Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

Higher-Order Logic Application: Denotational Semantics for Functional Languages

Burkhart Wolff

Global Outline (1)

- Foundations
  - Foundational Axioms, Methodology, Historical Background, Principia Structure
  - Fixpoints and Inductive Sets
  - Well-founded Orders and Recursors
  - Arithmetic, Data-Types

Global Outline (2)

- Embeddings
  - Foundations,
    Functional Languages and Denotational Semantics
  - Imperative Languages, Refinement Calculus
  - Z and Data-Refinement,
    CSP and Process-Refinement
  - Object-oriented Languages (Java-Light . . .)

Global Outline (2)

- Embeddings
  - Foundations,
    Functional Languages and Denotational Semantics
  - Imperative Languages, Refinement Calculus
  - Z and Data-Refinement,
    CSP and Process-Refinement
  - Object-oriented Languages (Java-Light . . .)

Motivation

- Current stage of our course:
  - we have a logical framework
    for computer science
  - with set theory,
    total function recursion theory
  - proof support for: inductive sets,
    datatypes, primitive recursion definition
  - rich library

$\Rightarrow$ how can we apply this framework to specification and programming languages?

Representation Techniques for Semantics

Outline:

- Representing Languages in HOL
  - shallow
  - deep
- Foundation for Functional Programming
  - sets and relations
  - cpo's
- Deriving Operational Semantics . . .

Question:

What is the *Meaning* of a "Language"?

Syntax and Semantics

- syntax: language = set of symbols
- semantics:
  - set of denotations, the "semantic *domain*"
  - meaning function (or: *interpretation*) relating these two

Syntax and Semantics

A Language:

- set of "words" (strings)
  - ⇒ concrete syntax
  - ◦ definition techniques:
    inductive sets of strings in HOL

- set of "trees" (terms)
  - ⇒ abstract syntax
  - ◦ definition techniques:
    - ▷ abstract data types
    - ▷ constant definitions in HOL

Representation Techniques: An Example

Example: Regular Expressions

- The Language:
  - concrete syntax in a BNF-grammar:

    rex ::= char "−" char

    rex ::= char

    rex ::= "."

    rex ::= "(" rex ")"

    rex ::= "[" rex "]"

    rex ::= rex "∗"

    rex ::= rex "|" rex

    rex ::= rex rex

Example: Regular Expressions

- The Language:
  - concrete syntax: BNF-grammar as inductive definition (Version 1):

consts rex :: string set
inductive "rex"
intros

range [(x::char)] @ "−" @ [(y::char)] ∈ rex
char [(x::char)] ∈ rex
dot "." ∈ rex
par r_1 ∈ rex ⟹ "(" @ r_1 @ ")" ∈ rex
bracket r_1 ∈ rex ⟹ "[" @ r_1 @ "]" ∈ rex
star r_1 ∈ rex ⟹ r_1 @ "∗" ∈ rex
alt ⟦r_1 ∈ rex; r_2 ∈ rex⟧ ⟹ r_1 @ "|" @ r_2 ∈ rex
seq ⟦r_1 ∈ rex; r_2 ∈ rex⟧ ⟹ r_1 @ r_2 ∈ rex"

Example: Regular Expressions

- The Language:
  - Well-known problems: grammars not deterministic, . . .
  - therefore precedences, auxiliary non-terminals, . . .

rex ::= sx ["|" rx]
sx ::= tx [sx]
tx ::= ax ["*" | "+" | "?"]
ra ::= char "–" char
mx ::= ra [mx]
ax ::= char
ax ::= "."
ax ::= "(" rex ")"
ax ::= "[" mx "]"

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as data-type (Version 2):

```
datatype rex =
range char char ("_-_")
  | char char ("(_)")
  | dot (".")
  | bracket rex ("[_]")
  | star rex ("_*")
  | alt rex rex ("_ |_")
  | seq rex rex ("_ _")
```

- Note:
  - no "par"-variant necessary!
  - priorities ommited!

