Formal Methods
1. Rewriting

Nachum Dershowitz

February 2000

Consider the following program to re-arrange a list of natural numbers in non-increasing order by inserting elements one-by-one into position:

$$\begin{array}{rcl}
max(0,x) & \rightarrow & x \\
max(x,0) & \rightarrow & x \\
max(s(x),s(y)) & \rightarrow & s(max(x,y)) \\
min(0,x) & \rightarrow & 0 \\
min(x,0) & \rightarrow & 0 \\
min(s(x),s(y)) & \rightarrow & s(min(x,y)) \\
sort(\varepsilon) & \rightarrow & \varepsilon \\
sort(x:y) & \rightarrow & insert(x,sort(y)) \\
insert(x,\varepsilon) & \rightarrow & x:\varepsilon \\
insert(x,y:z) & \rightarrow & max(x,y):insert(min(x,y),z))
\end{array}$$

Lists are represented in "cons" notation (with : and ε, as constructors) and numbers in successor (unary) notation. Each *(rewrite) rule* is an ordered pair of terms, used replace instances of l by corresponding instances of r. For example, $sort(max(0,s(0)):\varepsilon)$ *rewrites* to $sort(s(0):\varepsilon)$. We would like to know that every term constructed from *sort*, :, ε, s, and 0 leads (in zero or more rewrites) to a unique term not containing *sort*, nor the auxiliary symbols, *insert*, *max*, and *min*.

Rewrite systems (sets of rewrite rules) can be used to "interpret" other programming languages: The state of an all-powerful two-counter machine can be represented as a pair $\langle x,\ y\rangle$. The semantics of its instruction set can be defined by the following rules for an interpreter:

$$
\begin{array}{rcl}
\mathit{eval}(\mathbf{zap0}\ , \langle x,\ y\rangle) & \to & \langle 0,\ y\rangle \\
\mathit{eval}(\mathbf{zap1}\ , \langle x,\ y\rangle) & \to & \langle x,\ 0\rangle \\
\mathit{eval}(\mathbf{inc0}, \langle x,\ y\rangle) & \to & \langle s(x),\ y\rangle \\
\mathit{eval}(\mathbf{inc1}, \langle x,\ y\rangle) & \to & \langle x,\ s(y)\rangle \\
\mathit{eval}(\mathbf{dec0}, \langle 0,\ y\rangle) & \to & \langle 0,\ y\rangle \\
\mathit{eval}(\mathbf{dec0}, \langle s(x),\ y\rangle) & \to & \langle x,\ y\rangle \\
\mathit{eval}(\mathbf{dec1}, \langle x,\ 0\rangle) & \to & \langle x,\ 0\rangle \\
\mathit{eval}(\mathbf{dec1}, \langle x,\ s(y)\rangle) & \to & \langle x,\ y\rangle \\
\mathit{eval}(\mathbf{ifpos0}\ p, \langle 0,\ y\rangle) & \to & \langle 0,\ y\rangle \\
\mathit{eval}(\mathbf{ifpos0}\ p, \langle s(x),\ y\rangle) & \to & \mathit{eval}(p, \langle s(x),\ y\rangle) \\
\mathit{eval}(\mathbf{ifpos1}\ p, \langle x,\ 0\rangle) & \to & \langle x,\ 0\rangle \\
\mathit{eval}(\mathbf{ifpos1}\ p, \langle x,\ s(y)\rangle) & \to & \mathit{eval}(p, \langle x,\ s(y)\rangle) \\
\mathbf{whilepos0}\ p & \to & \mathbf{ifpos0}\ (p;\ \mathbf{whilepos0}\ p) \\
\mathbf{whilepos1}\ p & \to & \mathbf{ifpos1}\ (p;\ \mathbf{whilepos1}\ p) \\
\mathit{eval}((p;\ q), u) & \to & \mathit{eval}(q, \mathit{eval}(p, u))
\end{array}
$$

The penultimate rule, for example, can clearly be applied *ad infinitum.* A specific strategy of rule application is needed to guarantee that a normal form is obtained whenever one exists.

In general, rules are usually applied nondeterministically, since, in general, more than one rule can be applied, and any one rule may apply at more than one position within a term.

For any binary relation $\to$, we use $s \leftrightarrow t$ to mean $s \to t$ or $t \to s$. We say s *derives* t and write $s \to^* t$ if $s \to \cdots \to t$ in zero or more steps. We say that s and t are *convertible*, symbolized $s \leftrightarrow^* t$, if $s \leftrightarrow \cdots \leftrightarrow t$ in zero or more steps, and that s and t are *joinable*, if s and t derive the same term. A *normal form* is an element s for which there is no t such that $s \to t$. We write $s \to^! t$ if s derives the normal form t.

