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.

1 Introduction

Even very different programming languages often share similar constructs. Consider OCaml's conditional '**if** E_1 **then** E_2 **else** E_3' and the conditional operator 'E_1 ? E_2 : E_3' in C. These constructs have different concrete syntax but similar semantics, with some variation in details. We would like to exploit this similarity when defining formal semantics for both languages by reusing commonalities between the OCaml and C specifications. With traditional approaches to semantics, reuse through 'copy-paste-and-edit' is usually the only option that is available to us. By default, this is also the case with the K Framework [9,13]. This style of specification reuse is not systematic, and prone to error.

The semantic framework currently being developed by the PLanCompS project[1] provides *fundamental constructs* (funcons) that address the issues of reusability in a systematic manner. Funcons are small semantic entities which express essential concepts of programming languages. These formally specified components can be composed to capture the semantics of concrete programming language constructs. A specification of Caml Light has been developed as an initial case study [3] and a case study on C# is in progress.

[1] http://www.plancomps.org/

S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 213–229, 2014.
DOI: 10.1007/978-3-319-12904-4_12

For example, the funcon `if-true` can be used to specify OCaml's conditional expression. Semantics is given by defining a translation from the concrete construct to the corresponding funcon term:

$$[\![\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3]\!] = \texttt{if-true}([\![E_1]\!], [\![E_2]\!], [\![E_3]\!])$$

Since the conditional operator in C uses integer valued expressions as the condition, its translation will reflect this:

$$[\![E_1\ \texttt{?}\ E_2\ \texttt{:}\ E_3]\!] = \texttt{if-true}(\texttt{not}(\texttt{equal}([\![E_1]\!], 0)), [\![E_2]\!], [\![E_3]\!])$$

We could also define an `if-non-zero` funcon that would match the C-conditional semantics exactly. However, the translation using `if-true` is so simple that there wouldn't be much advantage in doing so. We can reuse the `if-true` funcon, and with it, its semantic definition. This way, we also make the difference between the OCaml and C conditional construct explicit. Section 2 provides more information on funcons.

PLanCompS uses MSOS [10], a modular variant of structural operational semantics [11], to formally define individual funcons. However, the funcon approach can be seamlessly integrated with other sufficiently modular specification frameworks. We have tested the use of funcons with the K Framework by giving a specification of CinK [8,9], a pedagogical subset of C++. We have defined both the translation of CinK to funcons and the semantics of the funcons using K's rewrite rules. The complete prototyped specification is available online, together with the CinK test programs which we have used to test our specification.[2] Interested readers may run these programs themselves using the K tool.

In this paper, we present our specification of the CinK translation (Sect. 3) and illustrate the definition of the semantics of funcons involved in it (Sect. 4). Section 5 offers an overview of related work and alternative approaches. We conclude and suggest directions of future work in Sect. 6.

2 Fundamental Constructs

As mentioned in the Introduction, the PLanCompS project is developing an open-ended collection of fundamental programming constructs, or 'funcons'. Many funcons correspond closely to simplified programming language constructs. However, each funcon has *fixed* syntax and semantics. For example, the funcon written `assign(`*`E1`*`,`*`E2`*`)` has the effect of evaluating *`E1`* to a variable, *`E2`* to a value (in any order), then assigning the value to the variable; it is well-typed only if *`E1`* is of type `variables(`*`T`*`)` and *`E2`* is of type *`T`*. In contrast, the language construct written '*`E1`* `=` *`E2`*' may be interpreted as an assignment or as an equality test (and its well-typedness changes accordingly) depending on the language.

The syntax or *signature* of a funcon determines its name, how many arguments it takes (if any), the sort of each argument, and the sort of the result. The following *computation sorts* reflect fundamental conceptual and semantic distinctions in programming languages.

[2] http://cs.swan.ac.uk/~csfvesely/wrla2014/

- The sort `Comm` (commands) is for funcons (such as `assign(E1,E2)`) that are executed only for their effects; on normal termination, a command computes the fixed value `skip`.
- The sort `Expr` (expressions) is for funcons (such as `stored-value(E)` and `bound-value(I)`) that compute values of sort `Values`.
- The sort `Decl` (declarations) is for funcons (such as `bind-value(I,E)`) that compute values of sort `Environments`, which represent sets of bindings between identifiers and values.

