Fundamental Constructs in Programming Languages

Peter D. Mosses[1,2][0000−0002−5826−7520]

[1] Delft University of Technology, The Netherlands
[2] Swansea University, United Kingdom
p.d.mosses@swansea.ac.uk

Abstract. When a new programming language appears, the syntax and intended behaviour of its programs need to be specified. The behaviour of each language construct can be concisely specified by translating it to fundamental constructs (funcons), compositionally. In contrast to the informal explanations commonly found in reference manuals, such formal specifications of translations to funcons can be precise and complete. They are also easy to write and read, and to update when the language evolves.
The PLanCompS project has developed a large collection of funcons. Each funcon is defined independently, using a modular variant of structural operational semantics. The definitions are available online, along with tools for generating funcon interpreters from them.
This paper introduces and motivates funcons. It illustrates translation of language constructs to funcons, and funcon definition. It also relates funcons to the notation used in some previous language specification frameworks, including monadic semantics and action semantics.

Keywords: Funcons · Programming constructs · Formal specification

1 Introduction

Many constructs found in (high-level) programming languages combine several behavioural features. For example, call-by-value parameter passing in an imperative language involves order of evaluation, allocating storage and initialising its contents, local name binding, and lexical scoping. Such language constructs generally provide conciseness and clarity in programs, and may support efficient implementation techniques (e.g., stack-based storage); but their full behaviour can be quite difficult to understand, and tedious to specify directly.

Moreover, constructs in different languages may look the same but have very different behaviour (e.g., the notorious '`x=y`'), or look different but have exactly the same behaviour (e.g., '`while...do...`' and '`while(...){...}`'). Relatively minor differences between similar language constructs in different languages include order of evaluation in expressions, and the effect of arithmetic overflow. The evolution of programming languages has resulted in a huge diversity of language constructs and their variants. Some of the constructs are quite simple, but no programming *lingua franca* has emerged.

The PLanCompS project[3] has developed a large collection of *fundamental constructs* ('funcons') from which the behaviour of many high-level programming language constructs can be composed [26]. The behaviour of a complete programming language can be concisely specified by translating all its constructs, compositionally, to funcons. In contrast to the informal explanations commonly found in language reference manuals, formal specifications of translations to funcons can be precise and complete. They are also easy to write and read, and to update when the language evolves. This could make them especially useful during design and development of domain-specific languages.

Funcons are significantly simpler than typical language constructs, in general:

- Each funcon affects only a single behavioural feature, such as flow of control or data, name binding, storing, or interacting.
- Variants of funcon behaviour (e.g., evaluating their arguments in a different order) can be expressed by composite funcon terms.
- The funcons abstract from details related to implementation efficiency.

Funcon behaviour is defined using a modular variant [19,23] of small-step structural operational semantics [27], based on value-computation transition systems [7]. Any program behaviour that can be modelled by a labelled transition system can, in principle, also be specified by composing appropriately-defined funcons. Thus specification by translation to funcons does not, per se, restrict the features of specified languages. When the translation of a particular language construct is excessively complicated, however, its design may be questionable.

The definition of a funcon determines its *name*, its *signature*, and its *behaviour*. Each funcon name should have a *unique* definition, so that it always refers to the same signature and behaviour, regardless of where the reference occurs. To support reuse, funcon definitions need to be *fixed* and *permanent*: changing or removing funcons would undermine the validity of translations that use them. In particular, adding a new funcon to a collection must never require changes to the definitions already in it.

Version control is superfluous for funcons; translations of language constructs to funcons, in contrast, may need to change when the specified language evolves. For example, the illustrative language Imp includes a plain old while-loop with a Boolean-valued condition: '`while(`*BExp*`)` *Block*'. The following rule translates it to the funcon while-true, which has exactly the required behaviour:

Rule execute⟦ '`while`' '`(`' *BExp* '`)`' *Block* ⟧ =
while-true(eval-bool⟦ *BExp* ⟧, execute⟦ *Block* ⟧)

The behaviour of the funcon while-true is fixed. But suppose the Imp language evolves, and a *Block* can now execute a statement '`break;`', which is supposed to terminate just the *closest* enclosing while-loop. We can extend the translation with the following rule:

Rule execute⟦ '`break`' '`;`' ⟧ = abrupt(broken)

[3] https://plancomps.github.io

The translation of '`while(true){break;}`' is while-true(true, abrupt(broken)). The funcon abrupt(V) terminates execution abruptly, signalling its argument value V as the reason for termination. However, the behaviour of while-true(true, X) is to terminate abruptly whenever X does – so this translation would lead to abrupt termination of *all* enclosing while-loops!

We cannot change the definition of while-true, so we are forced to change the translation rule. The following updated translation rule reflects the extension of the behaviour of while-loops with the intended handling of abrupt termination due to break-statements, and that they propagate abrupt termination for any other reason:

Rule execute⟦ '`while`' '`(`' *BExp* '`)`' *Block* ⟧ =
handle-abrupt(
while-true(eval-bool⟦ *BExp* ⟧, execute⟦ *Block* ⟧),
if-true-else(is-equal(given, broken), null-value, abrupt(given)))

Computing null-value represents normal termination; given refers to the reason for the abrupt termination.

The specialised funcon handle-break can be used to specify the same behaviour more concisely:

Rule execute⟦ '`while`' '`(`' *BExp* '`)`' *Block* ⟧ =
handle-break(while-true(eval-bool⟦ *BExp* ⟧, execute⟦ *Block* ⟧))

Wrapping execute⟦ *Block* ⟧ in handle-continue would also support abrupt termination of the current *iteration* due to executing a continue-statement.

Overview. The reader is assumed to be interested in programming languages, and familiar with their main concepts. The research on which this paper is based has been published elsewhere [4,7,8,19,21,22,23]. The main aims here are to motivate the general idea of funcons, and illustrate how they can be used to specify the behaviour of programming language constructs.

- Section 2 explains some general features of funcons.
- Section 3 considers how to manage large collections of funcons.
- Section 4 analyses various facets of funcon behaviour.
- Section 5 illustrates specification of translation of language constructs to funcons, and explains how to validate such translations.
- Section 6 illustrates how to define funcons independently.
- Section 7 relates funcons to the auxiliary operations defined in denotational semantics, to monads, and to the combinators used in action semantics.
- Section 8 concludes with plans for future development of funcons,
- Appendices A–G give an informal summary of the currently defined funcons.[4]

The rest of this paper is structured as responses to questions that readers might ask about funcons. The author welcomes further questions, as well as comments on the given responses.

[4] At the time of writing, the collection has not yet been released, and could change.

2 The Nature of Funcons

Let us start by explaining some general features of funcons.

– *What aspects of behaviour do funcons represent?*

Funcons abstract from details related to implementation efficiency, such as storage allocation algorithms and communication protocols. They express implementation independent behaviour that arises when programs are executed. They also express linguistic features on which that behaviour depends, such as scopes of bindings.

– *Can funcons be implemented efficiently?*

Funcons need to be executable, to support validation of language translations. Their current implementation uses HASKELL interpreters generated directly from their definitions [5]. The efficiency of evaluating funcon terms is adequate for running unit tests and typical test programs, but not applications. However, it should be possible to implement certain sets of funcons more efficiently, e.g., using virtual machines that support just the required features, or by optimised compilation of funcon terms to other languages.

– *How complicated are funcons ?*

One might expect that funcons should be as simple as possible. In fact the aim is for funcons to be not too complicated, but not *too* simple – just right! In the physical sciences, molecules are characterised and understood primarily in terms of chemical bonds between their constituent atoms, and atoms are formed from protons, neutrons, and electrons; protons and neutrons are themselves composed from sub-atomic particles, such as quarks. To explain a molecule in terms of sub-atomic particles might be possible, but unhelpful. Language constructs are somewhat analogous to molecules, and funcons to atoms.

