Component-Based Dynamic Semantics for Caml Light

Martin Churchill and Peter D. Mosses

Department of Computer Science, Swansea University

We are exploring scalability of component-based programming language semantics via case studies. In particular, we translate Caml Light to language-independent fundamental constructs (funcons), each of which has independent semantics given by inductive rules. This factorisation provides an accessible semantic specification of the language, while retaining formal precision. The semantics can be validated by animating the translation and rules, and running test programs. Moreover, funcon equivalence laws (bisimulations) can be proved independently; a recently developed rule format ensures that this is a congruence for all of our funcons. The funcons used form a basis of a growing repository to be used in further language specifications. This work forms part of the PLanCompS project [`www.plancomps.org`].

1 Fundamental Constructs

Each of our fundamental constructs (funcons) represents a unit of computation: a token representing a piece of behaviour (either run-time or during static checking). Each has a particular arity, and may be applied to other funcon terms. For example, seq allows two computations to be run sequentially, assign updates the imperative store and deref can examine it, abs abstracts a term over a given pattern, apply may be used to apply such abstractions, scope scopes a series of declarations locally to a term, and so on. The collection is open ended, and each funcon should be reusable in component-based specifications of many languages.

By a translating a language to combinations of funcons, we provide it with a component-based semantics [5]. This semantics is formal – the semantics of the language is defined precisely by the translation of the language into funcons and the semantics of the funcons. The semantics is also accessible to the non-specialist – the translation is defined perspicuously by a number of simple equations, and each funcon has an informal description accessible to the non-semanticist. Further, once the collection of funcons has matured, the non-specialist might even *write* language semantics, by translating the language to combinations of funcons which they only informally understand. Good tool support is required to enable this, e.g. providing prototyping for the generated languages. Related tools for programming language semantics include [2,7]; our main novelty is the factorisation via reusable funcons, maximising scalability and perspicuity.

We illustrate this approach for the language Caml Light [3], comparable to the core of Standard ML. We first provide a couple of examples of translations from programming language constructs (productions in the context-free grammar for Caml Light) to funcons. In the first example, we translate Caml Light's while loop using the while_true funcon. Additional funcons are needed to specify, for example, that the final result is the empty tuple (unit):

```
expr[[ while E1 do E2 done ]] =
    seq(while_true(expr[[E1]], effect(expr[[E2]])),
        tuple_empty)
```

We next consider a translation of Caml Light's pattern matching, corresponding to the following production of the context-free grammar:

expr : ... | match *expr* with *simple-matching* | ...

In this case, the analysis decomposes this into an application of an abstraction to a matched expression . The abstraction is derived from the semantics of the *simple-matching* using a function abs[[_]] defined elsewhere in the semantics (see Section 3). The semantics of match must also take into account what happens when the pattern fails to match the given value:

```
expr[[ match E with SM ]] =
    apply(prefer_over(abs[[SM]], abs(any, throw('Match_failure'))),
          expr[[E]])
```

Signatures and descriptions of the funcons used above can be found in Fig. 1, with a larger collection (for full Caml Light) at www.plancomps.org/churchill2013b.

apply(abs,value) : expr	Applies an abstraction to a given value
abs(patt,expr) : abs	Abstracts an expression over a pattern
any : patt	Matches any value and produces no bindings
prefer_over(abs,abs) : abs	Tries to apply the first abstraction to a given argument, if undefined tries to apply the second abstraction
assign(var,value) : comm	Updates a variable to a given value
deref(var) : expr	Computes the value assigned to a variable
effect(expr) : comm	Evaluates an expression and discards the result
seq(comm,expr) : expr	Runs a command, then evaluates an expression
while_true(expr,comm) : comm	While an expression evaluates to true, runs a command
throw(exception) : expr	Throws the given exception

Fig. 1. Some funcon signatures and descriptions

2 Operational Specification of Funcons

We specify the semantics of each funcon independently using inductive SOS-style operational rules. The behaviour of various funcons may interact in effectful ways, and such effects are recorded in 'auxiliary entities'. Examples include the environment **env**, the store **store**, an exception tag **exception**. Crucially, each funcon specification only mentions the entities relevant for that particular funcon, and the other entities are propagated according to the mechanics of *Modular*

$$\frac{E1 \rightarrow E1'}{\mathsf{assign}(E1, E2) \rightarrow \mathsf{assign}(E1', E2)} \quad (1) \qquad \frac{E2 \rightarrow E2'}{\mathsf{assign}(E1, E2) \rightarrow \mathsf{assign}(E1, E2')} \quad (2)$$

$$(\mathsf{assign}(Var, Value), \mathbf{store}\ Store) \rightarrow (\mathsf{skip}, \mathbf{store}\ \mathsf{map\_update}(Store, Var, Value)) \quad (3)$$

Fig. 2. Operational rules for assign

SOS [4]. Our concrete notation is based upon I-MSOS [6]. Rules for the assign funcon are given in Fig. 2.