All computation sorts include their sorts of computed values as subsorts: a value takes no steps at all to compute itself.

One of the aims of the PLanCompS project is to establish an online repository of funcons (and data types) for anybody to use 'off-the-shelf' as components of language specifications. The project is currently testing the reusability of existing funcons and developing new ones in connection with some major case studies (including Caml Light, C#, and Java). Because individual funcons are meant to represent *fundamental* concepts in programming languages, many funcons (expressing, e.g., sequencing, conditionals, variable lookup and dereferencing) have a high potential for reuse. In fact, many funcons used in the Caml Light case study appear in the semantics of CinK presented in the following section.

The nomenclature and notation for the existing funcons are still evolving, and they will be finalised only when the case studies have been completed, in connection with the publication of the repository. Observant readers are likely to notice some (minor) differences between the funcon names used in this paper and in previous papers (e.g. [3]).

Regardless of the details of funcon notation, funcons can be algebraically composed to form funcon terms, according to their argument and result sorts (strictly lifted to corresponding computation sorts). Well-*formedness* of funcon terms is context-free: `assign(E1,E2)` is a well-formed funcon term whenever `E1` and `E2` are well-formed funcon terms of sort `Expr`. In contrast, well-*typedness* of funcon terms is generally context-sensitive. For example, the funcon term `assign(bound-value(I),42)` is well-typed only in the scope of a declaration that binds `I` to an integer variable. Dynamic semantics is defined for all well-formed terms; execution of ill-typed terms may fail.

The composability of funcons does *not* depend on features such as whether they might have side effects, terminate abruptly, diverge, spawn processes, interact, etc. This is crucial for the reusability of the funcons. The semantics of each funcon has to be specified without regard to the context in which it might be used, which requires a highly modular specification framework. Funcon specifications have previously been given in MSOS, Rewriting Logic, ASF + SDF, and action notation. Here, we explore specifying funcons in K, following Roșu.[3]

A component-based semantics of a programming language is specified by a context-free grammar for an abstract syntax for the language, together with a family of inductively specified functions translating abstract syntax trees to

[3] `k/examples/funcons` in the stable K distribution at http://www.kframework.org.

funcon terms. The static and dynamic semantics of a program is given by that of the resulting funcon term. As mentioned above, funcons have fixed syntax and semantics. Thus, evolution of a language is expressed as changes to translation functions. If the syntax or semantics of the programming language changes, the definition of the translation function has to be updated to reflect this.

Tool support for translating programs to funcon terms, and for executing the static and dynamic semantics of such terms, has previously been developed in Prolog [2], Maude [1] and ASF + SDF. We now present our experiment with K, focusing on dynamic semantics.

3 A Funcon Specification of CinK

This section presents an overview of our CinK specification using funcons. We include examples from the K sources of the specification. A selection of definitions of funcons involved in the specification can be found in Sect. 4.

CinK is a pedagogical subset of C++ [8,9] used for experimentation with the K Framework. The original report [8] presents the language in seven iterations. The first specifies a basic imperative language; subsequent iterations extend it with threads, model-checking, references, pointers, and uni-dimensional and multi-dimensional arrays. Our specification starts with only an expression language which we extend with declarations, statements, functions, threads, references, pointers, and arrays. The extensions follow the order of the CinK iterations; however, we omit support for model-checking.

The grammar which we have used for our specification is a simplified grammar matching CinK derived from the C++ grammar found in the standard [7, Appendix A].

We invite the reader to compare our specification by translation to funcons with the original K specification of CinK in [8]. Our hope is that our translation functions, together with the suggestive naming of funcons, give a rough understanding of the semantics of language constructs, even before looking at the semantics of funcons themselves.

3.1 Simple Expressions

To give semantics for expressions we use the translation function `evaluate⟦_⟧ : Expression → Expr`. It produces a funcon term (of sort `Expr`) which, when executed, evaluates the argument expression.