Introducing a funcon that corresponds directly to a complicated language construct would make the funcon analysis of that language construct trivial, but a direct definition of the funcon behaviour would necessarily be complicated. At the other extreme, taking pure function abstraction and application as the only funcons would make analysis and specification of language constructs as complicated as in (pre-monadic) denotational semantics.

Funcons aim to be unbiased towards any family of languages. Adding an intermediate layer of not-so-fundamental constructs that are closely related to some particular language constructs is thus undesirable. However, it is sometimes appropriate to define funcons that abbreviate particular compositions of other funcons. Section 1 mentioned handle-break, which handles abrupt termination caused only by abrupt(broken) in X; it abbreviates handle-abrupt(X, Y) where Y involves an explicit test whether a given signal is the value broken. Similarly, the funcon allocate-initialised-variable(T, V) abbreviates the sequential composition of allocate-variable(T) and initialise-variable$(_, V)$.

– *Can funcons have alternative behaviours?*

No, never. The behaviour of common *language* constructs, such as assignment expressions, often varies significantly between different languages. For example, the order of evaluation of the two sides of an assignment expression is left to right in some languages, right to left in others, or may even be implementation-dependent; and the result may be the target variable or the assigned value. The *funcon* for assignment needs to have a behaviour from which all those variations can be obtained by composition with other funcons.

– *Are funcons independent?*

Funcons are often independent, but not always. For instance, the definition of the funcon while-true specifies the reduction of while-true(B, X) to a term involving the funcons if-true-else and sequential:

$$\textit{Funcon}\quad \mathsf{while\text{-}true}(B :\Rightarrow \mathsf{booleans}, X :\Rightarrow \mathsf{null\text{-}type}) :\Rightarrow \mathsf{null\text{-}type}$$
$$\rightsquigarrow \mathsf{if\text{-}true\text{-}else}(B, \mathsf{sequential}(X, \mathsf{while\text{-}true}(B, X)), \mathsf{null\text{-}value})$$

Duplication of B before starting to evaluate it is essential, in case it needs to be re-evaluated after the execution of X. We could introduce an auxiliary term constructor for that, but it is simpler to make use of if-true-else and sequential.

– *Do features of funcons interact?*

No. Feature interactions in software development tend to arise when requirement specifications are incomplete. An example of feature interaction in [2] involves a flood prevention system that turns off the water supply, and a sprinkler system that depends on that water; the requirements regarding flood prevention had better include checking the safety of turning the water off...

The complete requirement for each funcon is to provide just the behaviour specified in its definition, propagating all *unmentioned* effects of evaluating its arguments. The values of the arguments are required to be consistent with the types in the funcon signature, but *no* further requirements arise when combining funcons.

– *Can funcons be used as a programming language?*

Composing funcons is similar to the original idea of UNIX: plugging simple commands together to produce complex behaviour.[5] Not-so-fundamental constructs could be defined as abbreviations for frequently-needed funcon compositions; a coating of 'syntactic sugar' would be needed to avoid an unwelcome plethora of parentheses in larger funcon compositions.

The main drawback of programming directly with funcons would be the comparatively low efficiency of their current implementation, which uses interpreters written in HASKELL, generated directly from funcon definitions.

[5] Nowadays, a UNIX command often has a multitude of obscure options, documented in a manual 'page' that fills many screens.

– *Can funcons be higher-order?*

Funcons can represent higher-order functions as values, but funcons are not themselves higher-order: they do not take (non-constant) funcons as arguments. However, it is easy to define funcons for common idioms of higher-order programming (maps, filters, folds, etc.).

– *Do funcons have algebraic properties?*

Yes: many binary funcons are associative, with left and right units; some are also commutative. These properties hold for a notion of bisimulation for the value-computation transition systems [7] that provide the foundations for funcon definitions. This bisimulation is preserved when new funcons are added. Funcon terms are written as applicative expressions, and associativity allows binary funcons to be extended to longer sequences of arguments.

– *Can I use my favourite proof assistant to prove properties of funcons?*

Some years ago, the modular variant [19] of structural operational semantics used to define funcons was implemented in the Coq proof assistant, and modular proofs of some properties were carried out [14]. In a related line of work [33], a different method for modular proofs in Coq was developed.

Modular proofs depend only on the definitions of the funcons involved, and remain sound when funcon definitions are combined. In principle, they could be released together with the funcons.

3 Collections of Funcons

The current collection of funcons is called Funcons-beta. It includes several hundred funcons. Management of such a collection is non-trivial.

– *How can we classify funcons?*

Most high-level programming languages distinguish syntactically between commands (a.k.a. statements), declarations, and expressions. Commands may assign to variables; declarations bind names; and expressions compute values. However, such syntactic distinctions are not universal: for instance, expressions sometimes subsume commands, and sequences of commands may include declarations. Grammars for programming language syntax (abstract as well as concrete) often introduce many further syntactic distinctions. A universal set of syntactic sorts that encompasses all programming languages is not available.

For funcons, we have a single syntactic sort of *terms*, with *values* as a subsort. A funcon term is similar to an expression in an (impure) functional programming language: it computes values of a specific (possibly generic) type. A funcon term corresponding to a command computes a fixed null value, and a term corresponding to a declaration computes a value environment, mapping names to values.

We may also classify funcons according to their effects. The behaviour of many funcons involves *auxiliary entities*, representing various kinds of effects. For instance, funcons for name binding use an auxiliary environment entity to represent the current bindings; funcons for imperative variables use an auxiliary store entity to represent the currently assigned values of variables.

– *How do we refer to a particular funcon in a collection?*

A collection of funcons is like an *open* package: the names of all the funcons are visible externally (except those marked as auxiliary). Neither the classification of funcons nor the paths to their definitions affects references to funcon names.

The name of each funcon should clearly suggest its behaviour, to support casual reading of funcon terms and the potential use of funcons as a controlled vocabulary for informal discussion and comparison of programming languages. Type names are *plural* words (e.g., lists).[6] When a funcon corresponds directly to a familiar concept, a single well-chosen word can be adequate, but otherwise several words (joined by hyphens) may be needed. Moreover, different datatypes may have closely related operations, yet the names for the corresponding funcons have to be distinct, due to the absence of overloading: the name of the datatype can be added as a prefix of the name, e.g., integer-add.[7]

Suggestive names can be quite long, and abbreviations may be needed in some situations (e.g., classrooms, examinations, presentations). Abbreviations can be defined as explicit aliases for funcons; for instance, alloc-init is defined as an alias for allocate-initialised-variable.

– *Do funcons evolve?*

After a collection of funcons has been released, the behaviour of all the funcons in it needs to be *fixed* and *permanent*, since changes could affect or break their uses in language translations (which do not need to be public or registered). All uses of a particular funcon name thus refer to the same behaviour.

However, the collection itself *can* evolve: by extension with new funcons. This must not require changes to the definitions of the previous funcons. New funcons need to be carefully checked and tested before they are added, since their definitions cannot be revoked.

Names of funcons always refer implicitly to the current version of a collection. Evolving collections of funcons have no need for version *numbers*, since once a funcon has been defined, adding definitions for new funcons (or an alias for an already defined funcon) cannot invalidate existing references to names.

– *Will the* Funcons-beta *collection of funcons ever be finalised?*

Funcons-beta is a release *candidate*. After further polishing, review, and use in language specifications, the collection of funcons and their documentation are to be released for general use. However, it will always be possible to add new funcons to the collection, so as to support new concepts or provide alternative ways of expressing existing concepts.