Here, rules (1) and (2) are 'patience rules' and are in fact generated automatically from the signature of assign. The declared signature assign(var,value) : comm is extended to assign(expr,expr) : comm by generalising value arguments to computation arguments. Such arguments are evaluated in an unspecified, possibly interleaved, order.

If one wished to specify left-to-right evaluation, one could use 'seq assign' instead. Here, 'seq' is an example of a second-order funcon, which takes a funcon as a special parameter. Another example used in our Caml Light semantics is invert: if F is a binary data constructor then the pattern invert $F(Patt1, Patt2)$ will match precisely those values of the form $F(Value1, Value2)$ where *Value1* matches *Patt1* and *Value2* matches *Patt2*. Second-order funcons are given behaviour via operational rules in the usual way, and enhance the scalability of our approach.

Other than (modular) SOS, one could give formal semantics to funcons in other ways, e.g. using frameworks such as [2,7]. The crucial point is that each funcon must have independent semantics which can be directly reused, so funcons need only be added when scaling up to larger languages.

3 Caml Light Semantics

Using these techniques, we have translated the Caml Light language [3] into funcons, producing a component-based semantics of the language. This submission focuses on the dynamic aspects, but the techniques naturally extend to static aspects, where each funcon is assigned static rules (type checking, evaluation of type expressions, compile-time resolution of terms). The complete semantics is given by 13 translation functions, 98 equations and 40 funcons (plus data operations) together with their rules. Each translation function maps a nonterminal in the Caml Light reference grammar [3] to funcons of a particular sort, with functions typically named by the sort that they produce. The main funcon sorts and translation equation signatures are given in Fig. 3.

For Caml Light, the value sort contains ground values (integers, Booleans, strings, floats, chars) as well as records (maps, wrapped in a data constructor), variants for disjoint unions (a value tagged with a constructor) and functions (an abstraction wrapped with a constructor to form a value). The language specification also defines initial bindings caml_light_library : env. The complete semantic translation can be found at `www.plancomps.org/churchill2013b`.

abs[[_]] : simple-matching → abs	An abs computes a value given a value
decl[[_]] : let-bindings → decl	An decl computes an env
expr[[_]] : expr → expr	An expr computes a value
patt[[_]] : pattern → patt	A patt computes an env given a value
value[[_]] : value → value	See prose for value description

Fig. 3. Some translations and funcon sorts

4 Validation

As well as funcon semantics being both easy to read and write, they also facilitate prototyping. In particular, the semantic equations can be implemented by simple term rewriting systems; and the funcon operational rules may be animated. The case study demonstrated here has been validated using the ASF+SDF MetaEnvironment for the former and by generating Prolog code for the latter. This way, we have been able to take Caml Light programs and run them according to the semantics. At the above URL, one may find a sequence of example programs together with the corresponding funcon term translation, and the output produced by animating the rules (the final result and the MSOS composed trace of the auxiliary entities). Thus we can *test* our semantics: an agile and light weight alternative to proving properties during language engineering.

Funcons may also be validated by proving equivalence laws. Each of the funcons used in the Caml Light semantics is in the *MSOS tyft* format which ensures that bisimulation is a congruence [1]. This means that each bisimulation between funcon terms (e.g. associativity of sequencing) is valid in arbitrary Caml Light contexts. We have found that proofs of bisimulation are well-suited to formalisation in theorem provers, and they may be stored in the envisioned repository with the relevant programs. This would provide a repository of constructs with tested semantics and proven laws, for use in future language specifications.

References

1. M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In *FoSSaCS 2013*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013.
2. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 1st edition, 2009.
3. X. Leroy. The Caml Light system, documentation and user's guide. http://caml.inria.fr/pub/docs/manual-caml-light/, 1997.
4. P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.
5. P. D. Mosses. Component-based semantics. In *SAVCBS '09*, pages 3–10. ACM, 2009.
6. P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. In *SOS '08*, Electr. Notes Theor. Comput. Sci., Vol. 229(4), pages 49–66. Elsevier, 2009.
7. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.