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as signature (Version 3):

```
type rex
  consts
  range   :: [char, char] ⇒ rex ("_–_")
  char    :: char ⇒ rex ("_")
  dot     :: rex ("._")
  bracket :: rex ⇒ rex ("[ _ ]")
  star    :: rex ⇒ rex ("_*")
  alt     :: [rex, rex] ⇒ rex ("_ |_")
  seq     :: [rex, rex] ⇒ rex ("_ _")
```

Example: Regular Expressions

- The Language:
  - Input into Isabelle:
    can be identical for all three versions,
    but highly different in their internal representation!

$$\mathrm{a}(\mathrm{c} - \mathrm{d})*$$

(provided a, c and d are the usual character constants . . .)

Question:

How can we represent semantics?

Semantic Representation

- Deep Embeddings:
  - syntax as *explicit* datatype (e.g. Version 2)
  - interpretation as explicit function mapping each element of the language to a value

- Shallow Embedding:
  - syntax implicit in *notation* for operators on the semantic domain (based on Version 3)

Semantic Representation: Example

- Deep Embeddings (based on Version 2):
  - semantic function L primitive recursive:

```
consts L :: rex ⇒ string set
primrec L
  L(range x y) = {[a] | a. x ≤ a ∧ a ≤ y}
  L(char x) = {[x]}
  L(dot) = {[x] | x. True}
  L(bracket r) = {[]} ∪ L(r)
  L(star r) = lfp(λX.{[]} ∪ {x@y| x,y. x∈ L(r) ∧ y ∈ X})
  L(alt r_1 r_2) = L(r_1) ∪ L(r_2)
  L(seq r_1 r_2) = {x @ y | x,y. x ∈ L(r_1) ∧ y ∈ L(r_2)}
```

where $\{f\ a \mid a.\ P\ a\} \equiv f`\{a.\ P\ a\}$

Representation

- Deep Embeddings: Question

Why does Version 1 does not work here for use with primitive recursion?

Semantic Representation: Example

- Shallow Embeddings (based on Version 3)
  - Operators are directly interpreted in domain:

type rex = string set

defs

range_def	range x y ≡ {[a] | a. x ≤ a ∧ a ≤ y}
char_def	char x ≡ {[x]}
dot_def	dot ≡ {[x] | x. True}
bracket_def	bracket r ≡ {[]} ∪ r
star_def	star r ≡ lfp (λX.{[]}∪{x@y | x,y. x∈r ∧ y∈X})
alt_def	alt r_1 r_2 ≡ r_1 ∪ r_2
seq_def	seq r_1 r_2 ≡ {x @ y | x,y. x∈r_1 ∧ y∈r_2}

where {f a | a. P a} ≡ f ‘ {a. P a}

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Can we induce over shallow embeddings?
  - Yes, if we can give the domain an inductive structure, e.g.:

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):

consts REX :: string set set

inductive "REX"

intros

range $\quad$ range x y $\in$ REX

char $\quad$ char x $\in$ REX

dot $\quad$ dot $\in$ REX

bracket $\quad$ r_1 $\in$ REX $\Longrightarrow$ {bracket x | x. x $\in$ r_1} $\in$ REX

star $\quad$ r_1 $\in$ REX $\Longrightarrow$ {star x | x. x $\in$ r_1} $\in$ REX

alt $\quad$ $[\![$r_1 $\in$ REX; r_2 $\in$ REX$]\!] \Longrightarrow$
{alt x y | x,y. x $\in$ r_1 $\wedge$ y $\in$ r_2} $\in$ REX

seq $\quad$ $[\![$r_1 $\in$ REX; r_2 $\in$ REX$]\!] \Longrightarrow$
{seq x y | x,y. x $\in$ r_1 $\wedge$ y $\in$ r_2} $\in$ REX

- from REX to rex we can go via type definition . . .

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage: we inherit the binding structure of HOL!
  - Example: We can add

$$\mu X.f\ X = lfp\ f$$

into the rex-language and have:

$$\mu X.a[X]a = \{a^{2n}.n\}!!!$$

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Note: In

$$\mu X.a[X]a = \{a^{2n}.n\},$$

X is a HOL-Variable, a a constant, etc.