[6] Singular forms of type names are used as value constructors.

[7] Currently, Funcons-beta does not support namespaces in collections of funcons.

4 Facets of Funcons

When funcon terms are evaluated, their behaviour may have many *facets*: apart from computing values, funcon behaviour can involve name bindings, imperative variables, abrupt termination, interaction, etc. Facets that are not needed for a particular term can be ignored.

In this section, we introduce the main facets of the FUNCONS-BETA collection. Appendices A–G provide an informal summary of the funcons; their definitions are available online [26].

– *How are funcon terms evaluated?*

Evaluation of a funcon term may terminate *normally*, *abruptly*, or *never*. The evaluation takes a sequence of argument *terms*; on normal termination, it computes a sequence of *values* (where a sequence of length 1 is identified with its only element). The funcon signature specifies how many arguments it takes, the type of values to be computed by each argument, and the types of values that the funcon computes. Individual arguments may be required to be *pre-computed* values; the funcon definition specifies how its behaviour combines the computations of any remaining arguments.

– *Does each funcon take a fixed number of arguments?*

Not necessarily: a funcon signature can specify that an argument at some position is optional, or that it can be a sequence. Sequence arguments are often used to extend associative binary funcons to longer argument sequences. They are also used for funcons that correspond directly to conventional notation for (finite) lists and sets, e.g., $\mathsf{list}(V_1, \ldots, V_n)$ for $[V_1, \ldots, V_n]$.

– *How do funcons represent data?*

Data that programs process when executed is represented by funcon terms classified as values. Some funcons are value constructors: they are *inert*, and have no computational behaviour themselves. Values are themselves classified as *primitive values*, *composite values*, or (procedural) *abstractions*.

Conceptually, primitive values are atomic, and not regarded as constructed from other values. Booleans, unbounded integers, IEEE floats, UNICODE characters, and a null value are all classified as primitive. Some of them have constant constructors; the rest are computed by built-in funcons.

Composite values are constructed from finite sequences of argument values. Value constructors are injective: different argument value sequences give different composite values. The types of composite values include parametrised algebraic data types, with a generic representation. Various algebraic datatypes are predefined, and new ones can be introduced. Composite values include also built-in parametrised types of sets, maps, multi-sets, and graphs.

Abstractions are values formed by the value constructor $\mathsf{abstraction}(X)$ with an *unevaluated* argument X. Values are called *ground* when they are constructed entirely from primitive and composite values, without any abstraction values.

Appendix A summarises the funcons for types of data, and some funcons for data operations.

– *What kind of behaviour do funcons for data operations have?*

Data operations in programs are generally represented by funcons whose only behaviour is to compute values from *pre-evaluated* arguments. The arguments are evaluated in *any* order, possibly with interleaving (the order of argument evaluation is irrelevant when the evaluations have no effects). *Partial* data operations (e.g., integer division, or selecting the head of a list) compute the empty sequence when their arguments are not in their domain of definition.

Value *types* are themselves values, so funcons can take types as arguments and give them as results. Apart from supporting dependent types, this generality is needed to represent ordinary type constructors as funcons (e.g., lists(T), where T is the type of the list elements).

– *How do funcons express normal flow of control?*

A funcon intended purely for specifying normal control flow generally specifies the potential order of evaluation of its arguments, but does not otherwise contribute to behaviour. Such funcons include sequential or interleaved command execution and expression evaluation, deterministic and non-deterministic choice between computations, and command iteration.

Appendix B summarises the funcons for representing control flow.

– *How do funcons express flow of data?*

A computation may involve multiple uses of the same data (e.g., so as to assign it to a variable as well as provide it as a result). It may also involve repeating the same computation with different data. The computations of funcons for specifying such data flow involve an auxiliary entity given-value(V) that can be set to a computed value V; the funcon given gets the current value.

Appendix C summarises the funcons for representing data flow.

– *How do funcons specify scopes of bindings?*

An occurrence of a name in a program either *binds* the name, or *references* whatever is currently bound to the name. Binding occurrences are usually found in declarations, parameter specifications, and patterns; references to names are ubiquitous. Sequences of declarations have the effect of successively extending (or perhaps overriding) the current bindings with the bindings due to the individual declarations.

Funcons use conventional environments ρ (mapping names to values) to represent both the current bindings and bindings computed by declarations. The auxiliary entity environment(ρ) represents the current bindings; the current binding for an individual name I is inspected using the funcon bound-value(I). An environment representing computed bindings is an ordinary composite value, and can be inspected using data operations.

Some languages include various constructs for composing declarations, and these are represented directly by funcons that compute environments. However, the funcons corresponding to recursive declarations represent circularity by creating cut-points called links, which involves a separate auxiliary entity.

Appendix D summarises the funcons for representing name binding.

– *Do funcons have static scopes for bindings?*

The difference between static and dynamic scopes concerns procedural abstraction. A value that represents an abstraction is constructed from an *unevaluated* argument X by the funcon abstraction(X). The abstraction value can be subsequently enacted, which evaluates the argument X – potentially in a different context from that where the abstraction value was constructed.

Constructed abstraction values thus naturally have *dynamic* scopes for bindings. To obtain *static* scopes, the funcon closure(X) computes a closure value: an abstraction whose argument evaluation starts by ignoring the current bindings and (locally) re-declaring the abstraction-time bindings.

– *How do funcons distinguish between constant and mutable variables?*

In programming languages, imperative variables usually have names. It may be tempting to regard variable names as *bound* directly to values: bindings then need to be mutable, assignment to a variable name updates its binding, and constants correspond to single-assignment variables. However, such a simplistic analysis does not easily extend to features such as aliasing and call by reference.

A more satisfactory conceptual basis for imperative variables is to regard them as independent storage *locations*.[8] The declaration of a named variable involves allocation of storage (optionally with an initial value) together with binding the name of the variable to the storage location. Assignment to a named variable then affects what value is stored at the location, but leaves the bindings unchanged. Aliasing can now be understood simply as the simultaneous binding of different names to the same location.

The funcons for imperative variables involve an auxiliary mutable entity store(σ), mapping locations to their currently assigned values. The store supports allocation (and recycling) of locations for values of any type, and their initialisation, assignment, and inspection. It is completely independent of the auxiliary entity environment(ρ) used to represent the current name bindings.

In mathematical logic, a 'variable' corresponds to a name, and 'assignment' to binding. Imperative variables in programming languages are often called 'L-values', with 'R-values' being those that can be assigned to variables.[9] With funcons, all values can be assigned to variables – and variables are themselves values.

– *Can funcons represent structured variables with mixtures of constant and mutable fields?*

A simple variable consists of a location together with the type of values that it can store; assignment checks that the value to be assigned to the variable is in its type.[10] Simple variables may store primitive values (e.g., numbers) or composite

[8] Funcons have not yet been developed for 'relaxed' memory models or data marshalling.

[9] 'L' and 'R' refer to the left and right sides of typical assignment commands [32].

[10] Funcons for using un-typed locations as variables would be slightly simpler.

values (e.g., tuples), but assignment to a simple variable is always *monolithic*: the current value is replaced *entirely* by the new value.

Structured variables are composite values where some components are simple variables. These include hybrids having both mutable and immutable components. Assignment to a selected component variable corresponds to an in-place update; assignment of a composite value to an entire structured variable updates all the component simple variables with the matching values, and checks that the immutable components are the same.

Appendix E summarises the funcons for representing imperative variables.

