Reuse and co-evolution in CBS language specifications

Peter Mosses

Swansea University and TU Delft

pdmosses.github.io

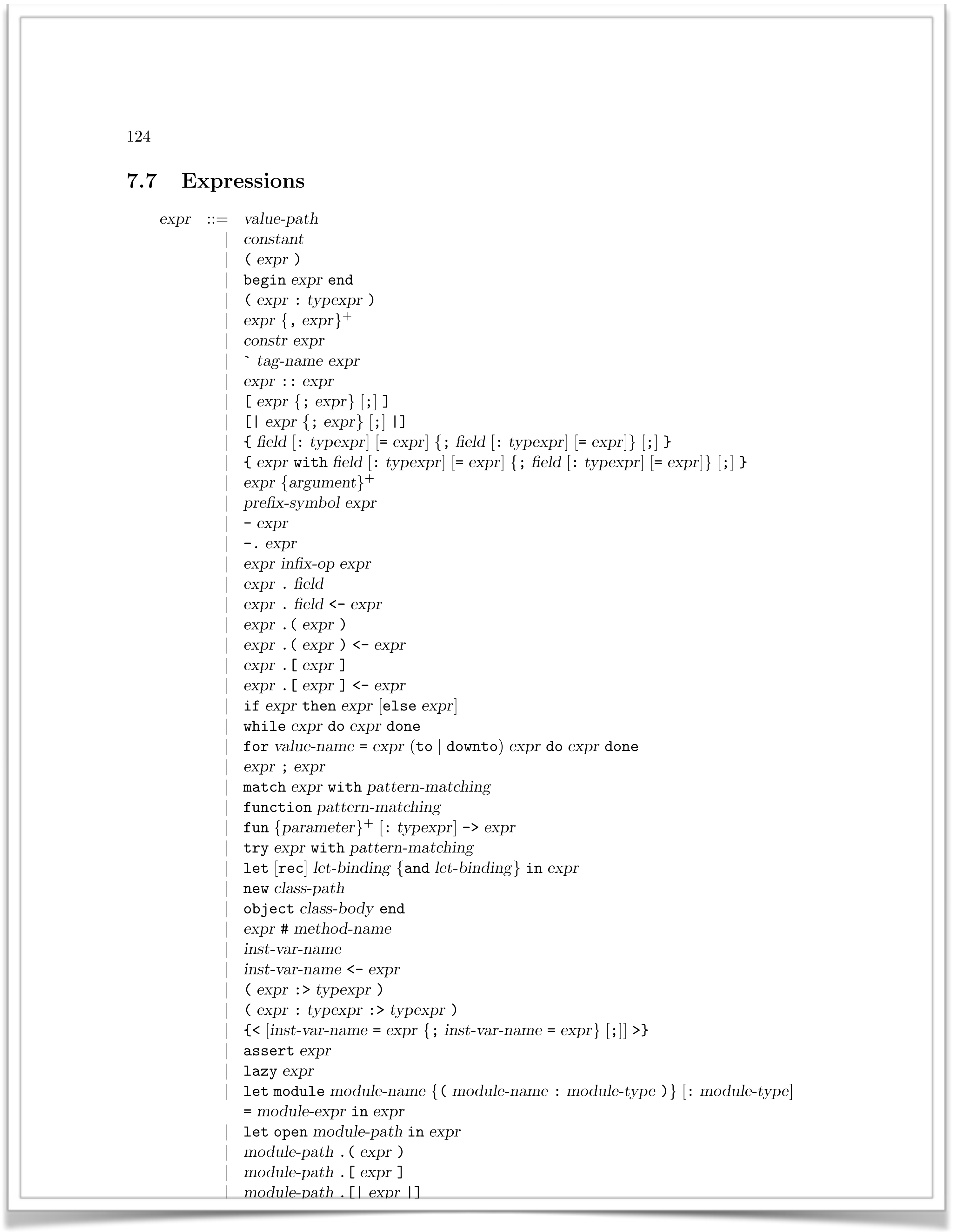
LangDev Meetup at Amazon Amsterdam, 21–22 March 2019

Formality of language specifications

Complete language specifications
produced by language developers themselves

- **syntax**
  - *reasonably* **formal** 😇

OCAML:



124

7.7 Expressions

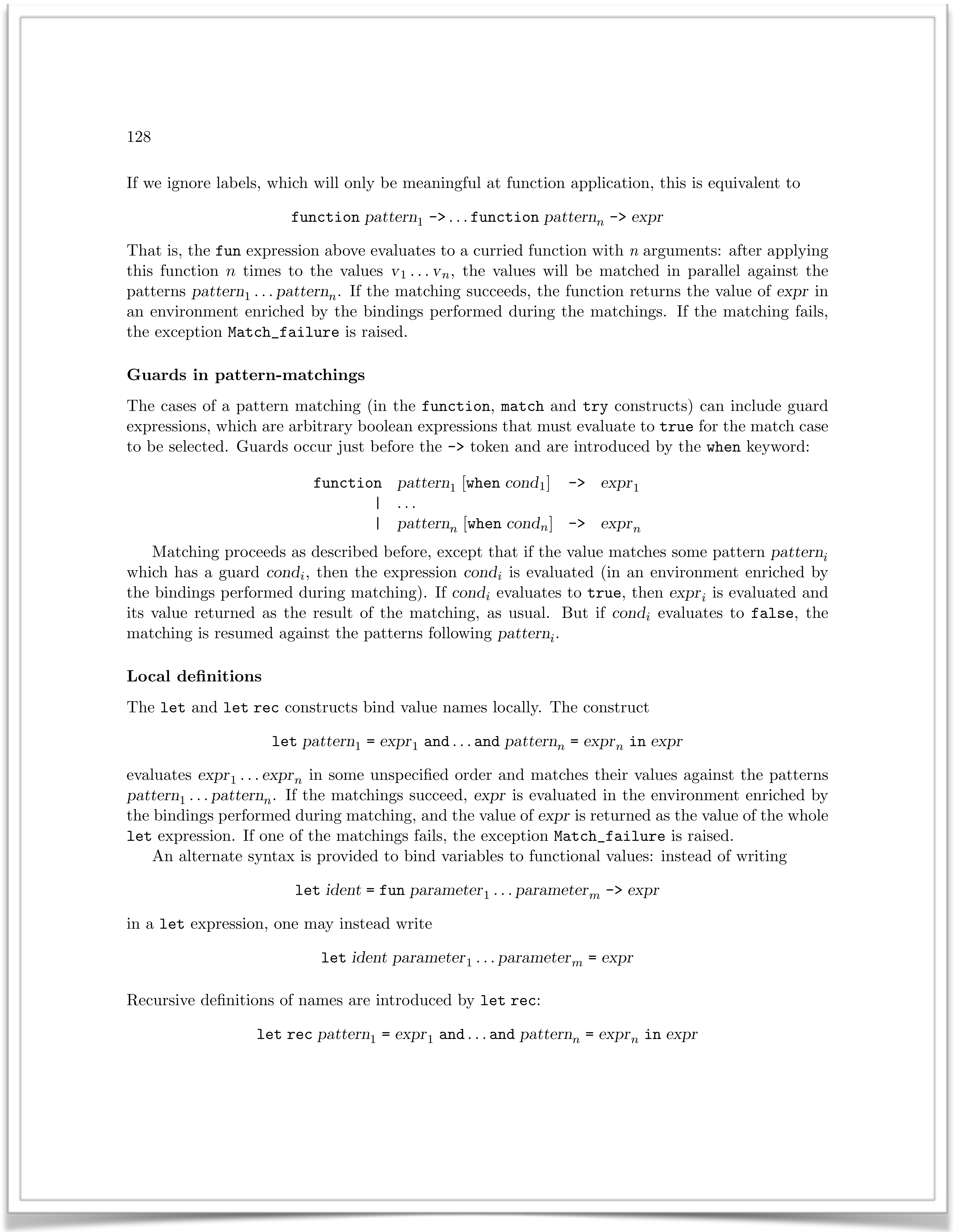
```
expr ::= value-path
       | constant
       | ( expr )
       | begin expr end
       | ( expr : typexpr )
       | expr {, expr}+
       | constr expr
       | ` tag-name expr
       | expr :: expr
       | [ expr {; expr} [;] ]
       | [| expr {; expr} [;] |]
       | { field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | { expr with field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | expr {argument}+
       | prefix-symbol expr
       | - expr
       | -. expr
       | expr infix-op expr
       | expr . field
       | expr . field <- expr
       | expr .( expr )
       | expr .( expr ) <- expr
       | expr .[ expr ]
       | expr .[ expr ] <- expr
       | if expr then expr [else expr]
       | while expr do expr done
       | for value-name = expr (to | downto) expr do expr done
       | expr ; expr
       | match expr with pattern-matching
       | function pattern-matching
       | fun {parameter}+ [: typexpr] -> expr
       | try expr with pattern-matching
       | let [rec] let-binding {and let-binding} in expr
       | new class-path
       | object class-body end
       | expr # method-name
       | inst-var-name
       | inst-var-name <- expr
       | ( expr :> typexpr )
       | ( expr : typexpr :> typexpr )
       | {< [inst-var-name = expr {; inst-var-name = expr} [;]] >}
       | assert expr
       | lazy expr
       | let module module-name {( module-name : module-type )} [: module-type]
         = module-expr in expr
       | let open module-path in expr
       | module-path .( expr )
       | module-path .[ expr ]
       | module-path .[| expr |]
```

