A component-based approach to semantics

Peter Mosses

Swansea University, UK

Modularity'15 • March18, 2015
Fort Collins, Colorado, USA

Modularity

Good to have!

What might be even better?

reusable components

Software development:

“The Unix Philosophy”

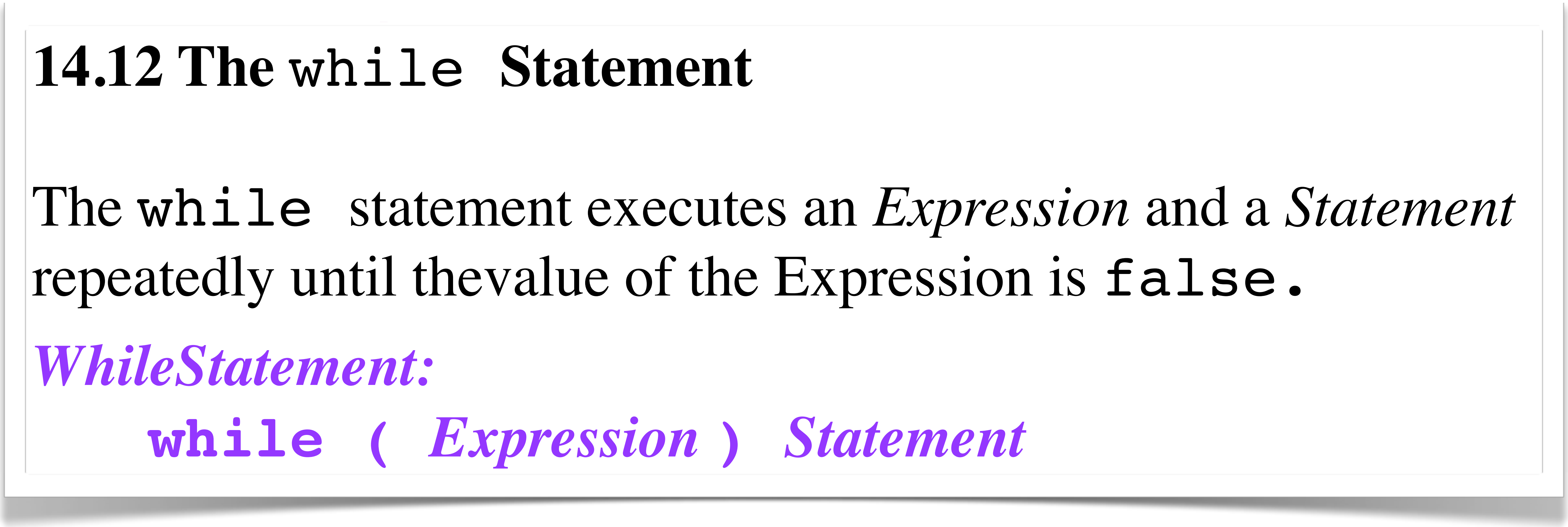
Programming language definitions:

component-based semantics

Programming language definitions

Reference manuals, standards documents

- ***syntax:***
  - always ***formal***



14.12 The `while` Statement

The `while` statement executes an *Expression* and a *Statement* repeatedly until thevalue of the Expression is `false.`

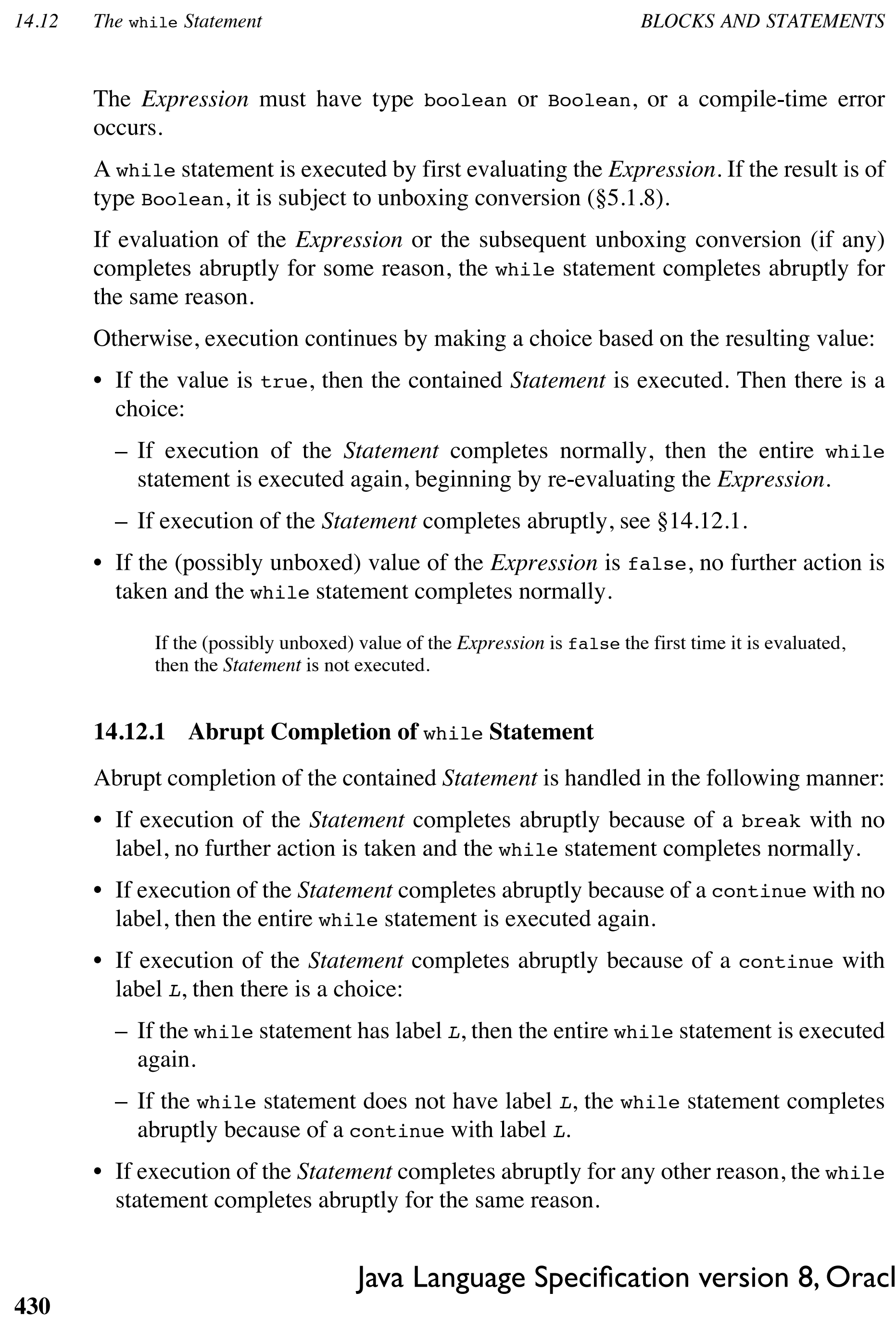
WhileStatement:

    while (Expression) Statement

Programming language definitions

Reference manuals, standards documents

- **syntax:**
  - always ***formal***
- **semantics:**
  - almost always ***informal***



14.12 *The* `while` *Statement* BLOCKS AND STATEMENTS

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

A `while` statement is executed by first evaluating the *Expression*. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `while` statement completes abruptly for the same reason.

Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the contained *Statement* is executed. Then there is a choice:
  - If execution of the *Statement* completes normally, then the entire `while` statement is executed again, beginning by re-evaluating the *Expression*.
  - If execution of the *Statement* completes abruptly, see §14.12.1.
- If the (possibly unboxed) value of the *Expression* is `false`, no further action is taken and the `while` statement completes normally.

If the (possibly unboxed) value of the *Expression* is `false` the first time it is evaluated, then the *Statement* is not executed.

14.12.1 Abrupt Completion of `while` Statement

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `while` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with no label, then the entire `while` statement is executed again.
- If execution of the *Statement* completes abruptly because of a `continue` with label *L*, then there is a choice:
  - If the `while` statement has label *L*, then the entire `while` statement is executed again.
  - If the `while` statement does not have label *L*, the `while` statement completes abruptly because of a `continue` with label *L*.
- If execution of the *Statement* completes abruptly for any other reason, the `while` statement completes abruptly for the same reason.

430

Java Language Specification version 8, Oracle

Formal semantics

Many semantic frameworks:

- operational, denotational, algebraic, axiomatic, …

Only a few *official language definitions* use formal semantics:

- ADA, MODULA-2, STANDARD ML, SCHEME

Some other languages have *unofficial* formal semantics:

- ALGOL 60, C, C#, JAVA, PL/I, PROLOG, …

Many major languages have no formal semantics:

- C++, HASKELL, OCAML, SCALA, …

Formal semantic frameworks

Operational

- VDL
- SOS (small- or big-step)
- reduction semantics, K
- ASM

Denotational

- Scott–Strachey
- VDM
- monadic

Axiomatic

- Hoare logic
- algebraic

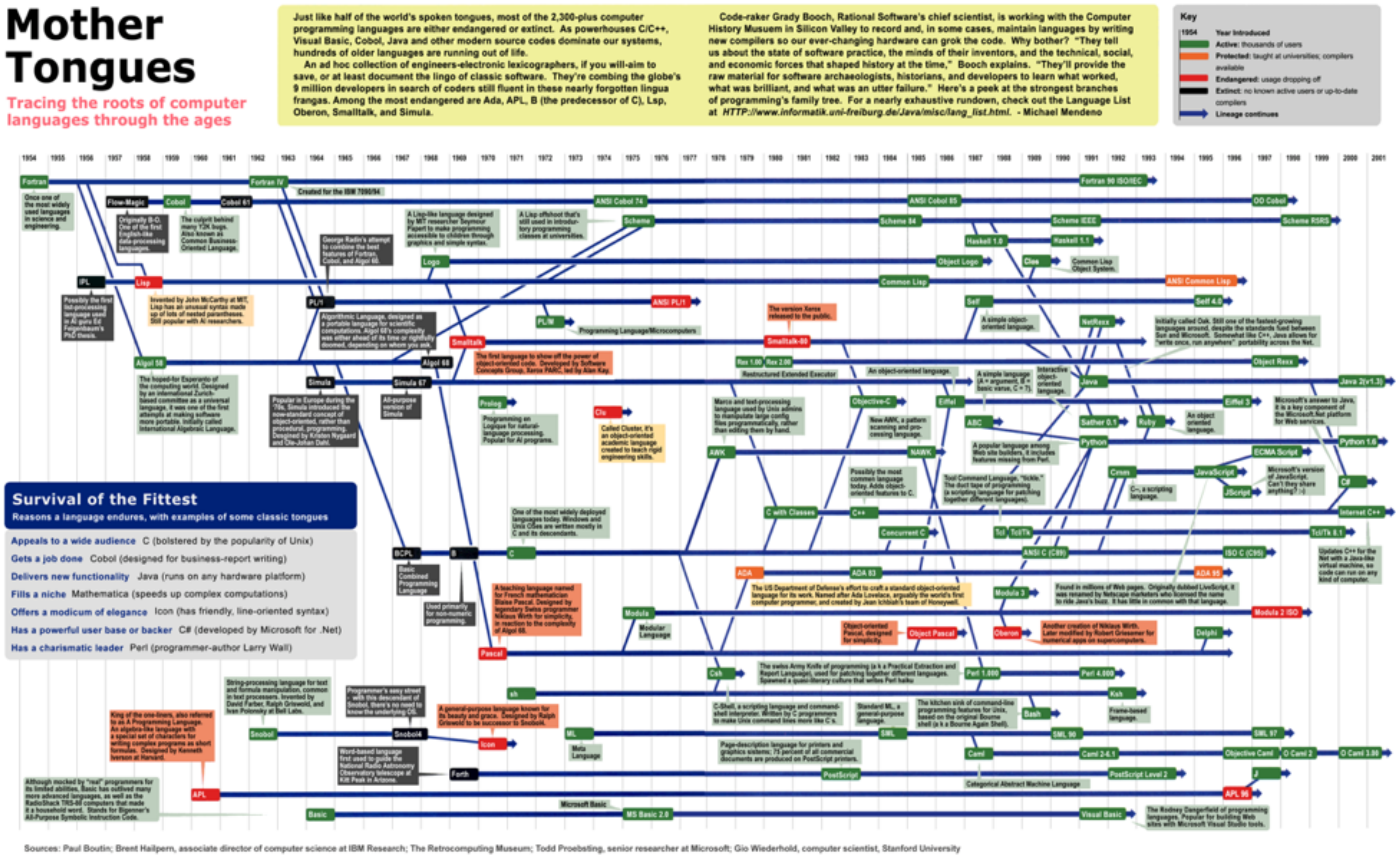
Hybrid

- action semantics
- UTP

Static

- typing rules
- abstract interpretation

Programming language evolution



Mother Tongues

Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.
An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Musuem in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html.* - Michael Mendeno

Key

1954	Year Introduced
🟩	**Active:** thousands of users
🟧	**Protected:** taught at universities; compilers available
🟥	**Endangered:** usage dropping off
⬛	**Extinct:** no known active users or up-to-date compilers
→	**Lineage continues**

Survival of the Fittest

Reasons a language endures, with examples of some classic tongues

Appeals to a wide audience	C (bolstered by the popularity of Unix)
Gets a job done	Cobol (designed for business-report writing)
Delivers new functionality	Java (runs on any hardware platform)
Fills a niche	Mathematica (speeds up complex computations)
Offers a modicum of elegance	Icon (has friendly, line-oriented syntax)
Has a powerful user base or backer	C# (developed by Microsoft for .Net)
Has a charismatic leader	Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

The importance of being formal

Only a ***formal*** semantics can be

- **precise**
- **concise**

and allow

- **validation**
- **reasoning**
- **prototyping**

How to improve?