– *How about abrupt termination?*

Various language constructs may cause abrupt termination when executed: throwing or raising an exception, returning the value of a function, breaking out of a loop, etc. Enclosing constructs can detect particular kinds of abrupt termination, and handle them appropriately. For example, a language construct may inspect an exception value, and conditionally handle it; a function application handles an abruptly returned value by giving it as the result; and a loop handles a break by terminating normally.

Funcons express abrupt termination and handlers *uniformly*. Evaluation of a funcon term may terminate normally, abruptly, or never. Abrupt termination leads to a stuck term, emitting an auxiliary entity abrupted(V) as a *signal* with a value V. The *closest* enclosing funcon that notices the emission of such a signal can inspect its value, and determine whether to handle it or not.

Appendix F summarises the funcons for abrupt termination.

– *Is it possible to define delimited control operators as funcons?*

Somewhat surprisingly, yes: see [26,30]. Control operators include continuation handling functions, such as '`call-cc`'.

– *Can non-terminating funcon evaluation have observable behaviour?*

Yes: through *interactive* input and output.

Program behaviour may depend on, and affect, data stored in files. Conceptually, files can be regarded as (complicated) structured variables: input from a file inspects the value stored at the current position, and advances the position; output to a file appends a value to it. Changes to a file system during program execution correspond to updating values stored in locations; they may subsequently be overwritten, so their *final* values can only be observed on program termination.

Interactive input and output, in contrast, *cannot* be regarded as effects on mutable storage. Acceptance of input data from a stream during program execution is irrevocable, as is output of data to a stream. Interaction may also involve inter-dependence between input and output. And a program that never terminates can have infinitely long streams of input and output.

Thus funcons for expressing interaction involve kinds of entities that differ fundamentally from those we previously introduced. The auxiliary entity standard-in(V^*) represents the (finite) sequence of values *input* at each step of a computation, where the empty sequence ( ) represents that no values are input. The value null-value indicates the end of the input. The auxiliary entity standard-out(V^*) represents the (finite) sequence of values *output* at a particular step, where the empty sequence ( ) represents the lack of output. Computations concatenate the input sequences of each step, and similarly for output – potentially resulting in infinite sequences for non-terminating computations.

Appendix G summarises the funcons for representing interaction. To support multiple streams, further entities and funcons would need to be added.

– *Do funcons currently support specification of any other language features?*

Tentative funcons for *multithreading* have been developed. They have not yet been rigorously unit-tested, nor used much in language definitions. These funcons are not included in FUNCONS-BETA, but in an unstable collection that extends FUNCONS-BETA [26].

The multithreading funcons involve multiple mutable auxiliary entities, representing the collection of threads, the set of active threads, the thread being executed, the values computed by terminated threads, and (abstract) scheduling information. Funcons that combine effects on multiple auxiliary entities are undesirable, and their definitions are somewhat verbose. It is currently unclear whether simpler funcons for multithreading can be developed.

Multithreading also involves *synchronisation*. The funcons for synchronising involve only the store entity. To inhibit preemption during synchronisation, multiple assignments need to be executed atomically, in a single transition.

Funcons for *distributed processes* have not yet been developed. They are expected to be based on asynchronous execution and message passing (cf. [17]).

Funcons for specifying *meta-programming* constructs have been defined [3]; they also enable a straightforward specification of call-by-need parameters.

5 Translation of Language Constructs to Funcons

In this section, we illustrate how a simple programming language construct can be specified by translation to funcons. Specifying such a translation for all constructs of a language defines the behaviour of programs, based on the behaviour of the funcons used in the translation. The PLANCOMPS project has developed some examples [26] and made them available for browsing on a website. We conclude this section with an overview of the examples, and indicate how they have been developed and tested.

– *How is call-by-value translated to funcons?*

The following fragments of a language specification illustrate how call-by-value parameter passing in an imperative programming language can be specified by

translation to funcons. The fragments originate from a published specification [8] of the SIMPLE language; for brevity, however, we here restrict SIMPLE function applications and declarations to a single parameter.

The translation specification in Fig. 1 declares exp as a phrase sort, with the meta-variable *Exp* (possibly with subscripts and/or primes) ranging over phrases of that sort. The BNF-like production shows two language constructs of sort exp: an identifier of sort id (lexical tokens, here assumed to be specified elsewhere with meta-variable *Id*) and a function application written 'Exp_1`(`Exp_2`)`'.

Syntax Exp : exp ::= $\cdots$ | id | exp '`(`' exp '`)`' | $\cdots$

Semantics rval⟦ _ : exp ⟧ : ⇒ values

Rule rval⟦ Id ⟧ = assigned-value(bound-value(id⟦ Id ⟧))

Rule rval⟦ Exp_1 '`(`' Exp_2 '`)`' ⟧ = apply(rval⟦ Exp_1 ⟧, rval⟦ Exp_2 ⟧)

Fig. 1. Translation of identifiers and function applications in SIMPLE to funcons

The translation function rval⟦*Exp*⟧ maps phrases *Exp* of sort exp to funcon terms that compute elements of type values. Translation is compositional: the funcon term for a phrase combines the translations of its sub-phrases. The translation function id⟦*Id*⟧ maps lexical tokens *Id* of sort id to funcon values of type identifiers (its specification is omitted here).

In this illustrative language, the only values to which identifiers can be bound are simple imperative variables. When identifiers can be bound directly to other values (e.g., numbers) we would use current-value instead of assigned-value.

For call-by-value parameters in an imperative language, the argument value can be passed to the called function, which then has to allocate a variable to store the value. For call-by-reference, the argument would have to evaluate to a variable; for call-by-name, the evaluation of the argument would be deferred, which can be expressed by constructing a thunk abstraction value from it. When the mode of parameter-passing in function applications depends on the function, argument evaluation needs to be incorporated in the value that represents the function.

The translation specification for function declarations in Fig. 2 assumes a translation function exec⟦*Block*⟧ for phrases *Block* of sort block. A block is a statement, which normally computes a null value; but here, as in many languages, a block can return an expression value by executing a return statement, which terminates the execution of the block abruptly.

The use of closure ensures static (lexical) bindings for references to names in the function body. For dynamic bindings, we would replace closure by abstraction. The construction of a function value from the closure is needed so that apply can be used to give the argument value to the body of the abstraction.

```
Syntax    Decl : decl ::= ··· | 'function' id '(' id ')' block

Semantics declare⟦ _ : decl ⟧ : ⇒ environments

    Rule  declare⟦ 'function' Id1 '(' Id2 ')' Block ⟧ =
            bind-value(id⟦ Id1 ⟧,
              allocate-initialised-variable(functions(values, values),
                function(closure(
                  scope(
                    bind-value(id⟦ Id2 ⟧,
                      allocate-initialised-variable(values, given)),
                    handle-return(exec⟦ Block ⟧)))))))
```

Fig. 2. Translation of function declarations in SIMPLE to funcons

The scope funcon adds the bindings computed by its first argument to the current bindings for the evaluation of its second argument. In this simplified illustration, functions have only one formal parameter, which is bound to a freshly allocated variable containing the given argument value; for multiple parameters, the given value would be a tuple of the same length, matched by a pattern tuple.

The handle-return funcon concisely handles abrupt termination of the function body arising from evaluation of the return funcon. It has no effect on normal termination, nor on abrupt termination for other reasons.

In languages where function identifiers can be bound directly to function closures, the first allocate-initialised-variable in the translation rule could be eliminated. However, the possibility of recursive function calls would then need to be expressed directly, using the recursive funcon.

The call-by-value example illustrates how directly the behaviour of a language construct can be specified by translation to funcons.

– *Which other language constructs have been translated to funcons?*

The PLANCOMPS project has developed the following language specifications based on FUNCONS-BETA, and made them available for browsing online [26].