Formality of language specifications

Complete language specifications
produced by language developers themselves

- **syntax**
  - *reasonably* **formal** 😇
- **semantics** (static and dynamic)
  - *completely* **informal** 🙄
  - *a few exceptions:* ADA, SCHEME, STANDARD ML, WEBASSEMBLY

OCAML:



128

If we ignore labels, which will only be meaningful at function application, this is equivalent to

`function` $pattern_1$ `->` ... `function` $pattern_n$ `->` $expr$

That is, the `fun` expression above evaluates to a curried function with n arguments: after applying this function n times to the values $v_1 \ldots v_n$, the values will be matched in parallel against the patterns $pattern_1 \ldots pattern_n$. If the matching succeeds, the function returns the value of $expr$ in an environment enriched by the bindings performed during the matchings. If the matching fails, the exception `Match_failure` is raised.

Guards in pattern-matchings

The cases of a pattern matching (in the `function`, `match` and `try` constructs) can include guard expressions, which are arbitrary boolean expressions that must evaluate to `true` for the match case to be selected. Guards occur just before the `->` token and are introduced by the `when` keyword:

`function` $pattern_1$ [`when` $cond_1$] `->` $expr_1$
`|` ...
`|` $pattern_n$ [`when` $cond_n$] `->` $expr_n$

Matching proceeds as described before, except that if the value matches some pattern $pattern_i$ which has a guard $cond_i$, then the expression $cond_i$ is evaluated (in an environment enriched by the bindings performed during matching). If $cond_i$ evaluates to `true`, then $expr_i$ is evaluated and its value returned as the result of the matching, as usual. But if $cond_i$ evaluates to `false`, the matching is resumed against the patterns following $pattern_i$.

Local definitions

The `let` and `let rec` constructs bind value names locally. The construct

`let` $pattern_1$ `=` $expr_1$ `and` ... `and` $pattern_n$ `=` $expr_n$ `in` $expr$

evaluates $expr_1 \ldots expr_n$ in some unspecified order and matches their values against the patterns $pattern_1 \ldots pattern_n$. If the matchings succeed, $expr$ is evaluated in the environment enriched by the bindings performed during matching, and the value of $expr$ is returned as the value of the whole `let` expression. If one of the matchings fails, the exception `Match_failure` is raised.
An alternate syntax is provided to bind variables to functional values: instead of writing