Reusable components

to reduce the ***initial*** effort

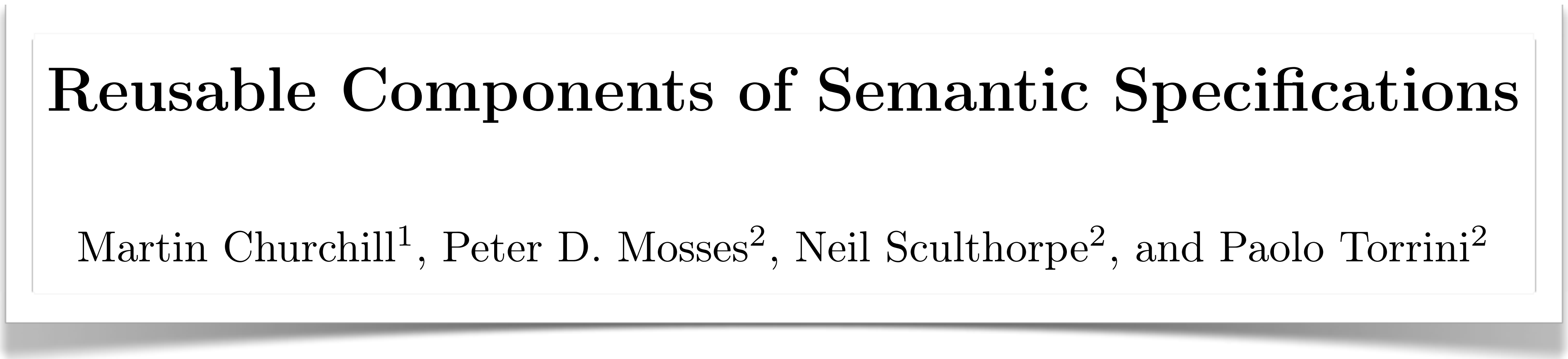
High modularity

to reduce the effort of ***change***

Tool support

to reduce the effort of ***getting it right!***

MODULARITY '14


Reusable Components of Semantic Specifications

Martin Churchill[1], Peter D. Mosses[2], Neil Sculthorpe[2], and Paolo Torrini[2]

Extended version: *Trans. AOSD*, special issue, 2015, in press.

- a component-based semantics of CAML LIGHT
- validated (by empirical testing)
- detailed introduction to the approach
- overview of preliminary tool support

Reusable components

Reusable software components

COTS – ‘Components Off The Shelf’

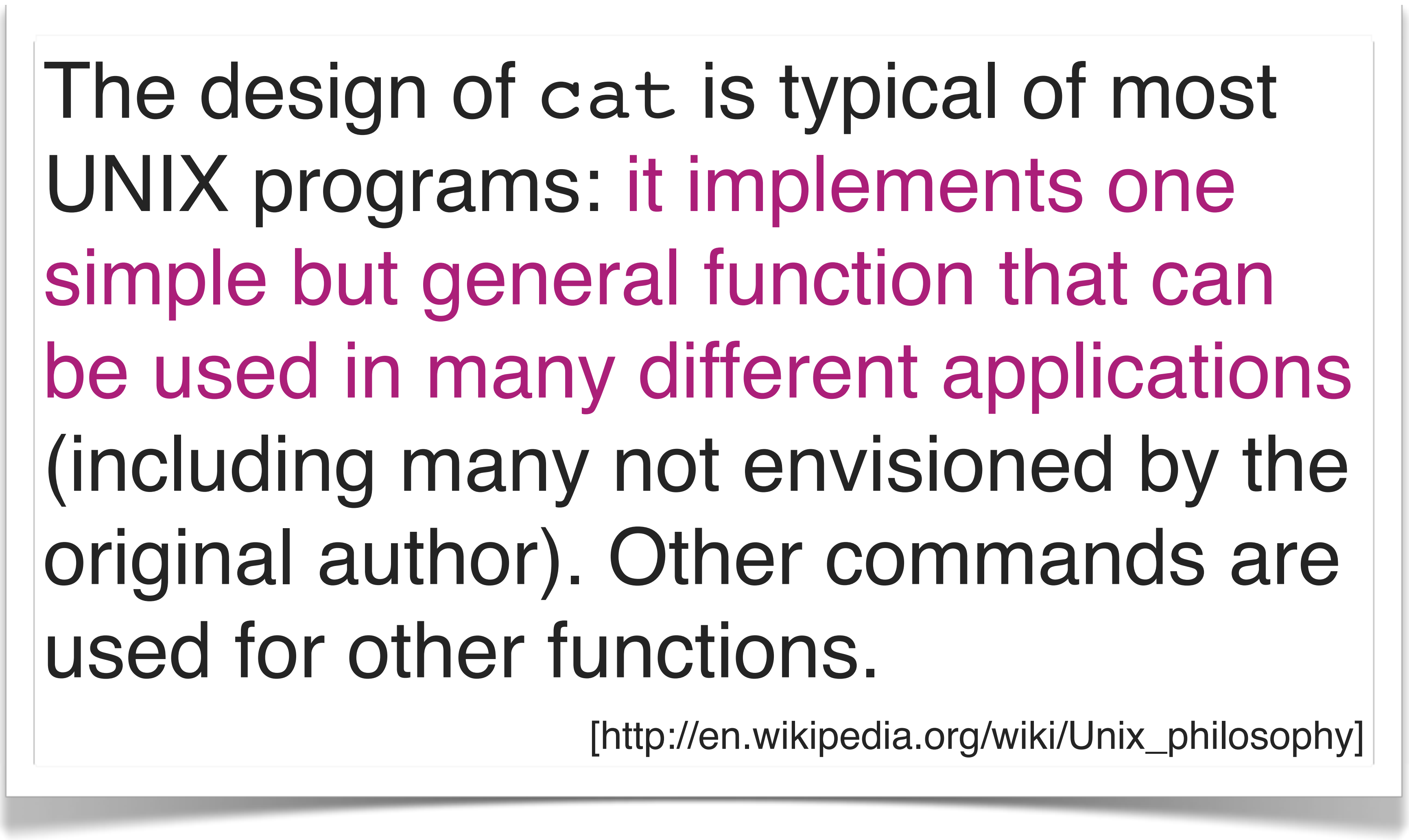
- typically complex software
  - *example:* Windows for driving medical devices

Libraries and packages

- greatly enhance productivity
- but upgrades to new versions can be problematic…

The Unix Philosophy

Formulated in the 1980s by Ken Thompson, Dennis Ritchie, Brian Kernighan, Doug McIlroy, Rob Pike, et al.



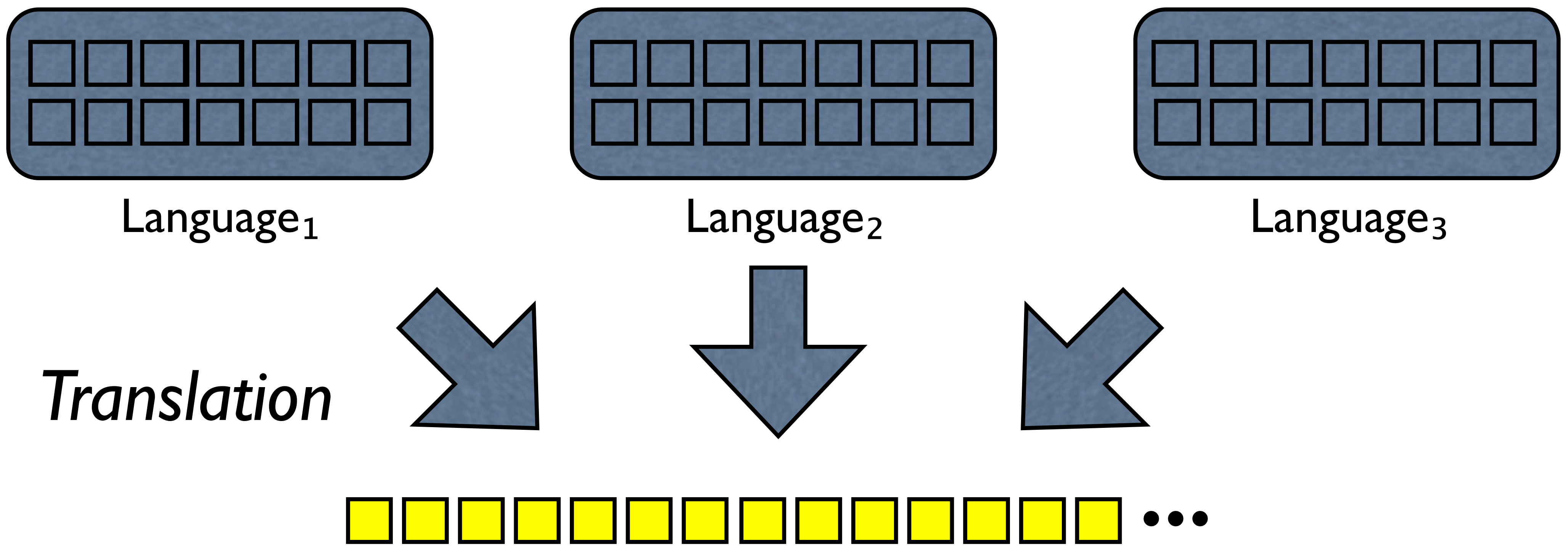
The design of `cat` is typical of most UNIX programs: it implements one simple but general function that can be used in many different applications (including many not envisioned by the original author). Other commands are used for other functions.