– IMP: a very small imperative language, often used in text books on semantics. Its translation to funcons illustrates basic features of the framework.
– SIMPLE: a somewhat larger imperative language than IMP. Its translation to funcons [8] illustrates most features of the framework. It is comparable to the specification of SIMPLE in K [28], except that multithreading is omitted.
– MINIJAVA: a very simple subset of JAVA, used in [1]. Its specification illustrates translation to funcons for classes and objects.
– SL: the SIMPLELANGUAGE used for demonstration of GRAALVM [10]. Its translation to funcons illustrates how dynamic bindings can be specified.

– OCaml Light: a core sublanguage of OCaml. Its specification illustrates how translations to funcons scale up to a medium-sized language.

Further examples of language specifications involve funcons from an unstable collection that extends Funcons-beta [26]:

– Imp++: extends Imp with multithreading and various other features.
– Simple-Threads: adds the previously-omitted multithreading constructs.
– LangDev-2019: demonstrates extensibility of language specifications.

A funcon-based specification of C♯ is currently being developed.

– *How can we check translations of language constructs to funcons?*

Consider our translation of function declarations with call-by-value parameters. Potential mistakes include spelling errors in names (primarily funcons, but also syntax sorts, translation functions, and meta-variables) and misplaced parentheses. The syntax of the language construct in the translation rule might not be consistent with the specified grammar. A less obvious mistake is when the arguments of a funcon could compute values that are not in the types required by the funcon signature. We might also have used a funcon that does not have the intended behaviour (e.g., using abstraction instead of closure).

Clearly, tool support for checking is essential. A workbench for specifying translations of languages to funcons has been developed [21]. Tools for evaluating funcon terms [5] allow us to check whether they have the expected behaviour.

The workbench checks references to names, term formation, and the syntax in translation rules. It checks that funcons have the right number of arguments, but not yet that the arguments compute values of the required types; we currently rely on testing to check for that.

The workbench also supports parsing complete programs and translating them to funcon terms, using parsers and translators generated from the specified grammar and translation rules. It is based on the Spoofax language workbench [12], and implemented using the declarative domain-specific meta-languages Sdf3, NaBL2, and Stratego. See [21] for further details. The tools for evaluating funcon terms [5] are implemented in Haskell, and can be called directly from the workbench.

6 Defining and Implementing Funcons

In this section, we illustrate how to define the behaviour of a funcon, once and for all, using a highly modular variant [19,23] of structural operational semantics [27]. Modularity of funcon definitions is crucial for extensibility of funcon collections.

$$\textit{Funcon}\quad \mathsf{scope}(\_ : \mathsf{environments}, \_ : \Rightarrow T) : \Rightarrow T$$

$$\textit{Rule}\quad \frac{\mathsf{environment}(\mathsf{map\text{-}override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \mathsf{scope}(\rho_1 : \mathsf{environments}, X) \longrightarrow \mathsf{scope}(\rho_1, X')}$$

$$\textit{Rule}\quad \mathsf{scope}(\_ : \mathsf{environments}, V : T) \rightsquigarrow V$$

Fig. 3. Definition of the funcon for expressing scopes of local declrations

– *How are funcons defined?*

The funcon signature in Fig. 3 specifies that scope takes two arguments. The first argument is required to be pre-evaluated to a value of type environments; the second argument should be unevaluated, as indicated by '$\Rightarrow T$'. Values computed by $\mathsf{scope}(\rho_1, X)$ are to have the same type (T) as the values computed by X.

The rules define how evaluation of $\mathsf{scope}(\rho_1, X)$ can proceed when the current bindings are represented by ρ_0. The premise of the first rule holds if X can make a transition to X' when ρ_1 overrides the current bindings ρ_0. Whether X' is a computed value or an intermediate term is irrelevant. When the premise holds, the conclusion is that $\mathsf{scope}(\rho_1, X)$ can make a transition to $\mathsf{scope}(\rho_1, X')$.

If X can terminate abruptly, or continue making transitions forever, then $\mathsf{scope}(\rho_1, X)$ can do the same. The last rule allows evaluation of $\mathsf{scope}(\rho_1, X)$ to terminate normally, computing the same value V as X. Transitions written with '$\rightsquigarrow$' correspond to term rewriting [7], and do not involve auxiliary entities.

– *How can funcon definitions remain fixed when new funcons are added?*

The use of the auxiliary entity $\mathsf{environment}(\rho_0)$ in the definition of scope restricts transitions to states that include it, but states might still include other auxiliary entities, such as $\mathsf{store}(\sigma)$ or $\mathsf{given\text{-}value}(V)$. If a transition $X \longrightarrow X'$ updates σ to σ', so does $\mathsf{scope}(\rho_1, X) \longrightarrow \mathsf{scope}(\rho_1, X')$; the transitions in the premise and conclusion use the same given value; and if $X \longrightarrow X'$ emits a signal on abrupt termination, so does the corresponding transition for $\mathsf{scope}(\rho_1, X)$.

Auxiliary entities are classified according to how they are *propagated*:

Contextual: A contextual entity remains *fixed* for successive steps in the computation of a term, but can be different for the computations of sub-terms.

Mutable: Sequential *changes* to a mutable entity are propagated between the computation of a term and the computations of its sub-terms.

Input: An input entity is a sequence of values *consumed* by evaluating a term, concatenating the sequences consumed by the computations of its sub-terms.

Output: An output entity is a sequence of values *produced* by evaluating a term, concatenating the sequences produced by the computations of its sub-terms.

Control: A control entity is a value that can optionally be *signalled* by a step. The corresponding step of an enclosing term may inspect the value, and signal the same value, signal a different value, or not signal.

The notation used for specifying auxiliary entities determines their classification. For instance, entities written before '⊢' are classified as contextual.

– *Can static semantics for funcons be defined in the same way as dynamic semantics?*

The modular structural operational semantics rules for funcon term evaluation are in the *small-step* style, where each rule has *at most one* transition premise. A static semantics for funcons would naturally use *big-step* rules, with a premise for each sub-term. It is currently unclear whether the same classification of entities can be used for static and dynamic semantics; the static semantics of abstractions generally requires making latent effects explicit, in contrast to dynamic semantics.

– *How have funcons been implemented?*

The initial implementation of funcons was in Prolog. Funcon definitions were translated to Prolog clauses defining transitions,[11] based on the original implementation of MSOS in Prolog.[12] Funcons have also been implemented in Maude.[13] The Prolog implementation of MSOS was subsequently enhanced to support the rewriting relation used in value-computation transition systems [7]. The Funcon Tools package [5] supports parsing funcon definitions and generating funcon interpreters in Haskell, as described in [4].

– *Could funcons be used for language specification in other frameworks?*

The K-framework [28] has a high degree of modularity. For an experiment with using the K-framework to define funcons, see [24]. The distinction between pre-evaluated and unevaluated arguments in funcon signatures is represented by strictness annotations in K. However, rules in K are unconditional, so funcons such as `scope` cannot be defined straightforwardly. The specification of the structure of states is monolithic, and may need updating when adding new funcons.

Redex [13] is a popular domain-specific metalanguage for operational semantics, embedded in the Racket programming language. It is based on reduction rules and evaluation contexts. The reduction rules are highly modular, and grammars for language constructs and evaluation contexts can be specified incrementally. However, evaluation context grammars associated with control operators appear to be inherently global. It should be possible to define a particular collection of funcons in Redex, but adding a new funcon could require updating the evaluation contexts for existing funcons.

7 Related Work

Many funcons are closely related to notation used in several previously developed language specification frameworks: denotational semantics, monads, abstract semantic algebras, and action semantics.