Definitions for arithmetic expressions in CinK can be given very straightforwardly using data operations, which all extend to strict funcons on `Expr`. For example, semantics of the multiplication operator is expressed as the application of the operation `int-times` to translations of operand expressions (numeric types in CinK are limited to integers with some common operations):

```
rule evaluate⟦ E1:Expression * E2:Expression ⟧ ⇒
    int-times(evaluate⟦ E1 ⟧, evaluate⟦ E2 ⟧)
```

The 'short-circuit and' operator can be readily expressed using a conditional funcon, which is strict only in its first argument. The (obvious) K definition for `if-true` can be found in Sect. 4.

```
rule evaluate⟦ E1:Expression && E2:Expression ⟧ ⇒
    if-true(evaluate⟦ E1 ⟧, evaluate⟦ E2 ⟧, false)
```

We will use the generic `if-true` funcon later in this section to define the conditional statement.

3.2 Variables, Blocks and Scope

Bindings and Variables. Semantics of declarations are given using the translation function `elaborate⟦_⟧ : DeclarationSeq → Decl`. The `bind-value(I,V)` funcon binds the identifier `I` to the value `V`, producing a 'small' environment containing only the newly created binding. To allocate a new variable of a specified type we use `allocate`. In Caml Light, `bind-value` was used for individual name-value bindings in let-expressions, and `allocate` for reference data types (e.g. '`ref int`').

```
rule elaborate⟦ T:TypeSpecifier I:Id ; ⟧ ⇒
    bind-value(I, allocate(variables(type⟦ T ⟧)))
```

In relation to variables, CinK (following C++) distinguishes between two general categories of expressions: *lvalue*- and *rvalue*-expressions. We express this distinction by having different translation functions for expressions in lvalue and rvalue contexts: in addition to `evaluate⟦_⟧`, we define `evaluate-lval⟦_⟧` and `evaluate-rval⟦_⟧`. The default function `evaluate⟦_⟧` produces terms evaluating lvalue and rvalue expressions according to their category. When an expression is expected to evaluate to an *lvalue*, we use `evaluate-lval⟦_⟧`. When an *rvalue* is expected, we use `evaluate-rval⟦_⟧` which produces terms evaluating all expressions into rvalues. For lvalue expressions it returns the corresponding stored value, i.e., it serves as an lvalue-to-rvalue conversion.

The addition of variables also affects our translations of simple expressions and we need to update them. For example, numeric operations expect an rvalue and thus the operands are now translated using `evaluate-rval⟦_⟧`.

To obtain the variable bound to an identifier in the current environment we use `bound-value`. A variable is dereferenced using `stored-value`. The semantics for an identifier appearing in an lvalue or rvalue context is thus:

```
rule evaluate-lval⟦ I:Id ⟧ ⇒ bound-value(I)
rule evaluate-rval⟦ I:Id ⟧ ⇒ stored-value(evaluate-lval⟦ I ⟧)
```

Blocks and Controlling Scope. We distinguish between declaration statements and other statements within a block using funcons `scope` and `seq`. The funcon `scope(D,X)` evaluates `X` in the current environment overridden with the environment computed by `D`. A declaration statement within a block produces a new environment that is valid until the end of the block:

```
rule execute[[ BD:BlockDeclaration SS:StatementSeq ]] ⇒
    scope(elaborate[[ BD ]], execute[[ SS ]])
```

The function `execute[[_]] : StatementSeq → Comm` translates statements to funcon commands.

For all other kinds of statements in a block we use the simple sequencing funcon `seq(C,X)` which executes the command `C` for side effects, then executes `X`.

```
rule execute[[ BS:BlockStatement SS:StatementSeq ]] ⇒
    seq(execute[[ BS ]], execute[[ SS ]])
```

To accumulate multiple declarations into one environment we use the `accum` funcon. The funcon `accum(D1,D2)` is similar to `scope`, except its result is the environment produced by elaborating declaration `D2` and overriding the environment computed by `D1` with it. This matches the semantics of a multi-variable declaration:

```
rule elaborate[[ T:TypeSpecifier ID:InitDeclarator ,
                                 IDL:InitDeclaratorList ; ]] ⇒
    accum(elaborate[[ T ID ; ]], elaborate[[ T IDL ; ]])
```

Note that `accum` is strict only in its first argument, so the correct order of evaluation is enforced.