[http://en.wikipedia.org/wiki/Unix_philosophy]

Component-based semantics

Reusable components of language definitions

- ***language*** constructs?
- ***kernel language*** constructs?
- ***fundamental*** programming constructs!



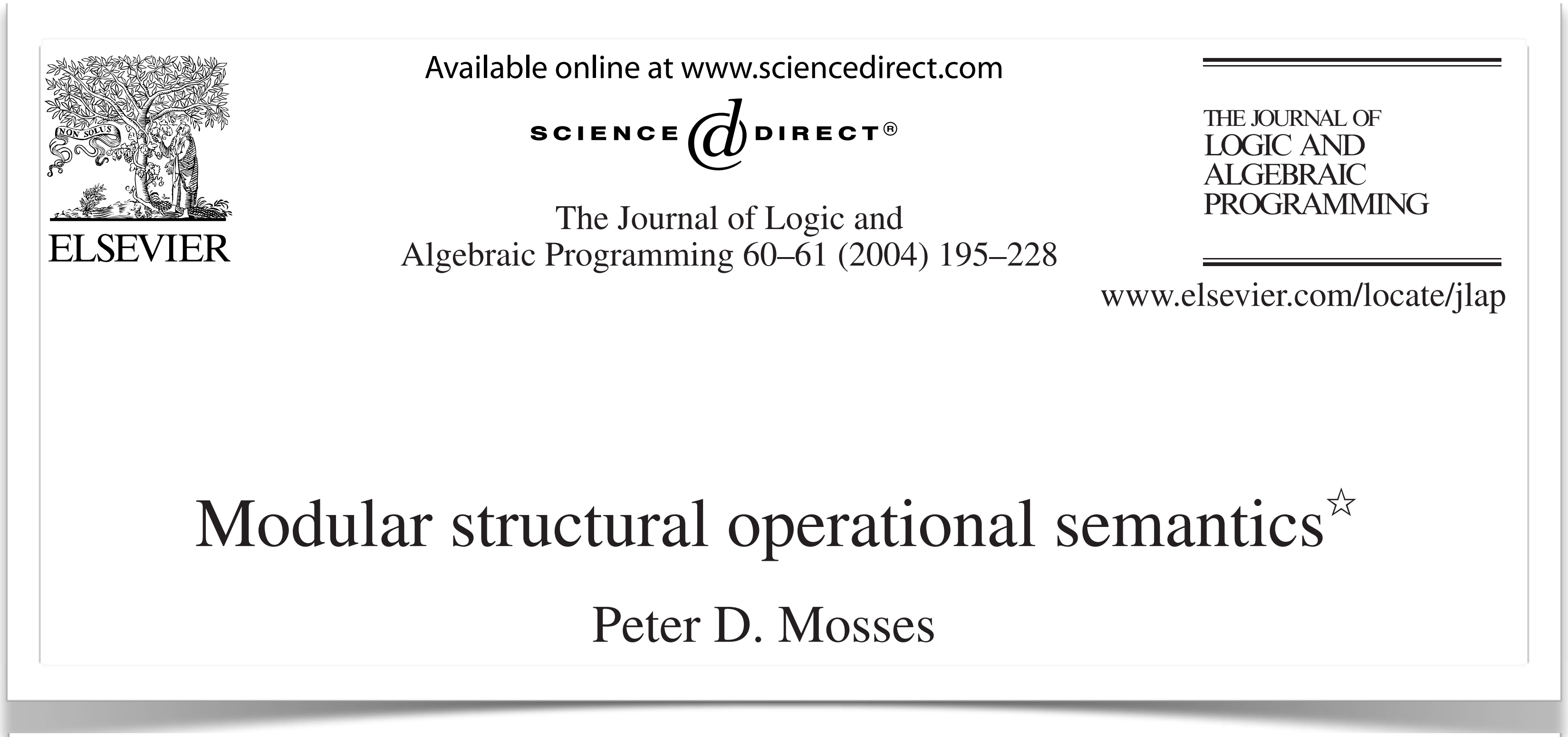
Reusable components

Fundamental constructs (funcons)

- correspond to ***individual*** programming constructs
  - each funcon is a separate component
- have *(when validated and released)*
  - ***fixed*** notation
  - ***fixed*** behaviour
  - ***fixed*** algebraic properties

specified/proved once and for all!

Modular foundations



Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

ELSEVIER

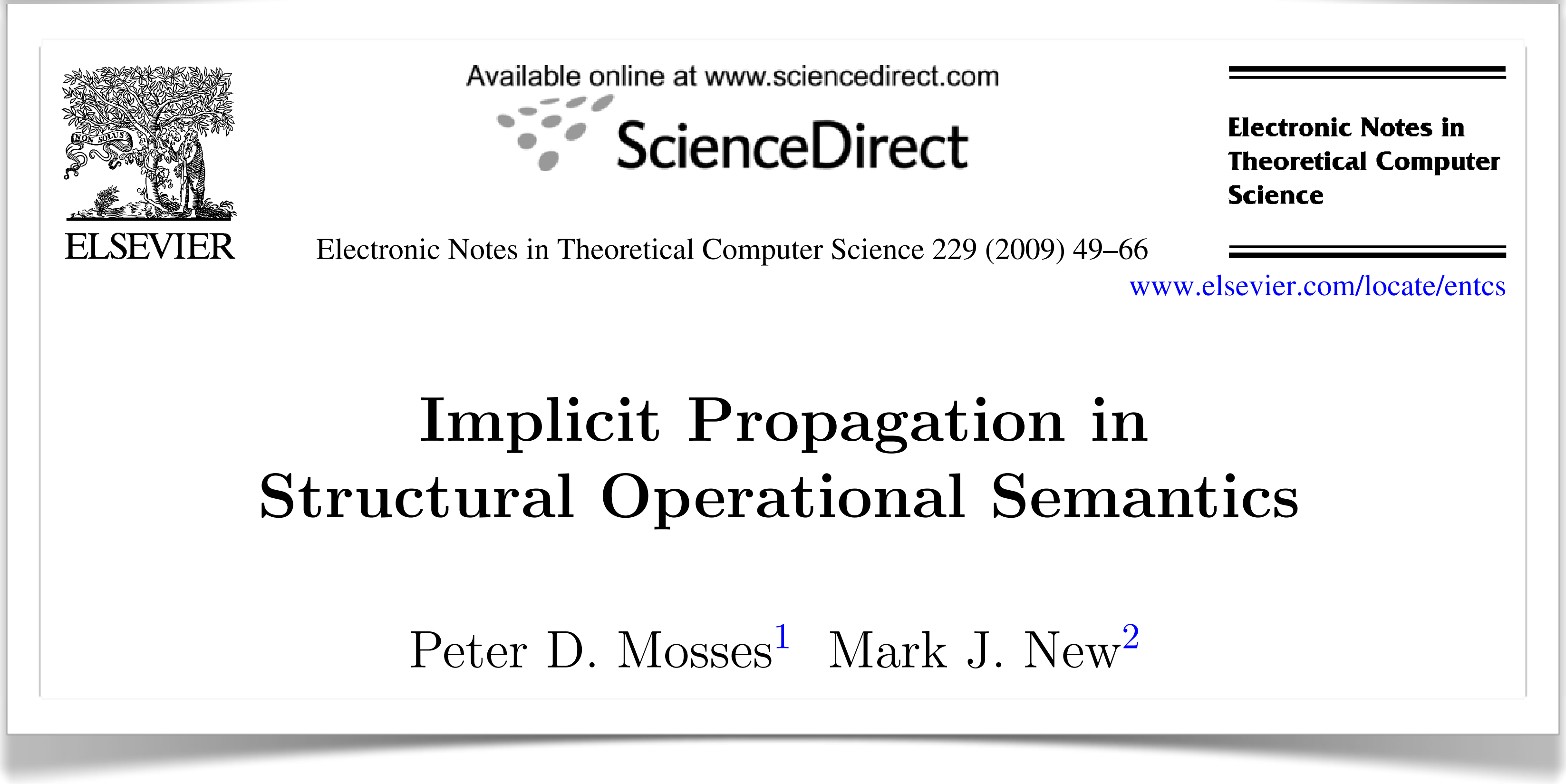
The Journal of Logic and Algebraic Programming 60–61 (2004) 195–228