`let` $ident$ `= fun` $parameter_1 \ldots parameter_m$ `->` $expr$

in a `let` expression, one may instead write

`let` $ident$ $parameter_1 \ldots parameter_m$ `=` $expr$

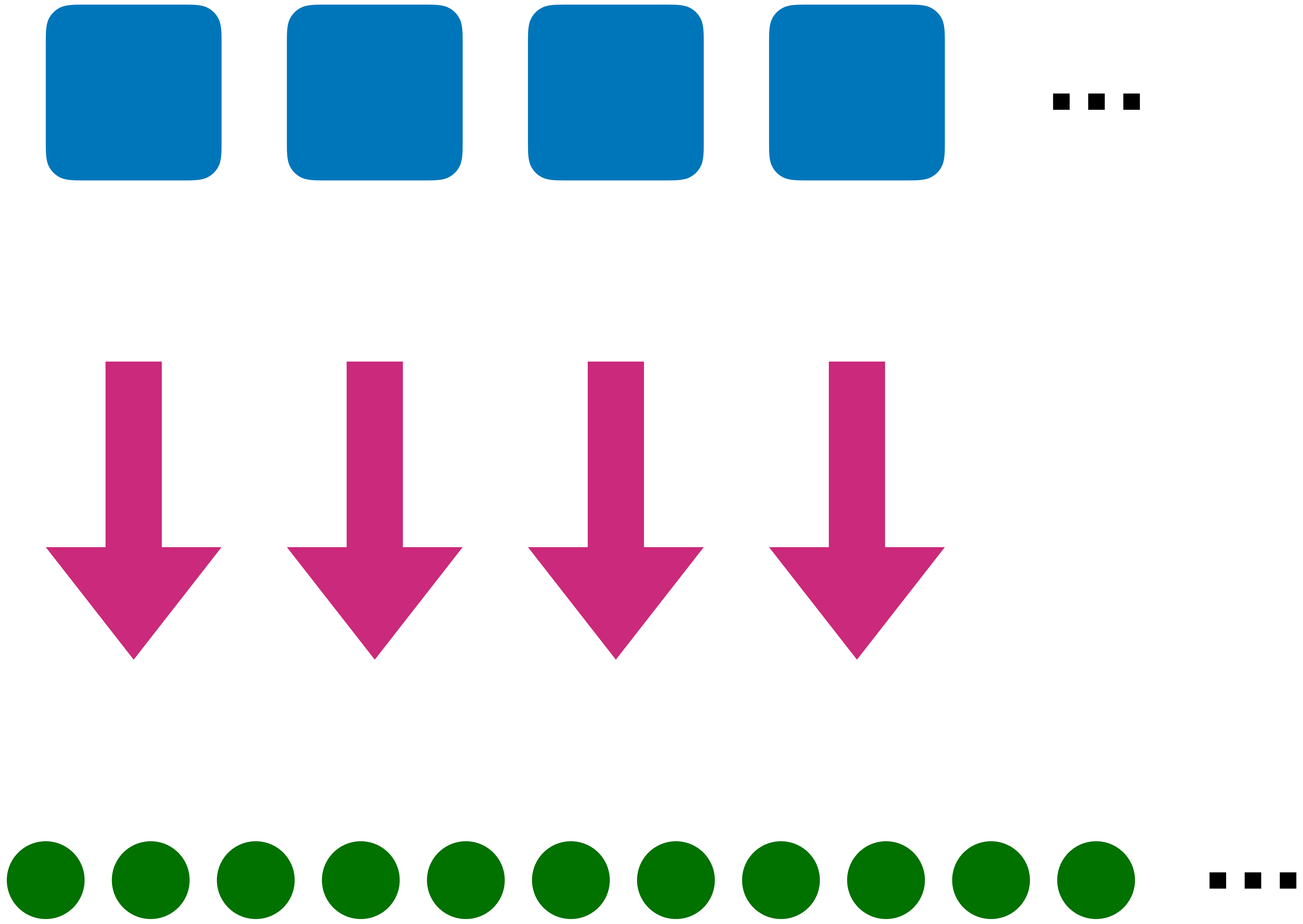
Recursive definitions of names are introduced by `let rec`:

`let rec` $pattern_1$ `=` $expr_1$ `and` ... `and` $pattern_n$ `=` $expr_n$ `in` $expr$

Reuse and co-evolution

CBS: component-based semantics

- **semantics** : language → funcons
  - context-free, compositional
- **funcons (fundamental constructs)**
  - open-ended library of *fixed* items



Developed by the PLANCOMPS project

- EPSRC funding 2011–16; now an open collaboration

Claim: CBS can **significantly** reduce the effort of formal semantics !

Reusable components

Funcons – *not languages !*

- **familiar** programming concepts
- **simpler** than language constructs
- **fixed** definitions
- **open-ended** library
- **unbiased** to any language class

Example:

```
Funcon
  sequential(_:(=>null-type)*, _:=>T) : =>T

Rule
                   X ---> X'
  ----------------------------------------
  sequential(X, Y+) ---> sequential(X', Y+)
Rule
  sequential(null-value, Y+) ~> sequential(Y+)
Rule
  sequential(Y) ~> Y
```

Co-evolution of languages and specifications

Translations

- **language** → funcons
  - *dependence on language syntax*
- **context-free** translation
  - *compositional*
  - *specified by equations*

