unification (Prolog) or matching (ML)

Good points

Recursive structure is immediate: recursion over syntax is easy to specify.

Term–subterm relationship is identified with tree-subtree relationship.

Algebra provides a model for many operations on syntax.

Bad points

Requires higher-level language support: pointers, linked lists, garbage collection, structure sharing.

Notions of scope, abstraction, substitution, and free and bound variables occurrences are not supported.

Immediate Notions Regarding Abstractions Are Not Immediate Using First-Order Terms

Bound variables are, like constants, global.

Thus, concepts like *free* and *bound variables occurrences* are derivative notions.

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.

Structure of First-Order Terms

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

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

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

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

Structure of λ-Terms

$$\Sigma' = \Sigma \cup \{h : (i \to i) \to i\}$$

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

provided that Γ is an extension of Σ' and x is not in Γ.

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

Designing a New Notion of Abstract Syntax

First: Recursion over terms with abstraction requires signatures (contexts) to be dynamically augmented.

Second: Equality of terms is (at least) α-conversion.

Since terms are not freely generated, simple destructuring is not a sensible operation.

$$\lambda x(fxx) \qquad\qquad \lambda y(fyy)$$

$$x \quad (fxx) \qquad\qquad y \quad (fyy)$$

This, of course, suggests unification modulo α-conversion.

Unification Modulo $\alpha\beta_0$-Conversion

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

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

This pair has no unifiers (modulo α-conversion).

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

This pair has one unifier:

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

provided a wee bit of β-conversion is permitted.

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

$$(\lambda x.B)x = B \qquad \beta_0\text{-conversion}$$

Some Matching Examples

$$a : i \qquad f : i \to i \qquad g : i \to i \to i$$

(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) $\lambda x \lambda y(g\ (H\ x)\ (L\ x))$	$\lambda u \lambda v(g\ (g\ a\ u)\ (g\ u\ u))$

(1)	$H \mapsto \lambda w(f\ w)$	
(2)	match failure	
(3)	$H \mapsto \lambda y \lambda x.x$	$L \mapsto \lambda x.x$
(4)	$H \mapsto \lambda x.(g\ a\ x)$	$L \mapsto \lambda x.(g\ x\ x)$

$$\texttt{fun F y z} = \texttt{t}$$

$$\Sigma \;\vdash \ldots \forall x \ldots \exists F \ldots \forall y \ldots \forall z \ldots [\; \ldots \; F \; y \; z = t \; \ldots \;]$$

$$F \mapsto \lambda y \lambda z . t$$

Under β_0 the λ-expression $\lambda x.B$ has a very weak functional interpretation: $\lambda x.B$ takes an increment of a signature to a term over the incremented signature.

$\beta_0\eta$-unification in which all free variables (logic variables) are applied to at most distinct λ-bound variables is called L_λ-unification.

Properties of L_λ-Unification

Such unification is decidable and most general unifiers exist if unifiers exist.

L_λ-unification appears to be the simpliest extension to first-order unification that "respects" bound variables.

L_λ-unification does not require type information to determine unifiers or the possibility of unifiers.

$\beta\eta$-unification of simply typed λ-terms (sometimes called "higher-order" unification) can be encoded directly as logic programming using only L_λ-unification.

Abstract Syntax

Implementation

α-equivalence classes of $\beta\eta$-normal λ-terms of simple types

Access

$\beta\eta$-unification, L_λ-unification (both supported by λProlog), or L_λ matching (ML$_\lambda$).

Good points

Bound variable names are inaccessible so many technical problems regarding them disappear.

Substitution is easy to support for every data structure containing abstracted variables.

Semantics is provided by proof theory, logical relations, and Kripke models.

Bad points

Requires higher-level support: dynamic contexts, extended first-order unification, and a richer notion of equality.

No robust, well-defined, and available programming language supports this notion of syntax.

Some Simple Concepts

Alphabetic Variants: Bound variables can be systematically changed.

$$\forall x(\exists y[P(x,y) \supset Q(x)]) =_\alpha \forall u(\exists v[P(u,v) \supset Q(u)])$$

Substitution: Care must be taken not to "capture" free variables. Instantiating $\forall x$ in the formula above is

$$\exists z[P(f(y),z) \supset Q(f(y))]$$

and *not*

$$\exists y[P(f(y),y) \supset Q(f(y))].$$

If the λ-calculus is available in data structures then other binding constructions can be represented using them. For example, $\forall x(\exists y[P(x,y) \supset Q(x)])$ can be encoded as

$$\forall(\lambda x(\exists(\lambda y[P(x,y) \supset Q(x)]))),$$

where $\forall$ and $\exists$ are constructors of one argument.

β and η-Conversions

The λ-term $((\lambda x.M)N)$ β-converts to the term $M[N/x]$ (the result of substituting N for free occurrences of x in M). This equation adds a great deal of complexity to the λ-calculus.

The λ-term $(\lambda x.(Mx))$ η-converts to the term M if x is not free in M. This equation adds very little complexity to the λ-calculus.