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

Modular structural operational semantics☆

Peter D. Mosses



Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 229 (2009) 49–66

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Implicit Propagation in Structural Operational Semantics

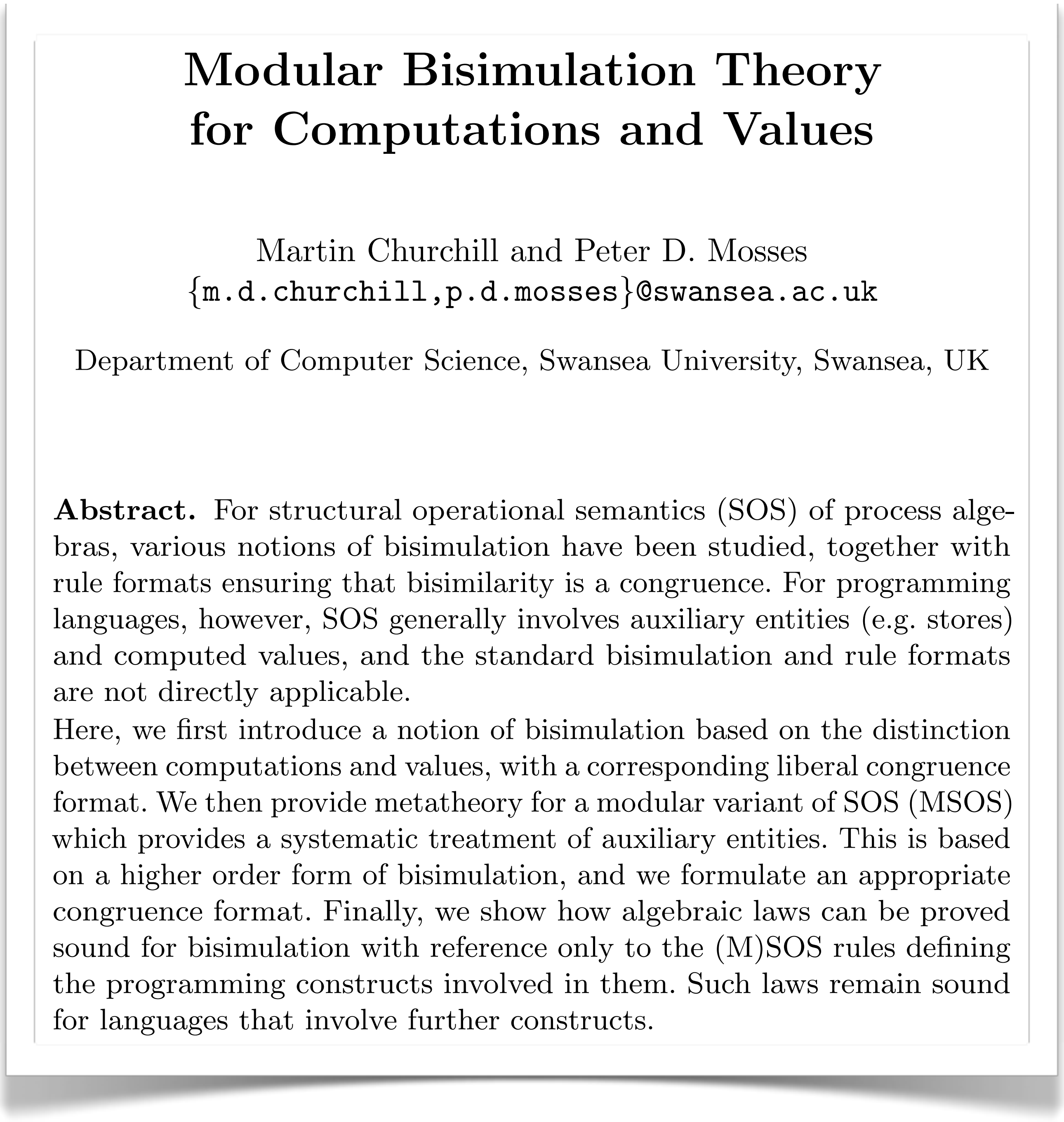
Peter D. Mosses[1] Mark J. New[2]

Modular foundations

FoSSaCS'13:

- bisimilarity
  congruence
  format

- preservation by
  disjoint extension



Modular Bisimulation Theory
for Computations and Values

Martin Churchill and Peter D. Mosses
{m.d.churchill,p.d.mosses}@swansea.ac.uk

Department of Computer Science, Swansea University, Swansea, UK

Abstract. For structural operational semantics (SOS) of process algebras, various notions of bisimulation have been studied, together with rule formats ensuring that bisimilarity is a congruence. For programming languages, however, SOS generally involves auxiliary entities (e.g. stores) and computed values, and the standard bisimulation and rule formats are not directly applicable.
Here, we first introduce a notion of bisimulation based on the distinction between computations and values, with a corresponding liberal congruence format. We then provide metatheory for a modular variant of SOS (MSOS) which provides a systematic treatment of auxiliary entities. This is based on a higher order form of bisimulation, and we formulate an appropriate congruence format. Finally, we show how algebraic laws can be proved sound for bisimulation with reference only to the (M)SOS rules defining the programming constructs involved in them. Such laws remain sound for languages that involve further constructs.