Examples:

```
Semantics eval[[ _:exp ]] : => ld-values

Rule eval[[ E1 ':=' E2 ]] =
  assign( eval[[ E1 ]], eval[[ E2 ]] )

Rule eval[[ '!' E ]] = assigned( eval[[ E ]] )

Rule eval[[ E1 ';' E2 ]] =
  sequential( effect( eval[[ E1 ]] ), eval[[ E2 ]] )

Rule eval[[ 'while' E1 'do' E2 ]] =
  while-true( eval[[ E1 ]], eval[[ E2 ]] )
```

Tool support for CBS specifications

Creating, editing, browsing

- *grammars, funcons, translations*

Generating prototypes

- language **parser**
- funcon **interpreter**
- **translator** : language → funcons
  - *hence language interpreter*

CBS workbench

- based on SPOOFAX

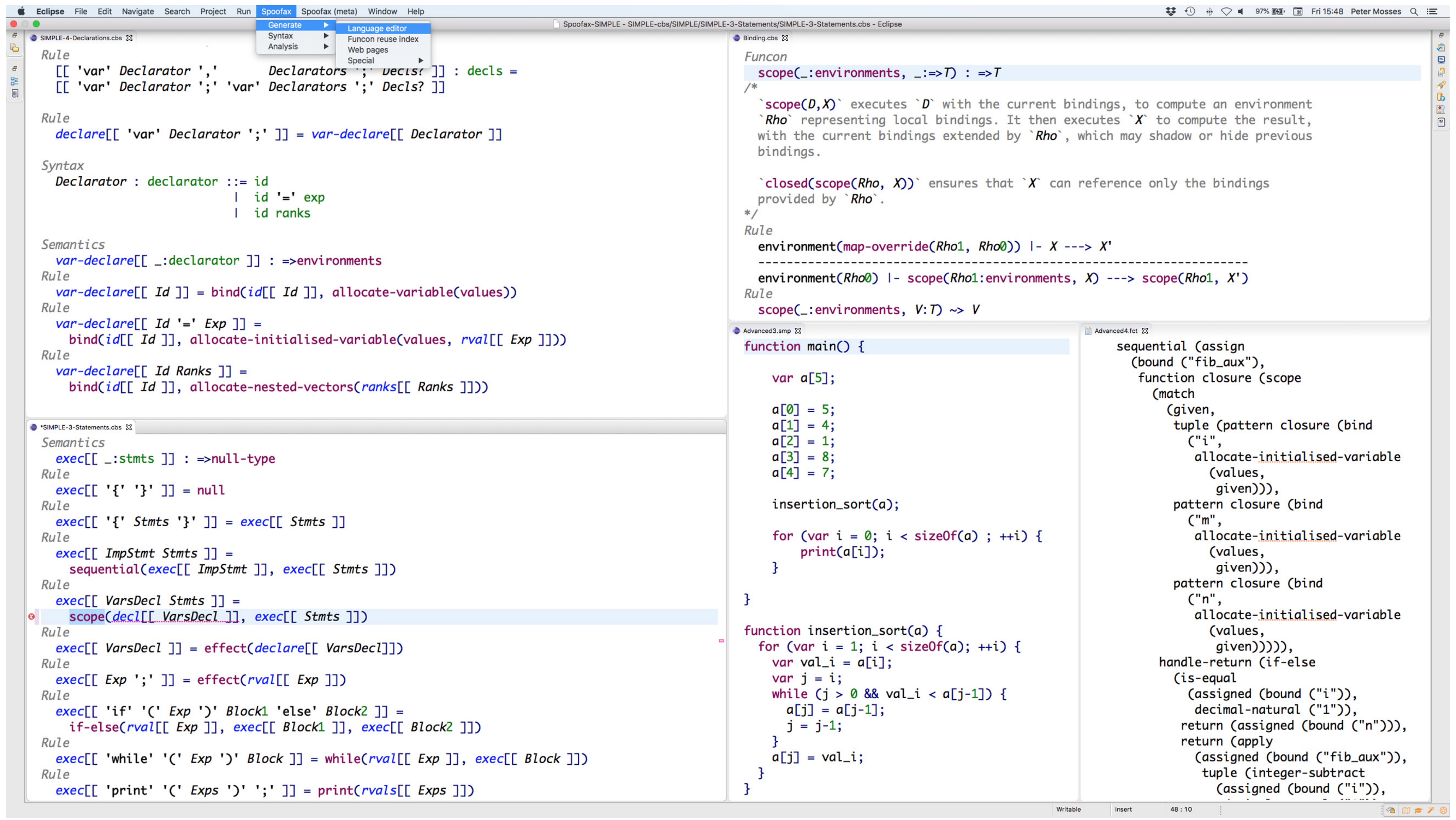


Fig. 5. The IDE for CBS in action. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

L.T. van Binsbergen et al. / Journal of Logical and Algebraic Methods in Programming 103 (2019) 184–212

Demo

Incremental specification

- LD: a demo language
  - *literals*
    lambda-calculus
    arithmetic
  - *references*
  - *threads* x

No changes to previous rules!