In a deep embedding, we would have to introduce a constant set (e.g. $X\ 1$, $X\ 2$, . . .) for variables and handle them in own substitution functions . . .

and take care of name clashes and name captures ourselves !

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage:
    we inherit the binding structure of HOL!
  - In deep embeddings, this requires extra

    variable symbols,

    substitution functions,

    typing functions, . . .

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Drawbacks:
    - ▷ binding structure may be too tight (blockstructured, λ-calculus oriented)
    - ▷ typing may be too tight
    - ▷ the semantic domain may not have an inductive structure (α,$\alpha \Rightarrow \beta$, $\alpha\ set$)
    - ▷ unfortunately, this rules out some crucial meta-language proofs (e.g. completeness proofs)

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - ▷ Theory/Collection "Lambda" (Isabelle99)
    - ▷ Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:
    - ▷ HOL itself !!! (λ, $\exists$, @, . . .)
    - ▷ HOLCF ([MNOS99]; see Isabelle distribution)
    - ▷ HOL-Z [BRW03], HOL-CSP [TW97]
    - ▷ MiniML (discussed in the following)
  - In-between: MicroJava [NOP00]

Scott's Approach to Denotational Semantics

- Overall aim: Having a structure with a fixpoint property:

$$\mu X.E(X) = E(\mu X.E(X))$$

- Topological Approach
  (denotational, Scott/Stratchey Approach):

  $(\sqsubseteq, C)$ is a "complete partial order" (CPO)
- Idea: We represent $(\sqsubseteq, C)$ as subclass of order!

Denotational Semantics

- Some basic definitions:

$$
\begin{array}{lll}
\text{upper bound} & S <| x & \equiv \forall\, y.\ y \in S \longrightarrow y \leq x \\
\text{is\_least\_ub} & S <<| x & \equiv S <| x \wedge (\forall\, u.\ S <| u \longrightarrow x \leq u) \\
\text{lub} & \text{lub}(S) & \equiv \text{THE } x.\ S <<| x \\
\text{directed} & \text{directed}\ X & \equiv (X \neq \{\}) \wedge \\
 & & \quad (\forall\, a \in X,\ b \in X.\ \exists c \in X.\ a \leq c \wedge b \leq c) \\
\textbf{fix} & \textbf{fix}\ f & \equiv \text{lub}(\text{range}(\lambda\, i.\ \ \text{iterate}\ \ i\ \ F\ \bot))
\end{array}
$$

Denotational Semantics

- Some basic definitions:

```
classes cpo0 < order
consts ⊥ :: 'a :: cpo0    (*In cpo's there is a constant Bottom *)
axclass
  cpo < cpo0
  least    "⊥ ≤x"          (* ... which is least*)
  complete "(directed X) ⟹ (∃ b. X <<| b)
```

Denotational Semantics

- Some basic definitions:
  - continuity:

    cont f ≡∀ Y. Y⊆A∧directed Y⟶f‘Y <<|f(lub Y)

    where f ::’ a::cpo⇒’b::cpo

  - admissibility:

    adm(P) ≡
    ∀ Y. Y⊆A∧directed Y ⟶(∀ x∈ Y. P x) ⟶P(lub Y)

    where P ::’ a::cpo⇒ bool

Denotational Semantics

- Main Theorems:
  - recursion (Knaster-Tarski):

    $\mathsf{cont}\ f \Longrightarrow \mathbf{fix}\ f = f\ (\mathbf{fix}\ f)$

    where f ::' a :: cpo⇒'a :: cpo

  - fixpoint induction:

    $[\![\ \mathsf{cont}\ f;\ \mathsf{adm}(P);\ P(\bot);\ \forall x.\ P(x) \Longrightarrow P(f\ x)]\!] \Longrightarrow P(\mathbf{fix}\ f)$

Denotational Semantics

- Minor Theorems:

$(\forall\ x.\ \mathsf{cont}(f\ x)) \Longrightarrow \mathsf{cont}(\lambda\ x.\ \ \mathbf{fix}\ \ (f\ x))$

$\mathsf{cont}(\mathsf{Pair}),\ \ \mathsf{cont}(\mathsf{fst}), \mathsf{cont}(\mathsf{snd})$