Fundamental constructs (funcons)

Funcons ***normally compute values***

- values compute themselves

Funcon computations may also:

- ***terminate abruptly***
  - signalling some value as the reason
  - failure is a special case
- ***never terminate***
- ***have effects***

Values

Universe

- ***primitive*** (booleans, numbers, characters, symbols)
- ***composite*** (sequences, maps, sets, variants)
- ***types*** (names for sets of values)
- ***abstractions*** (encapsulating funcons)

New types of values are defined in terms of old ones

Funcon ‘aspects’

(Mostly) independent concerns

- control flow
- data flow
- binding
- storing
- interacting

each funcon has a primary ‘aspect’

Sorts of funcons

Notation

- ***commands***
  - *C* : computes ( )
- ***declarations***
  - *D* : computes environments (mapping ids *I* to values *V*)
- ***expressions***
  - *E* : computes values

Generic funcons

- *X* : could be commands, declarations, expressions

Control flow

Normal

- **seq**$(X_1, \ldots)$
  - left to right sequencing
  - concatenates computed values
- **null** is the empty sequence ( )
  - unit for **seq**(X_1, X_2)

Control flow

Conditional

- **if-true-else**(E, X_1, X_2)
  - E has to be boolean-valued
- **while-true**(E, C)
  - doesn't handle break or continue

Call

- **enact**(E)
  - evaluates E to an abstraction value **abs**(X)
  - executes X

Control flow

Alternatives

- **either**($X_1, \ldots$)
  - unordered alternatives
- **else**($X_1, \ldots$)
  - left to right alternatives
- **fail**
  - unit for **either**(X_1, X_2) and **else**(X_1, X_2)
- **when-true**(E, X), **check-true**(E)
  - **fail** when E false

Data flow

Lifting operations

- value operations $F(V_1, \ldots)$ lift to funcons ***F***$(E_1, \ldots)$
  - argument evaluation implicitly *interleaved*
  - ***F***(**seq**$(E_1, \ldots)$) ensures *left to right* evaluation

    e.g.: **not**(**is-equal**(**seq**(E_1, E_2)))

Discarding values

- **effect**(X)
  - executes X, but computes ( )

25

Control and data flow

Giving

- **give-val**(E, X)
  - first evaluates E to a value V
  - then executes X, with the funcon **given** referring to V
- **given**

Application

- **apply**(E_1, E_2)
  - evaluates E_1 to an abstraction **abs**(X), and evaluates E_2 to a value V
  - then executes X, with the funcon **given** referring to V

Control and data flow

Exception handling

- **handle-thrown**(X_1, X_2)
  - try to handle abrupt termination of X_1 by giving the thrown value to the execution of X_2
- **throw-val**(E)
  - terminates abruptly, throwing the value of E

Continuations

- see the paper by Neil Sculthorpe et al. at the ETAPS 2015 *Workshop on Continuations*

Binding

Scopes

- **scope**(*D*, *X*)
  - localises the bindings computed by *D* to *X*
- **bind-val**(*I*, *E*)
  - computes the binding of the id *I* to the value of *E*
- **bound-val**(*I*)
  - inspects the current binding of the id *I*

Binding

Scopes

- **override**(D_1, D_2)
- **unite**(D_1, D_2)
- **accumulate**(D_1, D_2)
- **recursive**($Iset, D$)
  - various ways of composing declarations

Binding

Scopes in abstractions

- **close**(*E*)
  - evaluates *E* to an abstraction **abs**(*X*)
  - returns the *closure* incorporating the current bindings

Patterns

- *simple:* abstractions **abs**(*D*)
- *composite:* formed using value *constructors*
  - structure (and any immutable components) required to be identical when matching

Binding

Pattern matching

- **match-val**(E_1, E_2)
  - evaluates E_1 to a pattern P and E_2 to a value V
  - matching P to V computes bindings
- **case**(E, X)
  - evaluates E to a pattern P,
    then matches P to a given value
  - the scope of the computed bindings is X
  - equivalent to **scope**(**match**(E, **given**), X)

Storing

Variables

- *simple:* representing independent storage locations
  - for storing values of a fixed type
  - monolithic update
- *composite:* formed using value *constructors*
  - component variables can be independently updated
  - structure (and any *immutable* components) required to be identical when updating

32

Storing

Variable allocation

- **alloc**(E_1, E_2)
  - evaluates E_1 to a type T, and E_2 to a value V
  - allocates a simple or composite variable for storing values of type T
  - assigns V to the variable
- **release**(E)
  - evaluates E to a variable
  - terminates the allocation of the variable

Storing

General assignment

- **assign**(E_1, E_2)
  - evaluates E_1 to V_1, and E_2 to V_2
  - when V_1 and V_2 have the same structure, updates the stored values of any simple variables in V_1 by the corresponding component values of V_2
- **current-val**(E)
  - evaluates E to V
  - gives the value formed by replacing any simple variables in V by their stored values

A component reuse example

Language construct:

stm ::= **`while`**`(`*exp*`)` *stm*

Translation to funcons:

exec ⟦ **`while`**`(`*E*`)`*S* ⟧ =

while-true(current-val(*eval* ⟦ *E* ⟧), *exec* ⟦ *S* ⟧)

For languages with break statements:

exec ⟦ **`while`**`(`*E*`)`*S* ⟧ =

handle-thrown(
  while-true(current-val(*eval* ⟦ *E* ⟧), *exec* ⟦ *S* ⟧),
  case(‘break’, null))

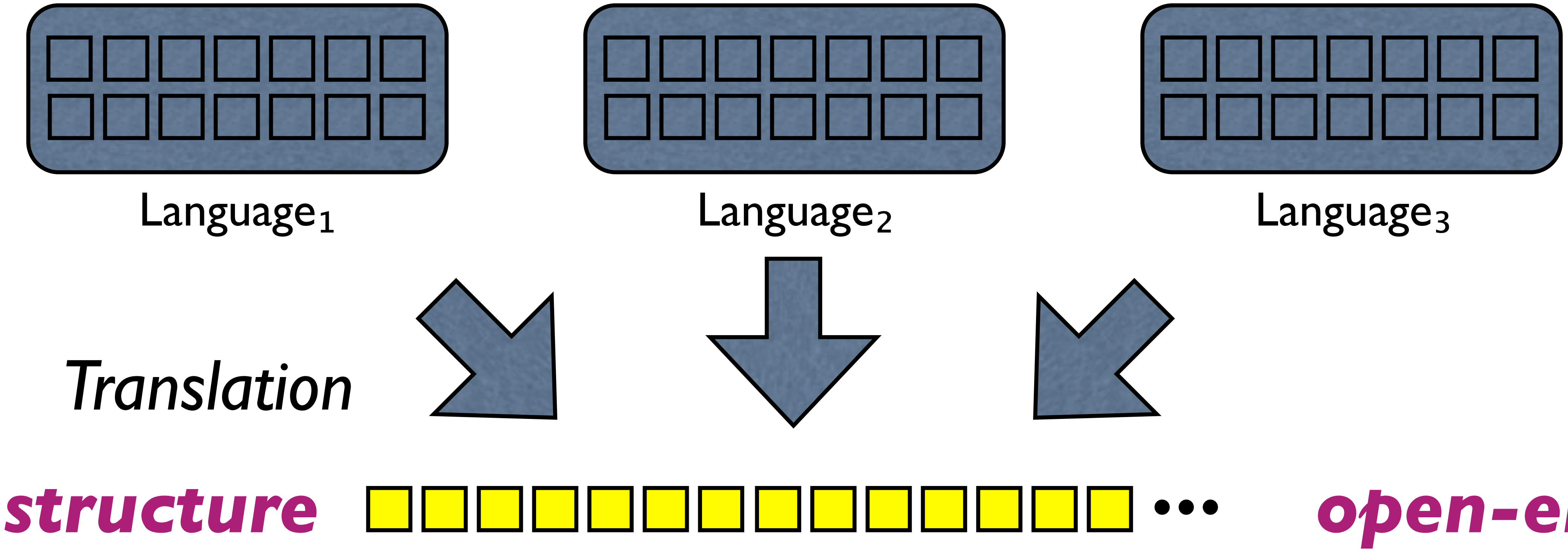
35

High modularity

Component-based semantics

Reusable components of language definitions

- ***fundamental*** programming constructs



Flat structure ***open-ended***

Moderated – no versioning!