Grammar:

```
Syntax E:exp ::= int
               | id
// Call-by-value lambda-calculus:
               | 'lambda' id '.' exp
               | exp exp
               | 'let' id '=' exp 'in' exp
               | '(' exp ')'
// Arithmetic and Boolean expressions:
               | exp '+' exp
               | exp '*' exp
               | exp '/' exp
               | exp '<=' exp
               | exp '&&' exp
               | 'if' exp 'then' exp 'else' exp
// References and imperatives:
               | 'ref' exp
               | exp ':=' exp
               | '!' exp
               | exp ';' exp
               | '(' ')'
               | 'while' exp 'do' exp
// Multithreading:
               | 'spawn' exp
               | 'join' exp
```

Current status

CBS-beta [plancomps.github.io/CBS-beta]

- **Funcons-beta**
- **Languages-beta**
  - *toy:* IMP, SIMPLE, MINIJAVA
  - *medium:* OCAML-LIGHT, SL
  - *pending:* IMP++, SIMPLE-THR
    - multithreading

CBS-Editor

- SPOOFAX/ECLIPSE plugin
- *under development…*

Funcons.Tools

- HASKELL package
- *generates interpreters for funcons from their definitions*

Conclusion

CBS: component-based semantics framework [plancomps.github.io]

- unified specification language with solid theoretical foundations
- support for reuse and co-evolution
- library of funcon definitions

CBS language workbench

- creating, editing, browsing specifications
- generating editors, translators, interpreters

CBS	Availability
CBS-beta	July 2018
threads	April 2019
CBS workbench	June 2019 ?
major case studies	2020 ?

Recent references

- [Executable component-based semantics]

Journal of Logical and Algebraic Methods in Programming 103 (2019) 184–212

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Executable component-based semantics

L. Thomas van Binsbergen [a,*], Peter D. Mosses [b,c], Neil Sculthorpe [d]

[a] Department of Computer Science, Royal Holloway, University of London, TW20 0EX, Egham, United Kingdom
[b] Department of Computer Science, Swansea University, SA2 8PP, Swansea, United Kingdom
[c] EEMCS, Programming Languages, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, the Netherlands
[d] Department of Computing and Technology, Nottingham Trent University, NG11 8NS, Nottingham, United Kingdom

ARTICLE INFO

Article history:
Received 6 July 2018
Received in revised form 18 December 2018
Accepted 21 December 2018
Available online 4 January 2019

Keywords:
Programming languages
Formal semantics
Reuse
Components
Tool support

ABSTRACT

The potential benefits of formal semantics are well known. However, a substantial amount of work is required to produce a complete and accurate formal semantics for a major language; and when the language evolves, large-scale revision of the semantics may be needed to reflect the changes. The investment of effort needed to produce an initial definition, and subsequently to revise it, has discouraged language developers from using formal semantics. Consequently, many major programming languages (and most domain-specific languages) do not yet have formal semantic definitions.
To improve the practicality of formal semantic definitions, the PLanCompS project has developed a component-based approach. In this approach, the semantics of a language is defined by translating its constructs (compositionally) to combinations of so-called fundamental constructs, or 'funcons'. Each funcon is defined using a modular variant of Structural Operational Semantics, and forms a language-independent component that can be reused in definitions of different languages. A substantial library of funcons has been developed and tested in several case studies. Crucially, the definition of each funcon is fixed, and does not need changing when new funcons are added to the library.
For specifying component-based semantics, we have designed and implemented a meta-language called CBS. It includes specification of abstract syntax, of its translation to funcons, and of the funcons themselves. Development of CBS specifications is supported by an integrated development environment. The accuracy of a language definition can be tested by executing the specified translation on programs written in the defined language, and then executing the resulting funcon terms using an interpreter generated from the CBS definitions of the funcons. This paper gives an introduction to CBS, illustrates its use, and presents the various tools involved in our implementation of CBS.