Definition 1 (Church-Rosser Property) *A binary relation is* Church-Rosser *if elements are joinable whenever they are convertible.*

Definition 2 (Confluence) *A binary relation is* confluent *if elements are joinable whenever they are derivable from the same term.*

Lemma 1 *The Church-Rosser and confluence properties are equivalent.*

Proof The proof is by induction on the number of "peaks" in the conversion. □

Definition 3 *A binary relation is* locally confluent (weakly confluent) *if for all elements r, s, and t, if $r \to s$ and $r \to t$ then $s \downarrow t$.*

Definition 4 (Termination) *A binary relation is* terminating *if there are no infinite derivations $t_1 \to t_2 \to \cdots$.*

Termination is an undecidable property of rewrite systems.

Theorem 1 (Newman's Lemma) *If a binary relation is locally confluent and terminating, then it is confluent.*

Proof The proof is by well-founded induction. □

The normalizability relation $\to^!$ for uniquely-normalizing systems defines a function.

Definition 5 (Orthogonality) *A rewrite system is* orthogonal *if*

- *no variable appears on the right side of a rule but not on the left;*
- *no variable appears more than once on any left side;*
- *no left side unifies with a (renamed) non-variable subterm of any other left-hand side or with a (renamed) proper subterm of itself.*

Combinatory Logic is a prime example of a (nonterminating) orthogonal system:

$$\begin{aligned} I \circ x &\to x \\ (K \circ x) \circ y &\to x \\ ((S \circ x) \circ y) \circ z &\to (x \circ z) \circ (y \circ z) \end{aligned}$$

This system can be used to implement any recursive function. The combinators K and S were dubbed "kestrel" and "starling" by Smullyan; I is the identity combinator; $\circ$ is composition.

An example of a non-orthogonal non-confluent (non-terminating) system is:

$$\begin{aligned} f(x,x) &\to 0 \\ f(x,s(x)) &\to 1 \\ a &\to s(a) \end{aligned}$$

Definition 6 *A binary relation is* strongly confluent *if for all elements r, s, and t, if $r \to s$ and $r \to t$ then either s and t are identical or $s \to t$ or $t \to s$ or there is a term u such that $s \to u$ and $t \to u$.*

Lemma 2 *Every strongly confluent binary relation is confluent.*

Proof Straightforward. □

The importance of orthogonal systems stems from the following:

Theorem 2 *Every orthogonal system is confluent.*

In particular, the two-counter interpreter is confluent.

Proof The idea is to show that a parallel rewriting relation associated with the system R is strongly confluent. Parallel rewriting is rewriting at one or more disjoint positions at the same time. We need to consider parallel rewriting, because if s rewrites to t_1 and s rewrites to t_2, then a subterm of s may appear many times in t_1 or t_2, and all of these occurrences may have to be rewritten in parallel to find a t to which both t_1 and t_2 rewrite in one (parallel) step. For example, if s is $f(a)$ and R is $\{a \to b, f(x) \to g(f(x), f(x))\}$ then $f(a)$ rewrites to both $f(b)$ and $g(f(a), f(a))$. Both of these terms parallel rewrite to $g(f(b), f(b))$, and this requires parallel rewriting of the two occurrences of a in $g(f(a), f(a))$. □

Definition 7 (Outermost Rewriting) *A rewriting step $s \to t$ is* outermost *if no rule applies at a symbol closer to the root symbol (in the tree representation of terms).*

Theorem 3 *For any orthogonal system, if no outermost step is perpetually ignored, the normal form—if there is one—will be reached.*

Outermost rewriting is used to compute normal forms in Combinatory Logic.

In this way, orthogonal systems provide a simple, pattern-directed (first-order) functional programming language, in which the orthogonal conditional operator

$$\begin{aligned} \mathit{if}(\mathbf{true}, x, y) &\to x \\ \mathit{if}(\mathbf{false}, x, y) &\to y \end{aligned}$$

can also conveniently be incorporated.

Formal Methods
2. Lambda Calculus

Nachum Dershowitz

March 2000

The lambda calculus was designed as a general theory of (comptable) functions.

Lambda expressions are constructed from formal parameters, applications (denoted by juxtaposition) AB, where A and B are lambda expressions, and function abstractions of the form $\lambda v.A$ where v is a formal parameter and A is a lambda expression in which occurrences of v are *bound*.

The λ-calculus consists of the axioms of equality plus the following schema:

$$(\lambda v.A[v])B \quad = \quad A[B]$$

Here A and B are arbitrary λ expressions and $A[B]$ is A with each free occurrence of the parameter v replaced by B. More formally, one can define substitutions, like $\{v \mapsto B\}$ and their application to expressions, $A\{v \mapsto B\}$, replacing all free occurences of v in A with B.