$\mathsf{cont}(\lambda\ x.\ \ c)$

$\mathsf{cont}(\lambda\ x.\ \ x)$

$\mathsf{cont}(f)\ \wedge \mathsf{cont}(g) \Longrightarrow \mathsf{cont}(f\ \circ\ g)$

$\mathsf{cont}(u)\ \wedge \mathsf{cont}(v) \Longrightarrow \mathsf{adm}(\lambda\ x.\ u\ x \leq v\ x)$

$\mathsf{cont}(u)\ \wedge \mathsf{cont}(v) \Longrightarrow \mathsf{adm}(\lambda\ x.\ u\ x \leq v\ x)$

$\mathsf{adm}\ P \wedge \mathsf{adm}\ Q \Longrightarrow \mathsf{adm}(\lambda\ x.P\ x \wedge Q\ x)$

$\mathsf{adm}\ P \wedge \mathsf{adm}\ Q \Longrightarrow \mathsf{adm}(\lambda\ x.P\ x \vee Q\ x)$

Denotational Semantics

- Constructions: The "flat cpo"

datatype 'a up = lift 'a | down
instance up :: (term) ord

constdefs
drop :: 'a::cpo up ⇒ 'a
"drop x ≡ case x **of** lift v ⟶ v | down ⟶ ⊥"

le_up_def "x ≤ y ≡ case x **of**
lift v ⟶ (case y **of** lift u ⟶ v = u
| down ⟶ False)
| down ⟶ True"

Denotational Semantics

- Extremely Nifty Constructions: Products, Function Space.

instance

"×" :: (ord, ord) ord

defs

le_pair_def "x ≤ y ≡ ((fst x ≤ fst y) ∧ (snd x ≤ snd y))"

arities

fun :: (term, order) order (* $fun = \longrightarrow *$)

defs

le_fun_def "f ≤ g ≡ (∀x. f x ≤ g x)

Denotational Semantics

- Extremely Nifty Consequences

Dom2 = Dom1 +
instance up :: (term) cpo (⊥_up_def, le_up_least , le_up_complete)
instance
"×" :: (cpo,cpo) cpo (⊥_pair_def , le_pair_least ,
instance
fun :: (term,cpo) cpo (le_fun_least , le_fun_complete)

⇒ CPO-checking is type checking!

Example: Shallow Embedding of MiniML

- Core Definitions of MiniML [Win96]:
  one exception (EXN), apply, closure, if_then_else, . . .

```
EXN_def     EXN ≡ ⊥
APPLY_def   F ^! x ≡ if x = ⊥ then ⊥
                     else if F = ⊥ then ⊥ else (drop F) x
LAM_def     Lam f ≡ lift f      (* LAM x. f x *)
REC_def     REC f ≡ fix f
 IF _def      ( IF x THEN y ELSE z) ≡ (case x of
                lift v ⇒ if v then y else z
              | down ⇒ ⊥ )
LET_def     LET s f ≡ f ^! s
```

Denotational Semantics

- More Definitions of MiniML [Win96]:
  Basic Operations, Global Declarations.

types Int = int up Bool = bool up

constdefs TIMES :: [Int, Int] ⇒ Int
"TIMES ≡ strictify (λx::nat. strictify (λy. lift (x∗y)))"
DIV :: [Int, Int] ⇒ Int
"DIV ≡ strictify (λx::nat. strictify (λy.
if y = lift 0 then ⊥ else lift (x∗y))))"

constdefs VAL :: "['a,'a] ⇒ bool"
"VAL f E ≡(f = E)"

```
  FUN :: " ['a :: cpo, 'a ⇒ 'a] ⇒ bool"
"FUN f F ≡(f = REC(F)) ∧cont F"
```

Denotational Semantics

- Now we can derive the operational semantics of MiniML: evaluation relation, canonical forms [Win96]

```
constdefs eval :: ['a::cpo,'a] ⇒ bool
          "eval s t ≡(s = t)"
          cf :: ['a::cpo] ⇒ bool
          "cf t ≡(t~=⊥)"

syntax    "-A -> "::['a,'a}] ⇒ bool ( infixl 50)

translations
          "s -A -> t" ≡"eval s t"
```