1. Introduction

New programming languages and domain-specific languages are continually being introduced, as are new versions of existing languages. Each language needs to be carefully specified, to determine the syntax and semantics of its programs. Context-free aspects of syntax are usually specified, precisely and succinctly, using formal grammars; in contrast, semantics (including static checks and disambiguation) is generally specified only informally, without use of precise notation. Infor-

* Corresponding author.
E-mail addresses: ltvanbinsbergen@acm.org (L.T. van Binsbergen), p.d.mosses@swansea.ac.uk (P.D. Mosses), neil.sculthorpe@ntu.ac.uk (N. Sculthorpe).

https://doi.org/10.1016/j.jlamp.2018.12.004

- [Software meta-languages and CBS]

Journal of Visual Languages and Computing 50 (2019) 39–48

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Software meta-language engineering and CBS

Peter D. Mosses[1]

Department of Computer Science, Computational Foundry, Bay Campus, Swansea University, Swansea SA1 8EN, United Kingdom

ARTICLE INFO

Keywords:
Semantics of programming languages
Meta-languages
Modularity

ABSTRACT

The SLE conference series is devoted to the engineering principles of software languages: their design, their implementation, and their evolution. This paper is about the role of language specification in SLE. A precise specification of a software language needs to be written in a formal meta-language, and it needs to co-evolve with the specified language. Moreover, different software languages often have features in common, which should provide opportunities for reuse of parts of language specifications. Support for co-evolution and reuse in a meta-language requires careful engineering of its design.

The author has been involved in the development of several meta-languages for semantic specification, including action semantics and modular variants of structural operational semantics (MSOS, I-MSOS). This led to the PLanCompS project, and to the design of its meta-language, CBS, for component-based semantics. CBS comes together with an extensible library of reusable components called 'funcons', corresponding to fundamental programming constructs. The main aim of CBS is to optimise co-evolution and reuse of specifications during language development, and to make specification of language semantics almost as straightforward as context-free syntax specification.

The paper discusses the engineering of a selection of previous meta-languages, assessing how well they support co-evolution and reuse. It then gives an introduction to CBS, and illustrates significant features. It also considers whether other current meta-languages might also be used to define an extensible library of funcons for use in component-based semantics.

1. Introduction

In general, it is good engineering practice to produce a full design specification of a new artefact before starting its construction. If the design needs to be adjusted during the construction, or a new version of the artefact is subsequently required, the design specification is updated accordingly. Moreover, a design often makes extensive use of pre-existing components that have precisely specified properties.

In software language engineering, however, developers seldom produce *complete and precise* language design specifications. This seems to be at least partly because of the effort required to specify a major software language in full detail, and subsequently co-evolve the specification together with the specified language. Perhaps a component-based approach could reduce the effort, and encourage language developers to specify the designs of new languages before implementing them?

The rest of this section recalls some general features of formal language specification, and discusses the relationship between formality and co-evolution. Section 2 examines some previous meta-languages, pointing out issues with co-evolution and reuse. Section 3 introduces CBS, a component-based framework for language specification; it illustrates how CBS facilitates co-evolution, then gives an overview of the initial library of reusable components provided with CBS. Section 4 indicates the current status of CBS and plans for its further development.

This article is based on the author's keynote at SLE 2017, extending [1]. Its contribution is an analysis of the support for co-evolution and reuse in selected meta-languages, together with an explanation of relevant CBS features; it does not present previously unpublished research results.

1.1. Formal language specification

A language specification defines requirements on implementations: which texts an implementation is to accept as well-formed, and what behaviour should be exhibited when executing such texts.[2] For conventional high-level programming languages, well-formedness may be divided into lexical syntax, context-free phrase structure, and context-sensitive constraints, all to be checked before program execution starts; the behavioural requirements generally include the relation between input and output, but exclude properties such as how much time or space program execution should take. Context-sensitive constraints are

E-mail address: p.d.mosses@swansea.ac.uk.
[1] Present address: EEMCS, Programming Languages, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands.
[2] Software languages and meta-languages can both be textual and/or graphical; we here consider purely textual languages, for simplicity.

https://doi.org/10.1016/j.jvlc.2018.11.003
Received 13 November 2018; Accepted 19 November 2018
Available online 20 November 2018