CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
Some Examples of Specifications

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

Encoding Functional Programs in λProlog

We have already seen how to use λ-tree syntax to represent types and (untyped) lambda terms

It is easy to extend this setup to obtain a framework for encoding arbitrary functional programs, e.g.

```
type  bool  ty.
type  lst   ty -> ty.

type  i          int -> tm.
type  tt, ff     tm.
type  nil, cons  tm.
type  sum        tm.
type  cond       tm -> tm -> tm -> tm.
type  fix        (tm -> tm) -> tm.
```

Other combinators, functions, data types, can also be encoded

If the types language has explicit polymorphism, this can also be encoded using λ-tree syntax

Typing and Evaluation for Functional Programs

The earlier judgements for typing and evaluation can be easily extended to accommodate the new constructs

For example, consider

```
(of  tt  bool) & (of ff bool).
of (i I) int.
of sum (int --> int --> int).
...
of  (cond C T E) A :-
     of C bool, of T A, of E A.
of (fix E) Ty :-
      pi x\ (of x Ty => of (E x) Ty).

...
eval (app (app sum E1) E2) (i I) :-
    eval E1 (i I1), eval E2 (i I2), I is I1 + I2.
eval (cond C T _) V :- eval C tt, eval T V.
eval (cond C _ E) V :- eval C ff, eval E V.
eval (fix E) V :- eval (E (fix E)) V.
```


Small-Step Evaluation via Evaluation Contexts

Lambda terms provide an elegant means for characterizing evaluation contexts in computation via repeated rewriting

```
type val, non_val, redex  tm -> o.
type reduce, eval        tm -> tm -> o.
type context             tm -> (tm -> tm) -> tm -> o.

context R (x\ x) R :- redex R.
context (cond M N P) (x\ cond (E x) N P) R :-
  non_val M, context M E R.
context (app M N) (x\ (app (E x) N)) R :-
  non_val M, context M E R.
context (app V M) (x\ (app V (E x)) R :-
  val V, non_val M, context M E R.

eval V V :- val V.
eval M V :- context M E R, reduce R N, eval (E N) V.
```

Here `non_val`, `val`, and `redex` recognize non-values, values and redexes at the root and `reduce` contracts redexes

Recognizing Tail Recursive Structure

In an expression of the form `(fix (f\ F))` we have to check that usage of `f` in `F` is suitably restricted

Assume that the signature of the language has been reified via the predicate `term`

```
type  tr, fn, trabs, headrec, trbody  tm -> o.

tr (fix M) :- pi F\ ((fn F) => (trabs (M F))).

trabs (abs R) :- pi X\ ((term X) => (trabs (R X))).
trabs R :- trbody R.

trbody (cond M N P) :- term M, trbody N, trbody P.
trbody M :- term M ; headrec M.

headrec (app M N) :- (fn M ; headrec M), term N.
```

Recursion over binding structure allows for a generalization of template matching a la Burstall and Darlington (Huet and Lang)

Binding Sensitive Analysis of Functional Programs

Some other examples in the book where λ-tree syntax is used to advantage:

- Partial evaluation, reductions under abstractions

  AUGMENT and GENERIC provide logical support for descent inside the body of an (object-language) abstraction

- Continuation-passing style transformation of programs

  An approach that uses λ-calculus equivalences to correctly transform programs to a tail recursive form

Specifying a Natural Deduction Calculus

We will think of formalizing this as a typing calculus

A sampling of the proof terms that might be used in this pursuit

```
kind  proof type.

type imp_i  (proof -> proof) -> proof.
type imp_e  form -> proof -> proof -> proof.
type or_i1,
     or_i2  proof -> proof.
type or_e   form  -> form -> proof
                  -> (proof -> proof)
                  -> (proof -> proof) -> proof.
type all_e  term  -> (term -> form) -> proof -> proof.
type all_i  (term -> proof) -> proof.
type some_e (term -> form) -> proof ->
            (term -> proof -> proof) -> proof.
type some_i term  -> proof -> proof.
```

Typically we store as much information in the proof term as is necessary to make type checking well-behaved