Component specification

SOS: Structural operational semantics

$$(X, \rho, \sigma, \ldots) \xrightarrow{\alpha} (X', \rho, \sigma', \ldots) \xrightarrow{\alpha'} (X'', \rho, \sigma'', \ldots)$$

Plotkin (1981)

- ***(optionally-)labelled*** transition relations
- ***states:*** include programs X, environments ρ, stores σ, …
  - environments preserved by $\rho \vdash (\ldots) \rightarrow (\ldots)$
- ***labels:*** simple synchronisation actions α

MSOS: Modular SOS

$$X \xrightarrow{(\rho, \sigma, \sigma', \alpha', \ldots)} X' \xrightarrow{(\rho, \sigma', \sigma'', \alpha'', \ldots)} X''$$

M (1999)

- ***arrow-labelled*** transition relations
- ***states:*** simple programs X, computed values V
- ***labels:*** include environments ρ, stores σ, actions α, …
  - adjacent labels required to be ***composable***
    - fixed environment (ρ)
    - store updates (σ, σ')

Component specification

Notation

if-true-else(E : computes(booleans), X_1, X_2 : computes(T)) : computes(T)

Static semantics

$$\frac{E : \text{booleans} \qquad X_1 : T \qquad X_2 : T}{\textbf{if-true-else}(E, X_1, X_2) : T}$$

Dynamic semantics

$$\frac{E \rightarrow E'}{\textbf{if-true-else}(E, X_1, X_2) \rightarrow \textbf{if-true-else}(E', X_1, X_2)}$$

$$\textbf{if-true-else}(\text{true}, X_1, X_2) \rightarrow X_1$$

$$\textbf{if-true-else}(\text{false}, X_1, X_2) \rightarrow X_2$$

Component specification

Notation

if-true-else(E : computes(booleans), X_1, X_2 : computes(T)) : computes(T)

Static semantics

$$\frac{E : \text{booleans} \qquad X_1 : T \qquad X_2 : T}{\textbf{if-true-else}(E, X_1, X_2) : T}$$

Dynamic semantics

$$\frac{E \rightarrow E'}{\textbf{if-true-else}(E, X_1, X_2) \rightarrow \textbf{if-true-else}(E', X_1, X_2)}$$

$$\textbf{if-true-else}(\text{true}, X_1, X_2) \rightarrow X_1$$

$$\textbf{if-true-else}(\text{false}, X_1, X_2) \rightarrow X_2$$

Component specification

Notation

$$\textbf{scope}(\ \text{computes}(\text{envs}), \text{computes}(T)\ ) : \text{computes}(T)$$

Static semantics

$$\frac{\text{env}(\rho) \vdash D : \rho' \qquad \text{env}(\rho'/\rho) \vdash X : T}{\text{env}(\rho) \vdash \textbf{scope}(D, X) : T}$$

Dynamic semantics

$$\frac{D \to D'}{\textbf{scope}(D, X) \to \textbf{scope}(D', X)} \qquad \frac{\text{env}(\rho'/\rho) \vdash X \to X'}{\text{env}(\rho) \vdash \textbf{scope}(\rho', X) \to \textbf{scope}(\rho', X')}$$

$$\textbf{scope}(\rho, V) \to V$$

43

Component specification

Notation

$\mathbf{scope}(\ \text{computes}(\text{envs}), \text{computes}(T)\ ) : \text{computes}(T)$

Static semantics

$$\frac{\text{env}(\rho) \vdash D : \rho' \qquad \text{env}(\rho'/\rho) \vdash X : T}{\text{env}(\rho) \vdash \mathbf{scope}(D, X) : T}$$

Dynamic semantics

$$\frac{D \to D'}{\mathbf{scope}(D, X) \to \mathbf{scope}(D', X)}$$

$$\frac{\text{env}(\rho'/\rho) \vdash X \to X'}{\text{env}(\rho) \vdash \mathbf{scope}(\rho', X) \to \mathbf{scope}(\rho', X')}$$

$\mathbf{scope}(\rho, V) \to V$

Tool support

Preliminary tool support

SPOOFAX/ECLIPSE

- parsing programs (SDF3, disambiguation, AST creation)
- translating ASTs to funcon terms (SDF3, STRATEGO)
- browsing and editing component-based specifications (SDF3, NABL, STRATEGO)

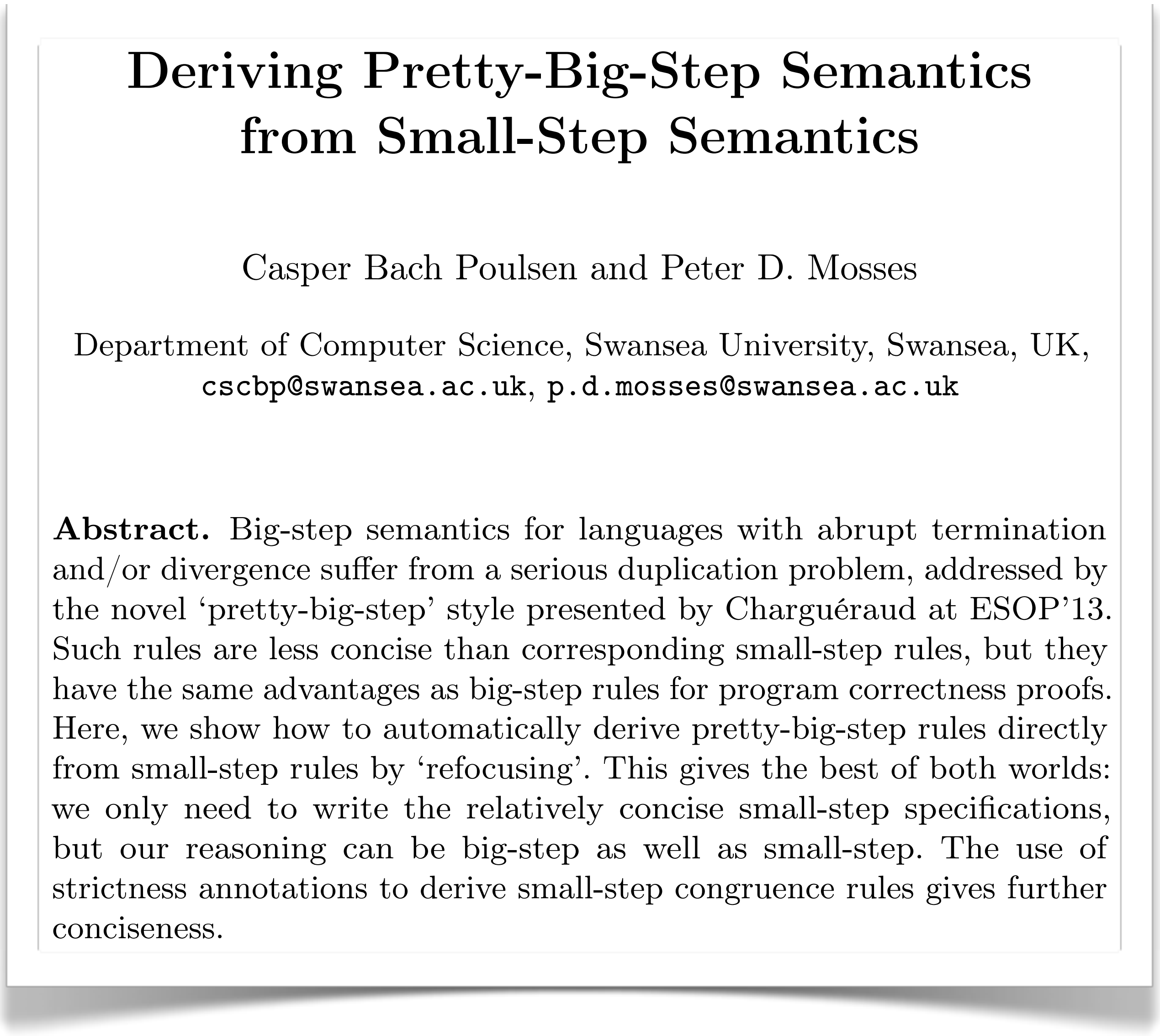
PROLOG

- translating MSOS rules for funcons to PROLOG
  - *currently migrating to STRATEGO*
- running funcon terms

