Relations:

their uses in programming and computational specifications

Dale Miller

INRIA - Saclay & LIX, Ecole Polytechnique

Outline

1. Logic and computation
2. Comparing programming with functions and relations
3. Examples of logic programs
4. Concluding observations

By way of introduction

Currently DR at INRIA-Saclay, team leader of Parsifal.

Former: positions at the University of Pennsylvania, Penn State University, and Ecole Polytechnique.

Research: symbolic logic and proof theory applied to computational logic: automated deduction, logic programming, model checking.

Influenced by: Church then Gentzen then Girard.

Roles of Logic in the Specification of Computation

Two approaches to using logic to specify computations.

Computation-as-model: Computations are mathematical structures: nodes, graphs, and state (for example, Turing machines, etc). Logic is used externally to make statements *about* those structures. E.g. Hoare triples, Hennessy-Milner logic.

Computation-as-deduction: Bits of logic are used directly.

Functional programming. Programs are proof; computation is *proof normalization* (λ-conversion, cut-elimination).

Logic programming. Programs are theories; computation is *proof search* (for cut-free sequent proofs).

Computing with functions

The lambda-operator will be written as an infix, backslash: $\lambda x.x$ as `(x\x)`, $\lambda x.(xx)$ as `(x\ x x)`, $\lambda x \lambda y \lambda z.(xz)(yx)$ as `(x\y\z\ (x z) (y z))`, etc.

Using Church numerals: `(plus 2 2)` β-reduces to `4`.

```
(n\m\f\x\ (n f) ((m f) x)) (f\x\ f (f x)) (h\u\ h (h u))
(m\f\x\ ((f\x\ f (f x)) f) ((m f) x)) (h\u\ h (h u))
(m\f\x\ (x\ f (f x)) ((m f) x)) (h\u\ h (h u))
(f\x\ (x\ f (f x)) (((h\u\ h (h u)) f) x))
(f\x\ (x\ f (f x)) ((u\ f (f u)) x))
(f\x\ (x\ f (f x)) (f (f x)))
(f\x\ f (f (f (f x))))
```

Emphasis on normal forms, confluence, termination, etc.

Computational dynamics are modelled by changes to the expression.

Computing with relations

Computation with relations is different.

```
?- append (1 :: 2 :: nil) (3 :: nil) L.
L = 1 :: 2 :: 3 :: nil ? y
no
?- append L K (1 :: 2 :: 3 :: nil).
K = 1 :: 2 :: 3 :: nil
L = nil ? y
K = 2 :: 3 :: nil
L = 1 :: nil ? y
K = 3 :: nil
L = 1 :: 2 :: nil ? y
K = nil
L = 1 :: 2 :: 3 :: nil ? y
no
?-
```

Simple List Operations

```
kind list      type -> type.
type nil       list A.
type ::        A -> list A -> list A.

type memb      A -> list A -> o.
type append    list A -> list A -> list A -> o.

memb X (X :: L).
memb X (Y :: L) :- memb X L.

append nil K K.
append (X :: L) K (X :: M) :- append L K M.
```

The gap between FP and LP is robust

We know enough to describe FP and LP succiently via proof theory.

The many recent advances in applying proof theory to computational logic do not help to bring them closer.

Higher-orders: both paradigms can use these but they yield different expressiveness.

Linear logic: Both pardigms get more programs: FP gets proof nets, geometry of interaction; LP gets two implications, two conjunctions, etc.

Game theory: In FP, games are used to model function-argument communication. In LP, proofs are winning strategies for winning arguments (dialog games).

Of course, there are trivial mergings, and as well as implementations of one in the other.

A Series of Logic Programming Languages

Horn clauses (Prolog) Unrestricted use of $\{\&, \top\}$ but $\forall, \Rightarrow$ are restricted to the top-level only. For example, $\forall \bar{x}(G \Rightarrow A)$.

Hereditary Harrop formulas (λProlog) Unrestricted use of $\{\forall, \Rightarrow, \&, \top\}$. For example, $\forall x((Bx \,\&\, \forall y(Cxy \Rightarrow Dy)) \Rightarrow Dx)$.

Lolli (a linear refinement of λProlog) Unrestricted use of $\{\forall, \multimap, \Rightarrow, \&, \top\}$. For example, $\forall x((Bx \multimap \forall y(Cxy \Rightarrow Dy)) \Rightarrow Dx)$.

Linear Objects (LO: Andreoli and Pareschi) Unrestricted use of $\{\&, \top, \parr, \bot\}$ with only top-level occurrences of $\forall, \multimap$.

$$\forall \bar{x}(G \multimap A_1 \parr \cdots \parr A_n)(n \geq 1)$$

Forum Unrestricted use of $\{\forall, \multimap, \Rightarrow, \&, \top, \parr, \bot\}$. For example,

$$\forall x(Bx \multimap \forall y(Cxy \Rightarrow Dy) \Rightarrow Dx \parr Bx)$$

Cut-free proofs are computation traces

Cut-free proofs (lemma-free proofs) of mathematically interesting theorems do not exist in nature.