Although Caml Light and CinK are quite different languages, all the funcons we needed here so far for CinK are reused from [3].

3.3 Assignment and Control Statements

The basic construct for updating variables in CinK/C++ is the assignment expression '`E1 = E2`', where the expression `E1` is expected to evaluate to an lvalue, to which the rvalue of `E2` will be assigned. The value of the whole expression is the lvalue of `E1`. Semantics of assignment is a rather simple translation using the `assign-giving-variable` funcon (defined in Sect. 4.4):

```
rule evaluate-lval[[ E1:Expression = E2:Expression ]] ⇒
    assign-giving-variable(evaluate-lval[[ E1 ]], evaluate-rval[[ E2 ]])
```

The funcon `assign-giving-variable` is strict in both arguments but not sequentially, so the arguments are evaluated in an unspecified order. The funcon assigns the value given as its second argument to the variable given as its first argument and returns this variable as result.

CinK has boolean-valued conditions and the translations of while- and if-statements are trivial:

```
rule execute[[ while ( E:Expression ) S:Statement ]] ⇒
    while-true(evaluate-rval[[ E ]], execute[[ S ]])
rule execute[[ if ( E:Expression ) S:Statement ]] ⇒
    execute[[ if ( E ) S else { } ]]
rule execute[[ if ( E:Expression ) S1:Statement else S2:Statement ]] ⇒
    if-true(evaluate-rval[[ E ]], execute[[ S1 ]], execute[[ S2 ]])
```

3.4 Function Definition and Calling

We represent functions as *abstraction values* which wrap any computation as a value. An abstraction can be passed as a parameter, bound to an identifier, or stored like any other value. To turn a funcon term into an abstraction, we use the `abstraction` value constructor. The funcon `apply` applies an abstraction to a value and the abstraction may refer to the passed value using `given`. Multiple parameters can be passed as a tuple constructed using tuple value constructors.

A function call expression simply applies the abstraction to translated arguments:

```
rule evaluate-rval⟦ E1:Expression ( E2:Expression ) ⟧ ⇒
    apply(evaluate-rval⟦ E1 ⟧, evaluate-params⟦ tuple(E2) ⟧)
```

At this stage the language only supports call-by-value semantics and so each parameter is evaluated to an *rvalue* before being passed to a function. The translation function `evaluate-params⟦_⟧` (defined in terms of `evaluate-rval⟦_⟧`) recurses through the parameter expressions and constructs a tuple.

```
rule evaluate-params⟦ tuple(E1:Expression , E2:Expression) ⟧ ⇒
    tuple-prefix(evaluate-rval⟦ E1 ⟧, evaluate-params⟦ tuple(E2) ⟧)
rule evaluate-params⟦ tuple(E:Expression) ⟧ ⇒
    tuple-prefix(evaluate-rval⟦ E ⟧, tuple(.))
```

We have introduced the auxiliary abstract syntax `tuple(E)` to ensure that parameters separated by commas are not interpreted as a comma-operator expression.

We use *patterns* as translations of function parameters. Patterns themselves are abstractions which compute an environment when applied to a matching value. The pattern for passing a single parameter by value allocates a variable of the corresponding type and binds it to an identifier; then it assigns the parameter value to the variable and returns the resulting environment.

```
rule pattern⟦ T:TypeSpecifier I:Id ⟧ ⇒
    abstraction(
      accum(bind-value(I, allocate(variables(type⟦ T ⟧))),
        decl-effect(assign(bound-value(I), given))))
```

Here we use the funcon `decl-effect(C)`, which allows using a command `C` as a declaration. It is an abbreviation for `seq(C,bindings(.))`.