As a rewrite rule, we use beta-reduction:

$$\beta: \quad (\lambda v.A[v])B \quad \to \quad A[B]$$

Theorem 1 *Beta-reduction is Church-Rosser.*

The reason is basically the same as for orthogonal systems (think of v as a constant symbol, B as a variable, and both A and $A[B]$ as schemata). Thus, $A = B$ is a theorem of the lambda calculus iff $A \downarrow B$. But β is not terminating, so not all lambda expressions have normal forms.

Theorem 2 *The lambda calculus is* consistent *in the sense that one cannot prove every closed equation.*

A *redex* is a subterm at which a rewrite rule applies (β in our case).

Theorem 3 (Leftmost Normalization) *The normal form of any expression having one can be computed by repeated application of β to the* leftmost *redex.*

Proof Suppose $A \to^{\ell} B$ at the leftmost redex, and $A \to C$ at an arbitrary redex. Then, $C \to^{\ell} D$ for some D such that $B \to^{||} D$, where $\to^{||}$ signifies parallel reduction. It follows that were there an infinite leftmost computation, then there could not be any normalizing computation. □

Functions of multiple arguments can be "Curried" so that only lambda abstraction with one formal parameter are needed. Hence, we will use $\lambda x_1 \ldots x_n.A$ as an abbreviation for $\lambda x_1.(\cdots(\lambda x_n.A)\cdots)$.

Lambda expressions and beta-reduction provide a (Turing-) complete model of computation. In particular, (partial) recursive functions over the natural numbers can be simulated by lambda expressions. Arithmetic and logical operations are simulated as in the following table:

$\mathbf{T}$	$\lambda uv.u$
$\mathbf{F}$	$\lambda uv.v$
if x **then** y **else** z	$(xy)z$
$\mathbf{cons}(y,z)$	$\lambda u.(\mathbf{if}\ u\ \mathbf{then}\ y\ \mathbf{else}\ z)$
$\mathbf{car}(x)$	$x\mathbf{T}$
$\mathbf{cdr}(x)$	$x\mathbf{F}$
0	$\lambda v.v$
$x+1$	$\mathbf{cons}(\mathbf{F},x)$
$x=0$	$x\mathbf{T}$

Recursion is effected by the following mechanism: Suppose

$$f(x) \stackrel{!}{=} A[f]$$

where A is the **body** of the definition, containing recursive calls to f. The expression

$$(\lambda v.(A[v](vv))(\lambda v.(A[v](vv))$$

computes f.

The lambda calculus as described above does not fully capture the notion of equality of functions. For one thing, the names of parameters are immaterial: $\lambda u.u$ is the same identity function as $\lambda v.v$. Considering expressions equal

if they are teh same except for parameter renaming is called α-conversion. A more serious deficiency is that $\lambda v.Av$ and A are not convertible, though for all x one has

$$Ax \downarrow (\lambda v.Av)x$$

In general, one may want to infer equality by extensionality:

$$\frac{\forall x.Ax = Bx}{A = B}$$

Definition 1 *A lambda expression of the form* $\lambda x_1 \ldots x_n.(x(\cdots(A_1A_2)\cdots A_k))$ *is called a* head normal form.

Head normal forms are not unique.

Formal Methods
5. Type Inference

Nachum Dershowitz

April 2000

Raw typed terms are like lambda terms, but abstractions give the type of the bound parameter: $\lambda v : \tau.A$ where τ is a type expression.

Given a closed raw typed term M, one can compute its type by first annotating each occurrence of a parameter v with its assigned type v^τ, and then applying the following *typing rules*:

$$\frac{M^{\sigma\to\tau} \quad N^\sigma}{(MN)^\tau} \qquad \frac{M^\tau}{(\lambda v : \sigma.M)^{\sigma\to\tau}}$$

The *type inference problem* is to determine all possible types (maybe none) of a type-free term M. For example, $\lambda x.x$ can be well typed. For every type σ, term $\lambda x : \sigma.x$ has type $\sigma \to \sigma$. Hence, $t \to t$ is the most general type of $\lambda x.x$, where t is a *type variable*. The expression $\lambda x.\lambda y.x$ has most general type $s(\to t \to s)$. On the other hand, $\lambda x.xx$ cannot be well typed. What is the type of $(\lambda x.xx)(\lambda y.y)$?

There is an algorithm which solves this problem, and returns the most general type for any term that can be well typed. All one needs to do is to consistently annotate each parameter with a (different) type variable and use the above inference rules as a *logic program*. "Consistenly" means that

1. The same type variable is assigned to all free occurrences of a parameter v.

2. Different free variables are assigned distinct type variables.

3. All free occurrences of v in the scope of $\lambda v.$ are assigned the same type variable.

In essence we solve a system of equations constructed according to the following rules:

1. Each application $(M^{\alpha}N^{\beta})^{\gamma}$ gives an equation $\alpha = (\beta \to \gamma)$.
2. Each abstraction $(\lambda x : \alpha.M^{\beta})^{\gamma}$ gives $\gamma = (\alpha \to \beta)$.