[11] https://pdmosses.github.io/prolog-msos-tool
[12] https://pdmosses.github.io/msos-in-prolog
[13] https://github.com/fcbr/mmt

Denotational Semantics. The funcons for flowing, binding, and storing are directly based on Christopher Strachey's original conceptual analysis of imperative programming languages. Strachey initiated the development of denotational semantics at the IFIP Working Conference on *Formal Language Description Languages* in 1964 [31]. At the time, he was working on the design and implementation of the high-level CPL programming language, and aiming to specify its semantics formally. In the paper, he focuses on representing imperative features of programming languages as pure mathematical functions, avoiding the introduction of abstract machines. For assignment commands, he distinguishes between L-values and R-values of expressions, with locations in stores σ being a special case of L-values. He defines the operation C to get the current content of a location, and U to update the content. For flow of control, he uses composition of functions from stores to stores, and the fixed-point operation Y. In his widely-circulated 1967 lecture notes [32], he also introduces environments that map names to values, and represents procedures as closures.

Strachey's original operation C on stores is renamed *Contents* in [29], and U is renamed *Assign*. Many subsequent denotational specifications define a large number of such auxiliary operations (e.g., [16] defines about 80). However, the definitions are ad hoc, and they are based on the domains defined for the specified language. Even the way lambda-expressions are written, and the notation used for modifying environments and stores, vary between denotational specifications.

The VDM metalanguage for denotational semantics, developed from 1974 [11], introduced fixed notation for operations expressing basic mathematical and computational concepts. The notation for data flow, control flow, storing, and exception handling looks rather like a programming language, but it is intepreted as pure mathematical functions (the interpretation depends on whether exceptions are used).

Monads. The types of the mathematical functions used in denotational semantics can be quite complicated. In 1989, Eugenio Moggi suggested that each feature should be seen as a monad, where the elements represent computations of values in arbitrary domains [15]; moreover, the required domains could be defined modularly, by applying a series of monad constructors. Monads have a binary operation for composing a computation with a function that takes its computed value,[14] corresponding to the funcon give(X, Y), and a unary operation for giving a value as the result of a computation (not needed with funcons). Each monad constructor adds further structure to the domain of computations, together with associated operations. For example, the monad constructor for stores in a domain S makes computations of values in T take an argument in S and return both a value in T and a store in S. The associated operations are *lookup*(l), to return the value at location l in the argument store and the unchanged store, and *update*(l, v), to return a null value and a store where the value at l is v. The funcons assigned(Var) and assign(Var, V) correspond to (a typed variant of) the operations defined by the store monad constructor. Other

[14] See [20] for discussion of earlier uses of similar operations in denotational semantics.

funcons closely correspond to the operations associated with monad constructors for a wide range of notions of computation. Monad constructors also need to lift definitions of operations to the resulting domains, which is non-trivial. The notation for monad constructors and operations varies (also between functional programming languages and proof assistants that support monads).

Abstract Semantic Algebras and Action Semantics. In a series of papers in the 1980s, the present author proposed various sets of combinators, together with algebraic laws that they were supposed to obey, giving so-called *abstract semantic algebras*. The elements of abstract semantic algebras were intended to have a clear operational interpretation; they were referred to as 'actions' from 1985.

The action notation used in the action semantics framework [17,18] was developed in collaboration with David Watt [25]. It was defined [17, App. C] using a novel (but non-modular) variant of structural operational semantics, and use of action semantics was supported by tools implemented in the ASF+SDF Meta-Environment [6,9].

Action notation involves actions, data, and yielders. The performance of an action represents information processing behaviour. Yielders used in actions may access, but not change, the current information. The evaluation of a yielder always results in a data entity. Many funcons correspond closely to the combinators of action notation. The crucial difference is that action notation could not be extended with new features, due to the non-modularity of its operational definition. The development of modular structural operational semantics [19] was directly motivated by the aim of making the definition of action notation extensible, and avoiding reduction of the many facets of action behaviour to pure functions in monads [34].

8 Conclusion

The PLANCOMPS project has defined the behaviour of a substantial collection of funcons, and illustrated translation of functional and imperative language constructs to funcons [8,26]. It has also developed their theoretical foundations [7]. Specifying languages by translation to funcons appears to be significantly less effort than with other frameworks. Funcon definitions and translations have been validated by testing, using generated interpreters; web pages and PDFs are generated from the same source files, with hyperlinks from names to definitions to support browsing and navigation.

Much remains to be done. Current and future work includes: completion and release of the initial collection of funcons and further tool support; demonstration of scaling up to translation of a major language such as C♯; improvement of the definitions of funcons for multithreading; defining the static semantics of funcons; defining funcons for expressing static semantics of language constructs; proving algebraic laws for funcons; and investigating whether funcons can be used also for specifying the semantics of declarative and domain-specific programming languages. PLANCOMPS welcomes new participants who would like to contribute to the development of funcons!

Acknowledgements. Helpful comments on a previous version were provided by Thomas van Binsbergen, Cliff Jones, Neil Sculthorpe, members of the PL Group at Delft, and the anonymous reviewers. Thanks to the track organisers Klaus Havelund and Bernhard Steffen for extra space for the appendices.

The initial development of funcons was supported by an EPSRC grant to Swansea University for the PLanCompS project (EP/I032495/1). The author is now an emeritus at Swansea, and a visitor at Delft University of Technology.

A Data

A.1 Datatypes

Primitive values. Conceptually, primitive values are atomic, and not formed from other values. For large (or infinite) types of primitive values, however, it is infeasible to declare a separate constant for each value. So in practice, funcons used to construct primitive values usually take other values as arguments.

- booleans are the values true, false; funcons corresponding to the usual Boolean operations are defined.
- integers is the built-in type of unbounded integers, with funcons for the usual mathematical operations. Funcons corresponding to associative binary operations are extended to arbitrary numbers of arguments. Subtypes include naturals and bounded(M, N); compositions with casts to such subtypes correspond to partial operations representing computer arithmetic.
- floats is the built-in type of IEEE floating point numbers, with funcons for the required operations.
- characters is the built-in type of all Unicode characters. Its subtypes include ascii-characters and iso-latin-1-characters. Its funcons incude the UTF-8, UTF-16, and UTF-32 encodings of characters as byte sequences.
- null-type has the single value null-value, alias null.

Composite values. Conceptually, composite values are constructed from finite sequences of argument values. The types of composite values include parametrised algebraic data types, with a generic representation. Various algebraic datatypes are defined, and new ones can be introduced. Composite values include also built-in parametrised types of sets, maps, multi-sets, and graphs.

Algebraic datatypes.

- datatype-values are generic representations for all algebraic datatype values.
- tuples$(T_1, \cdots, T_n)$ are grouped sequences of values of the specified types.
- lists(T) are grouped sequences of values of type T, with the usual operations; strings are lists of characters.
- vectors(T) are grouped sequences of values of type T, accessed by index.
- trees(T) are finite, with values of type T at nodes and leaves.
- references(T) are values that refer to values of type T.

- pointers(T) are references to values of type T or pointer-null.
- records(T) are unordered aggregate values, indexed by identifiers.
- variants(T) are pairs of identifiers and values of type T.
- classes are collections of features, allowing multiple superclasses, used to classify objects.
- objects are classified collections of features.
- bit-vectors(N) has instantiations for bits and bytes.

Built-in datatypes.

- sets(GT) are finite sets of ground values of type GT.
- maps($GT, T^?$) are finite maps from type GT to type $T^?$.
- multisets(GT) are finite multisets of ground values of type GT.
- directed-graphs(GT) have values of type GT as vertices.

See [26] for funcons that operate on the above types of values.