Roughly, the semantics of a function definition is to allocate storage for an abstraction of the corresponding type, bind it to the function name, and use it to store an abstraction of the function body. Looking closer, the definition has to deal with some more details:

```
rule elaborate⟦ T:TypeSpecifier I:Id ( PDL:ParameterDeclarationList )
                CS:CompoundStatement ⟧ ⇒
    decl-effect(assign(bound-value(I),
      close(abstraction(
        scope(match-compound(pattern-tuple⟦ PDL ⟧, given),
          catch(seq(execute⟦ CS ⟧, throw(variant(returned, null))),
            abstraction(original(returned, given))))))))
```

Within the abstraction we use `match-compound` to match the passed value against the pattern tuple constructed from individual parameter patterns. The translation of the function body is evaluated in the environment produced by this matching (`scope`). Since a return statement abruptly terminates a function returning a value, we represent return statements as exceptions containing a value tagged with the atom '`returned`' and wrap the function body in a handler. The `catch` funcon catches the exception and the handling abstraction retrieves the value tagged with '`returned`', making it the return value of the whole function. In case there was no return statement in the body of the function, we throw a '`returned`' with `null`. Using `close` we form a closure of the abstraction with respect to the definition-time environment, to ensure static scopes for bindings.

As mentioned above, an explicit return statement translates to throwing a value tagged with '`returned`'. A parameterless return throws a `null`.

```
rule execute⟦ return E:Expression; ⟧ ⇒
    throw(variant(returned, evaluate-rval⟦ E ⟧))
rule execute⟦ return ; ⟧ ⇒ throw(variant(returned, null))
```

As a simple way of allowing self- and mutually recursive function definitions, we pre-allocate function variables and bind all function names declared at the top-level in a global environment using `evaluate-forwards⟦_⟧`. Then we combine this environment with the elaboration of full function definitions and other declarations. The `main` function is called in the scope of the global environment.

```
rule translate⟦ DS:DeclarationSeq ⟧ ⇒
    scope(accum(elaborate-forwards⟦ DS ⟧, elaborate⟦ DS ⟧),
      effect(apply(evaluate-rval⟦ main ⟧, tuple(.))))
```

Because function identifiers are already bound when the full function definition is elaborated, the full definition only assigns the abstraction to the pre-allocated variable.

3.5 Threads

The second iteration in the original CinK report adds very basic thread support to the language. Spawning a thread in CinK mimics the syntax of using the `std::thread` class from the C++ standard library. However, instead of referring to the standard library, semantics is given to the construct directly.

```
rule elaborate⟦ std::thread I1:Id ( I2:Id , E:Expression ) ; ⟧ ⇒
    decl-effect(effect(spawn(close(abstraction(evaluate⟦ I2 (E) ⟧)))))
```

The funcon `spawn(A)` creates a new thread in which the abstraction `A` will be applied. In our case the abstraction contains a function call corresponding to the parameters given to the thread constructor.

3.6 References

A reference in C++ is an alias for a variable, i.e., it introduces a new name for an already existing variable.

```
rule elaborate[[ T:TypeSpecifier & I:Id = E:Expression ]] ⇒
    bind-value(I, evaluate-lval[[ E ]])
```

The expression `E` is expected to compute an lvalue and we bind the resulting variable to identifier `I`. We are assuming that the input program is statically correct and thus the variable will have the right type.

A reference parameter pattern simply binds `I` to the given variable.

```
rule pattern[[ T:TypeSpecifier & I:Id ]] ⇒
    abstraction(bind-value(I, given))
```

Before introducing references, we evaluated function parameters to an rvalue. Now the function `evaluate-param[[_]]` has to be *redefined* in terms of `evaluate[[_]]` instead of `evaluate-rval[[_]]`. Dereferencing is handled conditionally inside the parameter pattern.

```
rule pattern[[ T:TypeSpecifier I:Id ]] ⇒
    abstraction(
      accum(bind-value(I, allocate(variables(type[[ T ]]))),
        decl-effect(assign(bound-value(I), current-value(given)))))
```

The funcon `current-value` dereferences its argument if it is a variable (lvalue), otherwise it returns the parameter itself.