Cut-free proofs in the study of logic programs are used as a more declarative and principled version of computation traces, similar to the way one defines them with Turing machines.

Since computation traces are proofs, one hopes the results in proof theory can be applied directly to the study of computation.

Expressive strength: changes in sequents

Consider a cut-free proof of the sequent $\Gamma \longrightarrow A$. Let $\Gamma' \longrightarrow A'$ be a sequent somewhere in this proof. (A and A' are atoms.)

With **Horn clauses**, we have $\Gamma = \Gamma'$. That is, context is global and nothing is hidden.

The dynamics of computation is embedded in the changing of atoms from A and A'; that is, within *non-logical contexts*. Hence, computation is hidden away from the most basic logical principles (modus ponens, cut-elimination, etc).

With **hereditary Harrop formulas**, context can change: $\Gamma \subseteq \Gamma'$. Supports modular programming and abstract data-types.

With **linear logic**, Γ and Γ' can be related via much richer fashions (via multiset rewriting).

Where is the "relation"?

Given a fixed, Horn clause program $\mathcal{P}$, we attempt to prove

$$\longrightarrow p(t_1, \ldots, t_n)$$

One can confuse the relationship denoted by p and by the provability of the sequent (from $\mathcal{P}$).

In richer logic programming settings, the sequent is

$$\Psi; \Gamma \longrightarrow p(t_1, \ldots, t_n), \Delta$$

and the computed relation is more "situated" with other items.

A scheme for reasoning about computation

Let $\vdash$ be a provability in a sequent system to encode computation (e.g., intuitionistic or linear logic).

Let $\vdash^+$ be a stronger system with, for example, rules for induction/coinduction, etc.

Assume that $\vdash \subseteq \vdash^+$ and that cut-elimination holds for $\vdash$ and $\vdash^+$.

One scheme for reasoning is the following:

	$\vdash$	C	A computation is witnessed.
C	$\vdash^+$	D	An inference about computation.
	$\vdash^+$	D	By cut.
	$\vdash$	D	By inspection of a cut-free proof.

Thus the implication $C \supset D$ in the stronger system ($\vdash^+$) can be used to carry computations to computations.

Reversing a list in Prolog

Move one item from top of one list to the top of the other list.

```
(1::2::3::nil)     nil.
(2::3::nil)   (1::nil).
(3::nil)   (2::1::nil).
nil     (3::2::1::nil).
```

This computation can be encoded as the program

```
rv nil (3::2::1::nil).
rv (X::L) M :- rv L (X::M).
```

and the query

```
rv (1::2::3::nil) nil.
```

Not really a good program since it is written for one list only.

Notice that `reverse` is symmetric. Proof: Flip both rows and columns!

A better specification

$$\forall L, K\ [$$
$$(\forall rv\ ((\forall M, N, X(rv\ N\ (X :: M) \multimap rv\ (X :: N)\ M)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil))$$
$$\multimap reverse\ L\ K\ ]$$

The base case is assumed linearly! An attempt to prove

$$reverse\ (1 :: 2 :: 3 :: nil)\ (3 :: 2 :: 1 :: nil)$$

results in the introduction of a new predicate rv and the attempt to prove that from the two clauses

$$rv\ nil\ (3 :: 2 :: 1 :: nil)$$

$$!\forall M, N, X(rv\ N\ (X :: M) \multimap rv\ (X :: N)\ M)$$

it follows that

$$rv\ (1 :: 2 :: 3 :: nil)\ nil.$$

Reverse is symmetric

Theorem. Reverse is symmetric; that is, if $\vdash reverse\ L\ K$ then $\vdash reverse\ K\ L$.

Proof. Assume that $\vdash reverse\ L\ K$. Thus, the body of $reverse$'s clause must be provable.

$$\vdash \forall rv\ ((\forall M, N, X(rv\ N\ (X :: M) \multimap rv\ (X :: N)\ M)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil)$$

Instantiate this quantifier with the term $\lambda x \lambda y.(rv\ y\ x)^{\perp}$:

$$\vdash (\forall M, N, X((rv\ (X :: M)\ N)^{\perp} \multimap (rv\ M\ (X :: N))^{\perp}) \Rightarrow (rv\ K\ nil)^{\perp} \multimap (rv\ nil\ L)^{\perp}$$

Using the linear logic equivalence of the contrapositive rule: $p^{\perp} \multimap q^{\perp} \equiv q \multimap p$, we have

$$\vdash (\forall M, N, X(rv\ M\ (X :: N) \multimap rv\ (X :: M)\ N)) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil$$

By universal generalization over rv, we have

$$\vdash \forall rv\ ((\forall M, N, X (rv\ M\ (X :: N) \multimap rv\ (X :: M)\ N)) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil)$$

This matches the body of the reverse problem if we switch L and K. Thus, we conclude that $\vdash reverse\ K\ L$.

Higher-order logic and higher-order programming

A relation can take a relation as an argument.

Church's Simple Theory of Types [1940] provides a solid foundation for the syntax of higher-order logic using an elegant combination of λ-calculus and logic.