A.2 Abstractions

Generic Abstractions. These non-ground values are used for constructing thunks, functions, and patterns. An abstraction body of computation type $T \Rightarrow T'$ may refer to a given value of type T, and compute values of type T'.

- abstractions(CT) are procedural abstractions of computation type CT.
- abstraction(X) constructs an abstraction with dynamic bindings.
- closure(X) computes an abstraction with static bindings.
- enact(A) evaluates the body of the abstraction A.

Thunks. The abstraction body of a thunk does not reference a given value.

- thunks(T) are constructed from abstractions with bodies of type $( ) \Rightarrow T'$.
- thunk(A) constructs a thunk from the abstraction A.
- force(V) enacts the abstraction of the thunk V.

Functions. The abstraction body of a function may reference a given value.

- functions(T, T') are constructed from abstractions with bodies of type $T \Rightarrow T'$.
- function(abstraction(X)) constructs a function with dynamic bindings.
- function(closure(X)) computes a function with static bindings.
- apply(F, V) gives the value V to the body of the abstraction of function F.
- supply(F, V) determines the argument value of a function application, but returns a thunk that defers evaluating the body of the function.
- compose(F_2, F_1) returns the function that first applies F_1 then F_2.
- curry(F) takes a function F that takes a pair of arguments, and returns the corresponding 'curried' function.
- uncurry(F) takes a curried function F and returns a function that takes a pair of arguments.
- partial-apply(F, V) takes a function F that takes a pair of arguments, and determines the first argument, returning a function of the second argument.

Patterns. The abstractions of patterns match a given value.

- patterns are constructed from abstractions with bodies of computation type values ⇒ environments.
- pattern(A) constructs patterns from abstractions A.
- match(X, pattern(A)) enacts the abstraction A, giving it the value of X.

B Flow of Control

- left-to-right($\cdots$) evaluates its arguments sequentially, and concatenates the computed value sequences. Composing it with a funcon having pre-computed arguments prevents interleaving; e.g., integer-add(left-to-right(X, Y)) always executes X before Y.
  right-to-left($\cdots$) is analogous.
  interleave($\cdots$) evaluates its arguments in any order, possibly with interleaving, and concatenates the computed value sequences.
- sequential($X, \cdots$) executes the command X, then any remaining arguments, evaluating to the same value(s) as the last argument.
- effect($\cdots$) interleaves the evaluations of its arguments, discarding their computed values, and gives null-value.
- choice($Y, \cdots$) selects one of its arguments, then evaluates it.
- if-true-else(B, X, Y) evaluates B to a Boolean value, then evaluates either X or Y (which have to compute values of the same type).
- while-true(B, X) evaluates B to a Boolean value, then either executes X (which has to correspond to a command) and iterates, or terminates.

C Flow of Data

- given evaluates to the current value of the auxiliary entity given-value.
- give(X, Y) evaluates X. It then executes Y with the value of X as the value of the auxiliary entity given-value.
- left-to-right-map(F, V^*) evaluates F for each value in the sequence V^* in the same order, computing the sequence of resulting values.
  interleave-map(F, V^*) allows interleaving of the evaluations.
- left-to-right-repeat(F, M, N) evaluates F for each integer from M up to N sequentially, computing the sequence of resulting values.
  interleave-repeat(F, M, N) allows interleaving of the evaluations.
- left-to-right-filter(P, V^*) evaluates P for each value in V^*, computing the sequence of argument values for which the value of P is true.
  interleave-filter(P, V^*) allows interleaving of the evaluations.
- fold-left(F, A, V^*) reduces a sequence V^* to a single value by folding it from the left, using A as the initial accumulator value.
  fold-right(F, A, V^*) is analogous.

For any list L, the funcon term list-elements(L) evaluates to the sequence V^* of elements in L, and list(V^*) reconstructs L. Composition with these funcons allows the above funcons on sequences to be used with lists; similarly for vectors, sets, multisets, and the datatype of maps.

D Name Binding

- bind-value(I, X) computes the singleton environment mapping I to the value computed by X.
- unbind(I) computes an environment that hides the binding of I.
- bound-value(I) computes the value to which I is currently bound (possibly recursively, via a link), if any, and otherwise fails.
- scope(D, X) first evaluates D to compute an environment ρ. It then extends the auxiliary environment entity with ρ for the execution of X.
- closed(X) prevents references to non-local bindings while evaluating X.
- accumulate(D_1, D_2) first evaluates D_1, to compute an environment ρ_1. It then extends the auxiliary environment entity by ρ_1 for the evaluation of D_2, to compute an environment ρ_2. The result is ρ_1 extended by ρ_2.
- collateral$(D_1, \cdots)$ evaluates its arguments to compute environments. It returns their union as result, failing if their domains are not pairwise disjoint.
- bind-recursively(I, X) makes bind-value(I, X) recursive. It first computes a singleton environment ρ mapping I to a fresh link L. It then extends the auxiliary environment entity by ρ for the execution of X, to compute a value V. Finally, it sets L to refer to V, and gives ρ as the computed result.
- recursive(SI, D) makes D recursive on the identifiers in the set SI. It first computes an environment ρ mapping all I in SI to fresh links. It then extends the auxiliary environment entity by ρ for the execution of D, to compute an environment ρ'. Finally, it sets the link for each I to refer to the value of I in ρ', and gives ρ' as the computed result.

E Imperative Variables

- variables is the type of all simple variables.
- allocate-variable(T) constructs a simple variable for storing values of type T in a location not in the current store.
- recycle-variables$(Var, \cdots)$ removes locations allocated to variables from the current store.
- initialise-variable(Var, V) assigns V as the initial value of Var.
- allocate-initialised-variable(T, V) is a composition of allocate-variable(T) and initialise-variable$(_, V)$.
- assign(Var, V) stores V at the location of Var when the type contains V.
- assigned(Var) gives the value last assigned to Var.
- current-value(V) gives the same result as assigned(V) when V is a simple variable, otherwise V.
- un-assign(Var) makes Var uninitialised.
- structural-assign(V_1, V_2) assigns to all the simple variables in V_1 the corresponding values in V_2, provided that the structure and all non-variable values in V_1 match the structure and corresponding values of V_2.
- structural-assigned(V) computes V with all simple variables replaced by their assigned values. When V is a simple variable or a value with no component variables, structural-assigned(V) gives the same result as current-value(V).

F Abrupt Termination

- abrupt(V) terminates abruptly for reason V.
- handle-abrupt(X, Y) first executes X. If X terminates normally, Y is ignored. If X terminates abruptly for any reason, Y is executed, with the reason as the given value.
- finally(X, Y) first executes X. On normal or abrupt termination of X, it executes Y. If Y terminates normally, its computed value is ignored, and the funcon terminates in the same way as X; otherwise it terminates in the same way as Y.
- fail abruptly terminates for reason failed.
- else($X_1, X_2, \cdots$) executes the arguments in turn until either some X_i does *not* fail, or all arguments X_i have been executed. The last argument executed determines the result.
  else-choice($X_1, X_2, \cdots$) is similar, but executes the arguments sequentially in any order.
- check-true(X) terminates normally if the value computed by X is true, and fails if it is false.
- checked(X) fails when X computes the empty sequence of values ( ), representing that a value has not been computed. It otherwise computes the same as X.
- throw(V) abruptly terminates for reason thrown(V).
  handle-thrown(X, Y) handles abrupt termination of X for reason thrown(V) with Y.
  handle-recursively(X, Y) is similar to handle-thrown(X, Y), except that another copy of the handler attempts to handle any values thrown by Y.
- return(V) abruptly terminates for reason returned(V).
  handle-return(X) evaluates X. If X either terminates abruptly for reason returned(V), or terminates normally with value V, it terminates normally giving V.
- break abruptly terminates for reason broken.
  handle-break(X) terminates normally when X terminates abruptly for reason broken.
- continue abruptly terminates for reason continued.
  handle-continue(X) terminates normally when X terminates abruptly for reason continued.