Denotational Semantics

- Now we can derive the operational semantics of MiniML: strict beta-reduction, if_then_else with EXN (=⊥).

```
⟦cf c; cf c_2;
  t_1 −A −> (LAM x. t x);
  t_2 −A −> c_2; (t(c_2)) −A −> c ⟧
⟹ (t_1 ^! t_2) −A −> c

⟦t_1 −A −> EXN; t_2 −A −> c_2⟧
⟹ (t1 ^! t_2) −A −> EXN

⟦t_1 −A −> (LAM x. t x);
```

$$\begin{array}{l} \quad \mathsf{t\_2}\ -\mathsf{A}\ ->\ \mathsf{EXN}\ \rrbracket \\ \Longrightarrow (\mathsf{t1}\ \hat{}!\ \mathsf{t\_2})\ -\mathsf{A}\ ->\ \mathsf{EXN} \end{array}$$

$$\begin{array}{l} \llbracket \mathsf{cf}\ \mathsf{c\_2};\ \mathsf{t\_1}\ -\mathsf{A}\ ->\ \mathsf{TRUE}; \\ \quad \mathsf{t\_2}\ -\mathsf{A}\ ->\ \mathsf{c\_2} \rrbracket \\ \Longrightarrow (\ \mathtt{IF}\ \ \mathsf{t\_1}\ \mathtt{THEN}\ \mathsf{t\_2}\ \mathtt{ELSE}\ \mathsf{t\_3})\ -\mathsf{A}\ ->\ \mathsf{c\_2} \end{array}$$

$$\begin{array}{l} \llbracket \mathsf{cf}\ \mathsf{c\_2};\ \mathsf{t\_1}\ -\mathsf{A}\ ->\ \mathsf{FALSE};\ \mathsf{t\_3}\ -\mathsf{A}\ ->\ \mathsf{c\_2}\ \rrbracket \\ \Longrightarrow (\ \mathtt{IF}\ \ \mathsf{t\_1}\ \mathtt{THEN}\ \mathsf{t\_2}\ \mathtt{ELSE}\ \mathsf{t\_3})\ -\mathsf{A}\ ->\ \mathsf{c\_2} \end{array}$$

$$\begin{array}{l} \llbracket \mathsf{t\_1}\ -\mathsf{A}\ ->\ \mathsf{EXN}\ \rrbracket \\ \Longrightarrow (\ \mathtt{IF}\ \ \mathsf{t\_1}\ \mathtt{THEN}\ \mathsf{t\_2}\ \mathtt{ELSE}\ \mathsf{t\_3})\ -\mathsf{A}\ ->\ \mathsf{EXN} \end{array}$$

Denotational Semantics

- Now we can derive the operational semantics of MiniML: recursion, basic operations.

$$[\![\text{cont}(\lambda X.\ \ \text{lift}\ \ (f\ X))\ ]\!] \Longrightarrow$$
$$\text{REC}(\lambda X.(\text{LAM}\ x\_1.\ f\ X\ x\_1))\ -A\ ->$$
$$(\text{LAM}\ x\_1.\ f\ (\text{REC}(\lambda X.\ (\text{LAM}\ x\_1.\ f\ X\ x\_1)))\ x\_1)$$

$$[\![cf\ \ c\_1;\ \ cf\ \ c\_2;\ \ t\_1\ \ -A\ ->\ c\_1;\ t\_2\ -A\ ->\ c\_2\ ]\!] \Longrightarrow$$
$$(\text{TIMES}\ t\_1\ t\_2)\ -A\ ->\ \text{lift}(c\_1 * c\_2)$$

Summary

- Isabelle/HOL is a powerful framework for embedding programming languages
  - via deep embeddings
  - via shallow embeddings
  - denotational semantics can be used as definitional basis for operational semantics !!!

- First Priciple for language embeddings: KEEP IT AS SHALLOW AS POSSIBLE !!!

References

[BRW03] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. Hol-z 2.0: A proof environment for z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.

[MNOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[NOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[TW97] H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.

[Win96] Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction*. MIT Press, 1996. 3rd ed.