The early work on λProlog (higher-order Horn clauses and hereditary Harrop formulas) builds on Church's framework to explain higher-order relational programming.

Predicate abstraction in LP corresponds to function abstraction in FP.

What does function abstraction in LP correspond to in FP?
(another talk)

Some Higher-Order Programming Examples

```
type forevery   (A -> o) -> list A -> o.
forevery P nil.
forevery P (X :: L) :- P X, forevery P L.

type forsome   (A -> o) -> list A -> o.
forsome P (X :: L) :- P X.
forsome P (X :: L) :- forsome P L.

type mappred (A -> B -> o) -> list A -> list B -> o.
mappred P nil nil.
mappred P (X :: L) (Y :: K) :- P X Y, mappred P L K.

type mapfun    (A -> B) -> list A -> list B -> o.
mapfun F nil nil.
mapfun F (X :: L) ((F X) :: K) :- mapfun F L K.
```

The Mappred Program

```
type mappred (A -> B -> o) -> list A -> list B -> o.
mappred P nil nil.
mappred P (X::L1) (Y::L2) :- P X Y, mappred P L1 L2.
```

The predicate variable `P` appears both as an argument and as taking arguments. Consider the following simple clauses:

```
type age  person -> int -> o.
age bob 23.
age sue 24.
age ned 23.
```

and now consider the following query:

```
?- mappred (X\Y\ age X Y) (ned::bob::sue::nil) L.
```

The answer substitution for `L` is `(23::23::24::nil)`.

The Sublist Program

```
type sublist (A -> o) -> list A -> list A -> o.
sublist P (X::L) (X::K) :- P X, sublist P L K.
sublist P (X::L) K      :- sublist P L K.
sublist P nil nil.
```

```
type have_age   list person -> list person -> o.
have_age L K :- sublist (Z\ sigma X\ age Z X) L K.
```

```
type same_age  list person -> list person -> o.
same_age L K :- sublist (Z\ age Z A) L K.
```

Flexible Goals

```
?- P bob 23.
```

One answer to this query is the substitution `(X\Y\ age X Y)` for `P`. Many other substitutions are also valid. Let `G` be any provable closed query. The substitution `X\Y\G` for `P` is a legal answer substitution.

For example, substituting

```
X\Y\ memb 4 (3::4::5::nil)
```

for `P` is also an answer substitution.

Such queries are ill-posed.

Constraining Flexible Goals

```
type primrel, rel      (person -> o) -> o.
type mother,  wife     person -> o.
primrel mother.
primrel wife.
rel R :- primrel R.
rel (X\Y\ sigma Z\ R X Z, S Z Y) :- primrel R , primrel S.
mother jane mary.
wife john jane.
```

The query

```
?- rel R, R john mary,
```

has the unique answer substitution for `R` (namely, mother-in-law)

```
X\Y\ sigma Z\ wife X Z, mother Z Y
```

Operational semantics as inference rules

CCS and π-calculus transition system:

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin fn((y)P') \end{array}$$

Functional programming evaluation:

$$\frac{M \Downarrow \lambda x.R \quad N \Downarrow U \quad R[N/x] \Downarrow V}{(M\ N) \Downarrow V}$$

Simple typing of terms: used in functional (SML) and logic (λProlog) programming.

$$\frac{\Gamma, x{:}\tau \vdash t{:}\sigma}{\Gamma \vdash \lambda x.t{:}\tau \to \sigma} \quad x \notin fn(\Gamma)$$

Operational semantics of computation systems

Can these be seen as expressions in logic? Does proof theory, an approach to inference, have a role to play here?

Can $\dfrac{A_1 \quad \cdots \quad A_n}{A_0}$ be encoded as $\forall \bar{x}[(A_1 \wedge \ldots \wedge A_n) \supset A_0]$ $A_0 \text{ :- } A_1, \ldots, A_n.$

Particular problems:

- Ordered premises: particularly in functional programming with side-effects. But $\wedge$ is commutative.
- The status of bindings substitutions in terms must be explained.
- Side-conditions: many deal with occurrences of names and variables.

See my recent article in the *Concurrency Column* of the Bulletin of the EATCS.

Encoding intensional aspects of computation

A final point: the relational setting seems worth developing also to handle reasoning about computation.

The traditional steps to build a proof assistant:

First: Implement mathematics. Chose a classical or intuitionistic foundation: often a typed λ-calculus is picked (Coq, NuPRL)

Second: Reduce programming correctness problems to mathematics. Data structures, states, stacks, heaps, invariants, etc, all are represented as various kinds of mathematical objects. Use standard mathematical techniques (induction, primitive recursion, fixed points, well-founded orders, etc). Use denotational semantic and code everything functionally.

Relations allow a more intensional treatment

Many *intensional* aspects of computing are not served well using such extensional foundations: in particular, bindings in syntax and resources (as in linear logic).

A more direct, one-step approach to encoding computation seems possible using *relations* for the treatment of bindings and resources. Developing the foundations of such a proof assistant is active research.

Questions?