Future tool support

ESOP'14:

- refocusing small-step (M)SOS rules



Deriving Pretty-Big-Step Semantics from Small-Step Semantics

Casper Bach Poulsen and Peter D. Mosses

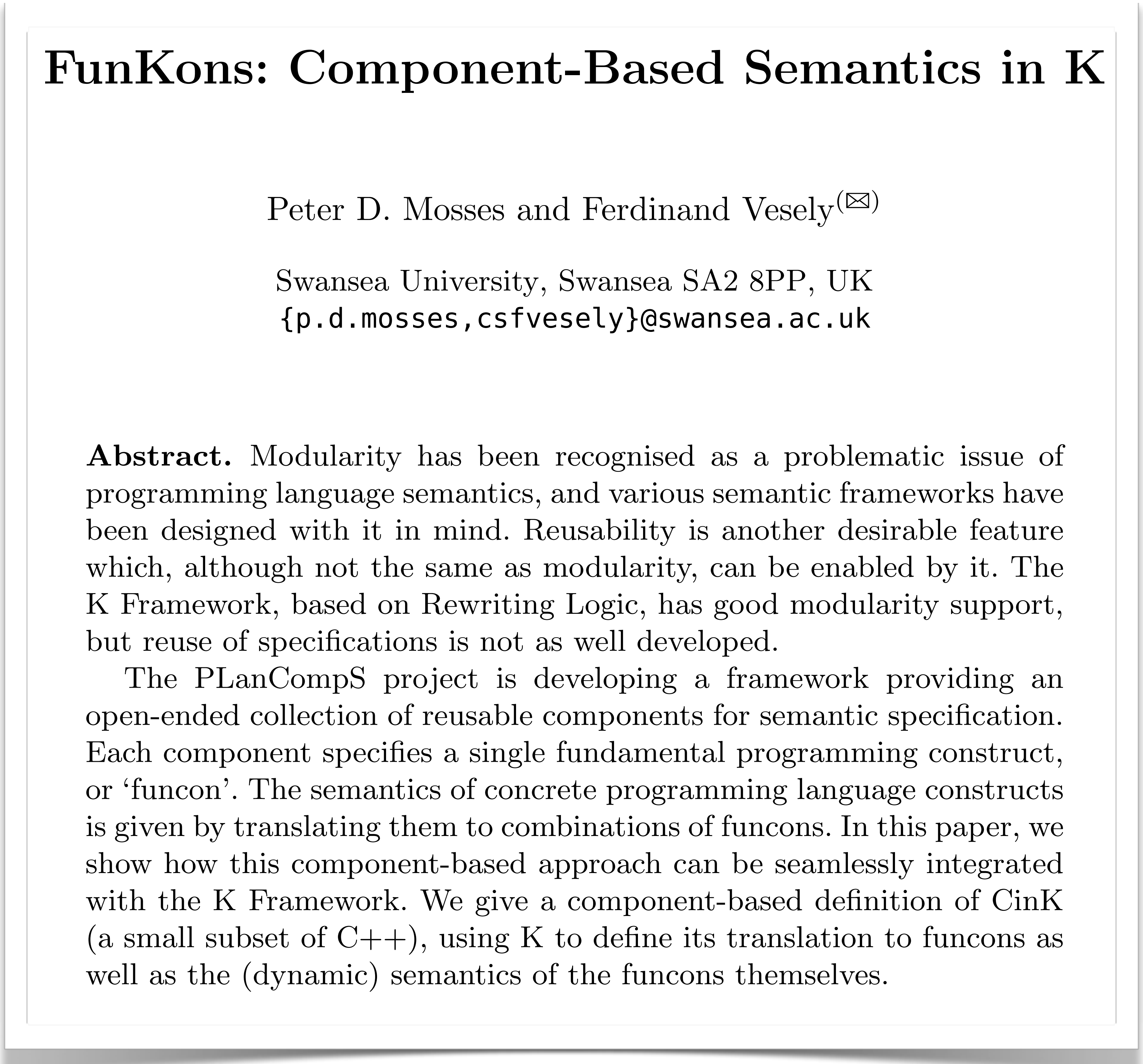
Department of Computer Science, Swansea University, Swansea, UK, cscbp@swansea.ac.uk, p.d.mosses@swansea.ac.uk

Abstract. Big-step semantics for languages with abrupt termination and/or divergence suffer from a serious duplication problem, addressed by the novel 'pretty-big-step' style presented by Charguéraud at ESOP'13. Such rules are less concise than corresponding small-step rules, but they have the same advantages as big-step rules for program correctness proofs. Here, we show how to automatically derive pretty-big-step rules directly from small-step rules by 'refocusing'. This gives the best of both worlds: we only need to write the relatively concise small-step specifications, but our reasoning can be big-step as well as small-step. The use of strictness annotations to derive small-step congruence rules gives further conciseness.

Alternative tool support

WRLA'14:

- using the K framework and tools



FunKons: Component-Based Semantics in K

Peter D. Mosses and Ferdinand Vesely(✉)

Swansea University, Swansea SA2 8PP, UK
{p.d.mosses,csfvesely}@swansea.ac.uk

Abstract. Modularity has been recognised as a problematic issue of programming language semantics, and various semantic frameworks have been designed with it in mind. Reusability is another desirable feature which, although not the same as modularity, can be enabled by it. The K Framework, based on Rewriting Logic, has good modularity support, but reuse of specifications is not as well developed.

The PLanCompS project is developing a framework providing an open-ended collection of reusable components for semantic specification. Each component specifies a single fundamental programming construct, or 'funcon'. The semantics of concrete programming language constructs is given by translating them to combinations of funcons. In this paper, we show how this component-based approach can be seamlessly integrated with the K Framework. We give a component-based definition of CinK (a small subset of C++), using K to define its translation to funcons as well as the (dynamic) semantics of the funcons themselves.

PLANCOMPS project (2011-2015)

Foundations

- component-based semantics [Swansea]
- GLL parsing, disambiguation [RHUL]

Case studies

- CAML LIGHT, C#, JAVA [Swansea]

Tool support

- IDE, funcon interpreter/compiler [RHUL, Swansea]

Digital library

- interface [City], historic documents [Newcastle]

Conclusion

Reusable components

to reduce the ***initial*** effort

High modularity

to reduce the effort of ***change***

Tool support

to reduce the effort of ***getting it right!***

Fundamental constructs:
The Unix philosophy for semantics
of programming languages