Specifying a Natural Deduction Calculus (Continued)

The inference rules of the calculus are then formalized as the definition of a predicate that relates proof terms and formulas

```
type  #  proof -> form -> o.     infix # 2.

(imp_i Q) # (A ==> B) :- pi p\ (p # A) => ((Q p) # B).
(imp_e A P1 P2) # B   :- (P1 # A), (P2 # (A ==> B)).
(or_i1 P) # (A !! B)  :- P # A.
(or_i2 P) # (A !! B)  :- P # B.
(or_e A B P Q1 Q2) # C :-
   (P # (A !! B)),
   (pi p1\ (p1 # A) => ((Q1 p1) # C)),
   (pi p2\ (p2 # B) => ((Q2 p2) # C)).
(all_i Q) # (all A)   :- pi y\ (Q y) # (A y).
(all_e T A P) # (A T) :- (P # (all A)).
(some_i T P) # (some A) :- P # (A T).
(some_e A P1 Q) # B :-
   (P1 # (some A)),
   pi y\ pi p\ (p # (A y)) => ((Q y p) # B).
```

Specifying versus Implementing Proof Systems

- Declarative specifications like that for the natural deduction calculus are well-suited for meta-theoretic reasoning
- Sometimes, such calculi can also be structured to provide a basis for proof search

  E.g, Dyckhoff's calculus and calculi that integrate focusing
- However typically such calculi, together with depth-first exploration, are not good for proof search
- Tactics and tacticals based approaches provide a means to "bake your own" control regime
  - Inference rules are encoded as standalone goal transformers called tactics
  - Tacticals provide a framework for combining such tactics into larger units
  - In the λProlog setting, definitions of tacticals use predicate variables

Process Expressions in the π-Calculus

A language for modelling processes that interact using names

Two important syntactic categories: *names* and *processes*

The process expressions in the finite π-calculus:

$$P ::= 0 \mid P\,'|'\,P \mid P + P \mid x(y).P \mid \bar{x}y.P \mid [x = y].P \mid \tau.P \mid (y)P$$

Here x and y represent names

The intended meaning of the various expressions

- 0 is the null process, $|$ and $+$ stand for parallel composition and choice, $\tau.P$ is a process that can evolve silently to P
- $x(y).P$ can accept a name along channel x and transform into P with y replaced by this name
- $\bar{x}y.P$ can evolve into P by outputting y along channel x
- $[x = y].P$ can become P if x and y are equal
- $(y)P$ represents the restriction of the name y to P

Representing Process Expressions in λProlog

Declarations provide the framework for an encoding

```
kind name          type.
kind proc          type.

type null          proc.
type plus, par     proc -> proc -> proc.
type in            name -> (name -> proc) -> proc.
type out, match    name -> name -> proc -> proc.
type taup          proc -> proc.
type nu            (name -> proc) -> proc.
```

Note the representation of input processes and restriction

E.g., the process $(b(z).P \mid (y)\bar{b}y.Q)$ will be represented by

```
(par (in b (z\ P')) (nu (y\ (out b y Q'))))
```

where `b` is declared to be a `name` and `P'` and `Q'` are encodings of P and Q

Transitions in the π-Calculus

The operational semantics is given by inference rules defining judgements of the form $P \xrightarrow{A} Q$

To be read as "process P evolves into Q via action A"

Note: This is what is often referred to as "small-step" semantics

There are four kinds of actions

τ	the *silent* action
$x(y)$	the (bound) input action
$\bar{x}y$	the free output action
$\bar{x}(y)$	the bound output action

The last differs from the third in that it emits a private name (bound by a restriction) on the channel x

The bound actions involve side conditions and formalizing their interaction correctly requires some care

Bound Actions and their Interaction

The rules of interest are the following

$$\frac{}{x(z).P \xrightarrow{x(w)} P\{w/z\}}\ \textsc{Input-Act},\ w \text{ not free in } (z)P$$

$$\frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}}\ \textsc{Open},\ x \neq y,\ w \text{ not free in } (y)P$$

$$\frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (w)(P'|Q')}\ \textsc{Close} \qquad \frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{Q|P \xrightarrow{\tau} (w)(Q'|P')}\ \textsc{Close}$$