Further funcons are provided for expressing delimited continuations [26,30].

G Communication

- read inputs a single non-null value from the standard-in entity, and gives it as the result.
- print(V^*) outputs the sequence of values V^* to the standard-out entity.

References

1. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java, 2nd edition. Cambridge Univ. Press (2002)
2. Batory, D.S., Höfner, P., Kim, J.: Feature interactions, products, and composition. In: Denney, E., Schultz, U.P. (eds.) GPCE 2011. pp. 13–22. ACM (2011). https://doi.org/10.1145/2047862.2047867
3. van Binsbergen, L.T.: Funcons for HGMP: the fundamental constructs of homogeneous generative meta-programming (short paper). In: Wyk, E.V., Rompf, T. (eds.) GPCE 2018. pp. 168–174. ACM (2018). https://doi.org/10.1145/3278122.3278132
4. van Binsbergen, L.T., Mosses, P.D., Sculthorpe, N.: Executable component-based semantics. J. Log. Algebr. Meth. Program. **103**, 184–212 (2019). https://doi.org/10.1016/j.jlamp.2018.12.004
5. van Binsbergen, L.T., Sculthorpe, N.: Funcons-tools: A modular interpreter for executing funcons, https://hackage.haskell.org/package/funcons-tools, software package, accessed 2021-08-08
6. van den Brand, M., Iversen, J., Mosses, P.D.: An action environment. Sci. Comput. Program. **61**(3), 245–264 (2006). https://doi.org/10.1016/j.scico.2006.04.005
7. Churchill, M., Mosses, P.D.: Modular bisimulation theory for computations and values. In: Pfenning, F. (ed.) FOSSACS 2013. LNCS, vol. 7794, pp. 97–112. Springer (2013). https://doi.org/10.1007/978-3-642-37075-5
8. Churchill, M., Mosses, P.D., Sculthorpe, N., Torrini, P.: Reusable components of semantic specifications. LNCS Trans. Aspect Oriented Softw. Dev. **12**, 132–179 (2015). https://doi.org/10.1007/978-3-662-46734-3_4
9. van Deursen, A., Mosses, P.D.: ASD: the action semantic description tools. In: Wirsing, M., Nivat, M. (eds.) AMAST '96. LNCS, vol. 1101, pp. 579–582. Springer (1996). https://doi.org/10.1007/BFb0014346
10. GraalVM: Introduction to SimpleLanguage, https://www.graalvm.org/graalvm-as-a-platform/implement-language, accessed 2021-08-08
11. Jones, C.B.: The transition from VDL to VDM. J. Univers. Comput. Sci. **7**(8), 631–640 (2001). https://doi.org/10.3217/jucs-007-08-0631
12. Kats, L.C.L., Visser, E.: The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) OOPSLA 2010. p. 444–463. ACM (2010). https://doi.org/10.1145/1869459.1869497
13. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run your research: On the effectiveness of lightweight mechanization. In: Field, J., Hicks, M. (eds.) POPL 2012. pp. 285–296. ACM (2012). https://doi.org/10.1145/2103656.2103691
14. Madlener, K., Smetsers, S., van Eekelen, M.C.J.D.: Formal component-based semantics. In: Reniers, M.A., Sobocinski, P. (eds.) SOS 2011. EPTCS, vol. 62, pp. 17–29 (2011). https://doi.org/10.4204/EPTCS.62.2
15. Moggi, E.: An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Edinburgh Univ. (1989)
16. Mosses, P.D.: The mathematical semantics of Algol 60. Tech. Mono. PRG-12, Oxford Univ. Comp. Lab. (1974)
17. Mosses, P.D.: Action Semantics, Cambridge Tracts in TCS, vol. 26. Cambridge Univ. Press (1992). https://doi.org/10.1017/CBO9780511569869
18. Mosses, P.D.: Theory and practice of action semantics. In: Penczek, W., Szalas, A. (eds.) MFCS'96. LNCS, vol. 1113, pp. 37–61. Springer (1996). https://doi.org/10.1007/3-540-61550-4_139

19. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. **60-61**, 195–228 (2004). https://doi.org/10.1016/j.jlap.2004.03.008
20. Mosses, P.D.: VDM semantics of programming languages: Combinators and monads. Formal Aspects Comput. **23**(2), 221–238 (2011). https://doi.org/10.1007/s00165-009-0145-4
21. Mosses, P.D.: A component-based formal language workbench. In: Monahan, R., Prevosto, V., Proença, J. (eds.) F-IDE@FM. EPTCS, vol. 310, pp. 29–34 (2019). https://doi.org/10.4204/EPTCS.310.4
22. Mosses, P.D.: Software meta-language engineering and CBS. J. Comput. Lang. **50**, 39–48 (2019). https://doi.org/10.1016/j.jvlc.2018.11.003
23. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. ENTCS **229**(4), 49–66 (2009). https://doi.org/10.1016/j.entcs.2009.07.073
24. Mosses, P.D., Vesely, F.: FunKons: Component-based semantics in K. In: Escobar, S. (ed.) WRLA 2014. LNCS, vol. 8663, pp. 213–229. Springer (2014). https://doi.org/10.1007/978-3-319-12904-4_12
25. Mosses, P.D., Watt, D.A.: The use of action semantics. In: Wirsing, M. (ed.) Formal Description of Programming Concepts III. pp. 135–166. North-Holland (1987)
26. PLanCompS Project: CBS: A framework for component-based specification of programming languages, https://plancomps.github.io/CBS-beta, accessed 2021-08-08
27. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60-61**, 17–139 (2004). https://doi.org/10.1016/j.jlap.2004.05.001
28. Rosu, G.: K: A semantic framework for programming languages and formal analysis tools. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, pp. 186–206. IOS Press (2017). https://doi.org/10.3233/978-1-61499-810-5-186
29. Scott, D.S., Strachey, C.: Toward a mathematical semantics for computer languages. In: Fox, J. (ed.) Proc. Symp. on Computers and Automata. Microwave Research Inst. Symposia, vol. 21, pp. 19–46. Polytechnic Inst. of Brooklyn (1971), also Tech. Mono. PRG-6, Oxford Univ. Comput. Lab
30. Sculthorpe, N., Torrini, P., Mosses, P.D.: A modular structural operational semantics for delimited continuations. In: Danvy, O., de'Liguoro, U. (eds.) WoC 2016. EPTCS, vol. 212, pp. 63–80 (2015). https://doi.org/10.4204/EPTCS.212.5
31. Strachey, C.: Towards a formal semantics. In: Formal Language Description Languages for Computer Programming. p. 198–216. North-Holland (1966)
32. Strachey, C.S.: Fundamental concepts in programming languages. High. Order Symb. Comput. **13**(1/2), 11–49 (2000). https://doi.org/10.1023/A:1010000313106, lecture notes, Int. Summer School in Comput. Prog., Copenhagen, 1967
33. Torrini, P., Schrijvers, T.: Reasoning about modular datatypes with Mendler induction. In: Matthes, R., Mio, M. (eds.) FICS 2015. EPTCS, vol. 191, pp. 143–157 (2015). https://doi.org/10.4204/EPTCS.191.13
34. Wansbrough, K., Hamer, J.: A modular monadic action semantics. In: Ramming, C. (ed.) DSL'97. USENIX (1997), http://www.usenix.org/publications/library/proceedings/dsl97/wansbrough.html