3.7 Pointers

Pointer variables either hold a reference to another variable or are null otherwise. In this iteration we introduce auxiliary syntax for types, which we use to extract type information from declarations. Our type syntax is not part of the original language. It mostly resembles the original C++ syntax, except for function types which are expressed using a functional (arrow) notation. Here we extract types from a pointer declaration and a function declaration:

```
rule type[[ FT:FunType (* D:Declarator) ]] ⇒ type[[ (FT *) D ]]
rule type[[ FT:FunType ( D:Declarator
          ( PDL:ParameterDeclarationList ) ) ]] ⇒
    type[[ (( PDL ) --> FT) D ]]
```

We translate these intermediate types into funcon types (just as we do with simple types). The funcon type `pointers(T)` is the type of pointers to variables of type `T`:

```
rule type[[ (FT:FunType *):FunType ]] ⇒ pointers(type[[ FT ]])
```

To illustrate, consider the pointer declaration `int **ppi;` which declares `ppi` to be a pointer to a pointer to an integer variable. The type of this variable in our auxiliary syntax is `((int *) *)` and the analysed type is `pointers(pointers(variables(integers)))`.

Pointer variables are allocated in the same manner as other variables: we simply pass the type of the pointer variable as the argument to the `allocate` funcon.

Explicit dereferencing of a pointer variable in an expression amounts to retrieving the value stored in the pointer. This value is the location to which the pointer is pointing. This is expressed in our translation:

```
rule evaluate-lval⟦ * E:Expression ⟧ ⇒ evaluate-rval⟦ E ⟧
```

If the pointer is `null`, dereferencing it or assigning to it will result in a stuck computation.

3.8 Arrays

This extension adds uni-dimensional and multi-dimensional array declarations and expressions to the specification. We analyse CinK arrays, which are indexed from zero, in terms of vectors. Similarly to pointers, we use auxiliary syntax for array types.

```
rule type⟦ FT:FunType (D:Declarator [ E:Expression ]) ⟧ ⇒
    type⟦ ([E] FT) D ⟧
rule type⟦ ([E:Expression] FT:FunType):FunType ⟧ ⇒
    vectors(evaluate⟦ E ⟧, type⟦ FT ⟧)
```

The arguments of the type constructor `vectors` are the length of the vector and the type of its elements. To allocate an array of a given type, we use the `allocate-vector` funcon:

```
rule elaborate⟦ ([ E:Expression ] FT:FunType) I:Id ; ⟧ ⇒
    bind-value(I, allocate-vector(type⟦ ([E] FT) ⟧))
```

Vectors allocated in this way are composed of the appropriate number of individual variables. These are read from and assigned to separately.

The semantics of accessing an array element via its index is given using the `vector-select` funcon. An array access expression in an lvalue position has the following semantics:

```
rule evaluate-lval⟦ E1:Expression [ E2:Expression ] ⟧ ⇒
    vector-select(evaluate-lval⟦ E1 ⟧, evaluate-rval⟦ E2 ⟧)
```

In CinK, multi-dimensional arrays are specified as vectors of vectors. As an illustration of translating array types, consider the declaration statement `int x[2][3];` in C++. Expressing the type of `x` using our auxiliary syntax gives us `([2] ([3] int))`. The translated type is `vectors(2, vectors(3, variables(integers)))`. The construct `allocate-vector` properly allocates variables for such multi-dimensional vectors and returns a compound value of the appropriate type.

A Note on Reuse. The complete funcon definition of CinK available online uses 27 funcons. Of these, 19 have been previously used in the specification of Caml Light and only 8 were introduced in the present work, 3 of which are just abbreviations for longer funcon terms. It is thus possible to conclude that the degree of reuse of funcons between the Caml Light and CinK specifications is high, even if the languages are quite different.

3.9 Configuration

The configuration of the final iteration of our specification is as follows:

```
configuration
    <T>
      <threads>
        <thread multiplicity="*">
          <name> main:Threads </name>
          <k> translate[[ $PGM:TranslationUnit ]] </k>
          <xstack> .List </xstack>
          <context>
            <env> .Map </env>
            <given> no-value </given>
          </context>
        </thread>
      </threads>
      <store> .Map </store>
      <output stream="stdout"> .List </output>
      <input stream="stdin"> .List </input>
    </T>
```

It appears that this configuration could be generated from the K rules defining the funcons used in our specification of CinK. It is unclear to us whether inference of K configurations from arbitrary K rules is possible, and whether it would be consistent with the K configuration abstraction algorithm.