Here

- The OPEN rule "opens" a scope represented by a restriction operator
- The CLOSE rule closes the corresponding scope after interaction with an input action

For example, consider the evolution of $b(z).P \,|\, (y)\bar{b}y.Q$

Encoding Transition Rules in λProlog

The key idea in capturing the interaction of bounded actions

Bounded actions will produce abstracted processes that can be combined by being fed a common name

To realize this idea, we encode transitions via *two* predicates

Specifically, we will use the following λProlog declarations

```
kind action         type.
type tau            action.
type up, dn         name -> name -> action.

type one     proc -> action -> proc -> o.
type onep    proc -> (name -> action)
                  -> (name -> proc)  -> o.
```

Some actions will yield clauses for only one predicate, some will yield clauses for both

Encoding Transition Rules in λProlog (Continued)

Using the signature, the actions $x(w)$ and $\bar{x}(w)$ will be represented by `(dn x)` and `(up x)` respectively

The (bound) name w will become an abstraction over the resulting process

The clauses for the INPUT-ACT, OPEN and CLOSE rules

```
onep (in X M) (dn X) M.
onep (nu P)   (up X) P' :-
   pi y\ one  (P y) (up X y) (P' y).
one  (par P Q) tau (nu y\ par (P' y) (Q' y)) &
one  (par Q P) tau (nu y\ par (Q' y) (P' y)) :-
            onep P (up X) P', onep Q (dn X) Q'.
```

The use of abstracted processes ensures all the side conditions are met

The clause for the CLOSE action applies these abstractions to a common name to realize the combination

Encoding Transition Rules in λProlog (Continued)

To complete the picture, we consider the encoding of some other typical rules

- The free output action will yield a clause only for `one`

$$\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}\ \textsc{Output-Act}$$

```
one  (out X Y P)  (up X Y) P.
```

- Bound input and free output actions can also interact

$$\frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, (Q[y/z])} \qquad \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{Q \,|\, P \xrightarrow{\tau} (Q[y/z]) \,|\, P}\ \textsc{Com}$$

```
one  (par P Q) tau (par S (T Y)) :-
     one  P (up X Y) S, onep Q (dn X) T.
one  (par P Q) tau (par (S Y) T) :-
     onep P (dn X) S, one  Q (up X Y) T.
```

Encoding Transition Rules in λProlog (Continued)

- The "congruence" over a choice must be reflected on both `one` and `onep`

$$\text{SUM}: \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{Q + P \xrightarrow{\alpha} P'}$$

```
one  (plus P Q) A P' :- one  P A P'; one  Q A P'.
onep (plus P Q) A P' :- onep P A P'; onep Q A P'.
```

- A similar kind of congruence applies to parallel composition

$$\text{PAR}: \frac{P \xrightarrow{\alpha} P'}{P\,|\,Q \xrightarrow{\alpha} P'\,|\,Q} \qquad \frac{P \xrightarrow{\alpha} P'}{Q\,|\,P \xrightarrow{\alpha} Q\,|\,P'}$$

```
one  (par  P Q) A (par P' Q) &
one  (par  Q P) A (par Q P') :- one P A P'.
onep (par  P Q) A (y\ par (P' y) Q) &
onep (par  Q P) A (y\ par Q (P' y)) :- onep P A P'.
```


Using the π-Calculus Specifications

- Specifications can be used to experiment with the behaviour of described systems

  λProlog allows the specifications to be animated, facilitating, for example

  - the inspection of one step transitions from processes
  - the examination of traces

  Here we are talking about the *may* behaviour of systems

- We can also consider the use of the specifications to *analyze* the behaviour of systems

  For example, showing that a process cannot make some transitions, showing similarity between processes, etc

  However, for this we also need methods for talking about the *only things* a process can do, i.e. its *must* behaviour

  To do this correctly, we will need a framework that treats the "only if" aspect of logic specifications