Component-Based Dynamic Semantics for Caml Light

Martin Churchill, Peter D. Mosses
Swansea University

SLS Workshop, Cambridge
June 2013

Component-based Semantics

Programming language semantics, via reusable components

- Reusable components = "funcons"
  - Expanding collection of fundamental, independent symbols for computational behaviour
  - Each with (fixed) intuitive meaning + formal specification
- Language semantics = translation
  - Mapping from programming language to funcons
  - Language constructs *decomposed* into combination of funcons

...for Caml Light

We demonstrate the approach for *Caml Light*

- ≈ core of Standard ML
- ≈ a sublanguage of OCaml
- functional + imperative, algebraic data types, pattern matching, exceptions, mutual recursion, ...

Also studied by [Owens et al., ESOP 2008] (OCaml light), [Charguéraud, ESOP 2013]

Caml Light

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
(* type 'a btree = Empty | Node of 'a * 'a btree * 'a btree *)

let rec member x btree =
   match btree with
     Empty -> false
   | Node(y, left, right) ->
      if x = y then true else
      if x < y then member x left else member x right;;
(* val member : 'a -> 'a btree -> bool = <fun> *)

let rec insert x btree =
  match btree with
    Empty -> Node(x, Empty, Empty)
  | Node(y, left, right) ->
      if x <= y then Node(y, insert x left, right)
                else Node(y, left, insert x right);;
(* val insert : 'a -> 'a btree -> 'a btree = <fun> *)
```

Ingredients

caml light grammar

caml light reference manual

collection of funcons

plus:

- context-free translation : caml light grammar $\rightarrow$ funcons
- funcon specifications (via inductive rules)

Examples

```
expr[[ while E1 do E2 done ]] =
   seq(while-true(expr[[ E1 ]], effect(expr[[ E2 ]])), tuple-empty)

expr[[ match E with P1 -> E1 ... ]] =
   apply(
     prefer-over
       (abs[[ P1 -> E1 ... ]],
        abs(any, throw('Match-failure'))),
     expr[[ E ]])
```

key points:

- perspicuity + preciseness

Ingredients

Caml Light Grammar

abstract syntax $+$ disambiguation

Caml Light Grammar

abstract syntax $+$ disambiguation
$\Downarrow$
funcons

Expression based language (`expr`). Other main non terminals:

- `patterns`, `types`, `let-bindings`, ...

(see html page)

Semantic Universe

Funcons partitioned into 'sorts'

loosely describing kind of behaviour

e.g.:

- value – a terminated computation (includes int, boolean, ...)
- expr – an expression computes a value
- env – an environment is an identifier $\to$ value map
- decl – a declaration computes an environment
- abs – an abstraction computes a value, given a value
- patt – a pattern computes an environment, given a value
- ...

Some funcon signatures

apply(abs,value)	:	expr	effect(expr)	:	comm
abs(patt,expr)	:	abs	seq(comm,expr)	:	expr
any	:	patt	while-true(expr,comm)	:	comm
prefer-over(abs,abs)	:	abs	throw(exception)	:	expr
	⋮			⋮	

Some funcon signatures (polymorphic)

apply(abs,value)	:	expr	effect(X)	:	comm
abs(patt,expr)	:	abs	seq(comm,X)	:	X
any	:	patt	while-true(expr,comm)	:	comm
prefer-over(abs,abs)	:	abs	throw(exception)	:	X
	⋮			⋮	

Translation

abs[[simple-matching]]	:	abs
decl[[implementation]]	:	decl
decl[[let-bindings]]	:	decl
expr[[expr]]	:	expr
patt[[pattern]]	:	patt
value[[value]]	:	value
	⋮	

(see CBS file)

Funcon specifications

Formally specified by:

- Inductive rules over small-step evaluation relation

Utilising:

- Modular SOS (MSOS) for treatment of environments, stores, exceptions... [Mosses, JLAP 2004]
- Supports **formal implicit propagation** for unmentioned components (each corresponds to a *category*)
- Concretely, we use I-MSOS notation [Mosses and New, SOS 2008]

(see CSF)

Some advanced features for scalability

Implicitly generated 'patience rules' (*cf.* strictness):

- scope(env,X) : X lifted to scope(decl,X) : X by

$$\frac{x \to x'}{\text{scope}(x, y) \to \text{scope}(x', y)}$$

- lifts data operations to computations
  e.g. int_plus(int,int) : int to int_plus(expr,expr) : expr
  with arbitrary (possibly interleaved) argument evaluation

Some advanced features for scalability

Implicitly generated 'patience rules' (*cf.* strictness):

- scope(env,X) : X lifted to scope(decl,X) : X by

$$\frac{x \to x'}{\text{scope}(x, y) \to \text{scope}(x', y)}$$

- lifts data operations to computations
  e.g. int_plus(int,int) : int to int_plus(expr,expr) : expr
  with arbitrary (possibly interleaved) argument evaluation

Second-order (parametrised) funcons:

- seq first evaluates arguments in left-to-right-order
  e.g. seq int_plus(expr,expr) : expr
- invert creates a pattern, inverting a particular data constructor
- ...

Validation

Correctness – laws

How can we verify correctness of our semantics?

1. Proving funcon equivalence laws
   - e.g. associativity, commutativity, ...
   - via (open) bisimulation techniques [Mosses, Mousavi and Reniers, EXPRESS 2010]
   - prove independently, just considering relevant rules
   - equations are preserved when adding funcons (and auxiliary entities)
   - bisimilarity as a congruence [Churchill and Mosses, FoSSaCS 2013]
     - rule format, satisfied by all funcons used for Caml Light

Correctness – prototyping

2. Testing (running programs according to semantics)
   - Checks funcon specs + translation
   - Translation – parsing programs + rewriting to funcon trees
     - current prototype uses ASF+SDF, moving to more contemporary tools
   - Funcon specs
     - funcon rules $\xrightarrow{\textit{generation}}$ Prolog rules
     - alternatives: e.g. K [Roșu and Șerbănuță, JLAP 2010]

⇒ run programs according to the semantics

Caml-Light Tests

- Chap. 1 of OCaml manual gives examples, mostly Caml Light
- We can run all of these via semantics (few mins)
  - runs program to yield final answer + entity trace

- Casper Bach Poulsen working on more efficient animation via partial evaluation techniques [Bach Poulsen, SLS]

Conclusions

In progress: static semantics (with Paolo Torrini)

Funcons:

- Each funcon has "static rules" over additional relations – type inhabitance (:), subtyping ($<$), ...
- MSOS framework for entities can also be used for static semantics

In progress: static semantics (with Paolo Torrini)

Funcons:

- Each funcon has "static rules" over additional relations – type inhabitance (:), subtyping ($<$), ...
- MSOS framework for entities can also be used for static semantics

Language translation:

- Single translation into funcons for static + dynamic aspects
  - Additions to the translation for type expressions & definitions
  - A few equations modified for Caml-Light specific typing behaviour
- Mostly complete for Caml Light
  - OCaml manual Chapter 1 examples work

Conclusions

- Component-based semantics applied to: Caml Light language
  - Dynamic semantics ✔, static semantics on its way...
- Developed and tested CBS techniques on a small language
  - including basic prototyping support for animating semantics
- Funcons in our rule format $\Rightarrow$ bisimulation is a congruence
- Next steps:
  - PLanCompS is now specifying C#, to prove the scalability of our techinques.

Thank You.