For example,

$$M = \lambda x^{\alpha}.(\lambda y^{\beta}.(\lambda z^{\gamma}.((x^{\alpha}y^{\beta})^{\delta}z^{\gamma})^{\epsilon})^{\tau_1})^{\tau_2})^{\tau_3}$$

gives

$$\begin{gathered}\alpha = \beta \to \delta \\ \delta = \gamma \to \epsilon \\ \tau_1 = \gamma \to \epsilon \\ \tau_2 = \beta \to \tau_1 \\ \tau_3 = \alpha \to \tau_2\end{gathered}$$

which have the solution:

$$\begin{gathered}\alpha = \beta \to \gamma \to \epsilon \\ \tau_3 = (\beta \to (\gamma \to \epsilon)) \to \beta \to \gamma \to \epsilon.\end{gathered}$$

Logic programming (but not Prolog!) employs unification to solve equations according to the following rules:

$$\frac{E,\ s_1 = t_1,\ \ldots,\ s_n = t_n}{E,\ f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)} \qquad \frac{E}{E,\ t = t} \qquad \frac{E\{x \mapsto t\},\ x \mapsto t}{E,\ x = t}\text{var } x \notin t$$

Theorem 1 *Beta reduction can only increase the possible types.*

Formal Methods
3. Typed Lambda Calculus

Nachum Dershowitz

March 2000

Most programming languages distinguish between values of different types. There are basic types, plus composite types, like records, arrays, and functions.

The lambda calculus allows one to write terms like $\lambda v.vv$ in which it is unclear what v can be, since it is used both as a function and as its own argument. This ambiguity allows one to compose non-terminating lambda expressions.

Let Λ be the set of lambda expressions and let $\Lambda^\top$ be the subset of terminating (strongly normalizing) lambda expressions.

Types are sets of lambda expressions, i.e. subsets of Λ. We write $f : \tau$ to mean $f \in \tau$.

Let Δ be the *basic* type. If σ, τ are types, then so is their arrow type, $\sigma \to \tau$.

For every type τ there is a supply of distinct variables $v : \tau$. Expressions are *typed* according to the following rules:

$$\frac{f : \sigma \to \tau \quad t : \sigma}{ft : \tau} \qquad \frac{v : \sigma \quad t : \tau}{\lambda v.t : \sigma \to \tau}$$

We will show that all well-typed expressions are terminating. This is not easy. We will first define a notion of *computability*, and then proceed in two steps:

- Show that every computable expression is terminating by induction on the type structure.

- Show that every expression is computable by induction on the term structure.

Let $\sigma^\top$ be the terminating subset of terms in σ.

Definition 1 *Denote the* computable *subset of* τ *by* τ^C.

$$\begin{array}{lcl} \Delta^C & = & \Delta^\top \\ [\sigma \to \tau]^C & = & \{f : \sigma \to \tau | f(\sigma^C) \subseteq \tau^C\} \end{array}$$

where $f(S) = \{fs | s \in S\}$.

Lemma 1

1. $(\cdots(v\Lambda^\top)\cdots\Lambda^\top) \subseteq \Lambda^C$
2. $\Lambda^C \subseteq \Lambda^\top$

Lemma 2

1. *If* v *does not occur free in* s *and* $s, t \in \Lambda^C$, *then* $(\lambda v.s)t \in \Lambda^C$.
2. *If* v *occurs free in* s *and* $s\{v \mapsto t\} \in \Lambda^C$, *then* $(\lambda v.s)t \in \Lambda^C$.

Lemma 3 $s\gamma \in \Lambda^C$ *for substitution* $\gamma = v_i \mapsto t_{ii}$ *if* $t_1, \ldots, t_n \in \Lambda^C$.

Proof By induction on the structure of the expression s.

If s is a variable v, then $s\gamma$ is either v (in which case it is in Λ^C by Lemma 1.1) or it is some t_i (given).

If s is an application fa then $s\gamma$ is the application $(f\gamma)(a\gamma)$, both parts of which are computable (by induction). Then $(f\gamma)(a\gamma) \in \Lambda^C$ by the definition of computability.

Suppose s is an abstraction $\lambda v.t[v] : \sigma \to \tau$. For every $a \in \sigma^C$, we have $t\gamma\{v \mapsto a\} \in \tau^C$ by induction. Thus, by the previous lemma, for every $a \in \sigma^C$, we have $(\lambda v.t)a \in \tau^C$. By definition, $s\gamma \in \Lambda^C$. □

Theorem 1 *Every typed term is terminating.*