3.10 Sequencing of Side Effects

Following the C++ standard [7], CinK decouples side effects of some constructs to allow delaying memory writes to after an expression value has been returned. This gives compilers more freedom for performing optimisations and during code generation. The newest C++ standard uses a relation *sequenced before* to define how side effects are to be ordered with respect to each other and to value evaluation. The original CinK specification in K [8] uses auxiliary constructs for side effects and uses a bag to collect side effects. An auxiliary sequence point construct forces finalisation of side effects in the bag.

We have experimented with funcons to express decoupled side effects and have developed a preliminary K specification of the relevant funcons. Our solution is based on a pair of funcons. The first funcon encapsulates an expression, which can potentially request to defer side effects. It also maintains a set of deferred side effects which are computed interleaved with the encapsulated expression. Finally, it ensures that all side effect computations have finished before returning the value of the original expression. The other funcon serves to defer a side effect: it signals to the encapsulating funcon that a computation is to be interleaved with the evaluation of the original expression.

4 Funcons in K

We now illustrate our K specification of the syntax and semantics of the funcons and value types used in our component-based analysis of CinK. We specify each funcon and value type in a separate module, to facilitate selective reuse. Since modularity is a significant feature of our specifications, we show some of the specified imports. The complete specifications are available online, together with the K specification of the translation of CinK programs to funcons.

4.1 Expressions

Expressions compute values:

```
module EXPR  imports VALUES
  syntax Expr ::= Values
  syntax KResult ::= Values
```

Our specifications of value types lift the usual value operations to expression funcons, each of which is strict in all its arguments:

```
module INTEGERS  imports EXPR ...
  syntax Expr ::= "int-times" "(" Expr "," Expr ")" [strict]
                | ...
  syntax Values ::= Int
  rule int-times(I1:Int, I2:Int) ⇒ I1 *Int I2
  rule ...
```

In contrast, the conditional expression funcon `if-true(E1,E2,E3)` is strict only in *E1*, and its rules involve unevaluated expression arguments:

```
module IF-TRUE-EXPR  imports EXPR ...
  syntax Expr ::= "if-true" "(" Expr "," Expr "," Expr ")" [strict(1)]
  rule if-true(true, E:Expr, _) ⇒ E
  rule if-true(false, _, E:Expr) ⇒ E
```

We specify a corresponding funcon for conditional commands separately, since it appears that K modules cannot have parametric sorts (although the rules above could be generalised to arbitrary *K* arguments).

4.2 Declarations

```
module DECL  imports BINDINGS
  syntax Decl ::= Bindings
  syntax KResult ::= Bindings
```

Bindings are values corresponding to environments (mapping identifiers to values), and come equipped with some operations that can be used to compose declarations:

```
module BINDINGS  imports DECL
  syntax Bindings ::= bindings(Map)
  syntax Decl ::= "bindings-union" "(" Decl "," Decl ")" [strict]
  rule bindings-union(bindings(M1:Map), bindings(M2:Map)) ⇒
    bindings(M1 M2)
```

We could have included the funcon `bind-value(I,E)` as an operation in the above module, since it is strict in its only expression argument:

```
module BOUND-VALUE  imports ...
  syntax Expr ::= "bound-value" "(" Id ")"
  rule <k> bound-value(I:Id) ⇒ V:Values ...</k>
       <env>... I |-> V ...</env>
```

In contrast, the following funcons involve inspecting or (temporarily) changing the current environment, which is assumed to be in an accompanying cell:

```
module BOUND-VALUE  imports ...
  syntax Expr ::= "bound-value" "(" Id ")"
  rule <k> bound-value(I:Id) ⇒ V:Values ...</k>
       <env>... I |-> V ...</env>

module SCOPE-COMM  imports ...
  syntax Comm ::= "scope" "(" Decl "," Comm ")" [strict(1)]
  rule <k> scope(bindings(Env:Map), C:Comm) ⇒
           reset-env(Env', C) ...</k>
       <env> Env':Map ⇒ Env'[Env] </env>

module ACCUM  imports ...
  syntax Decl ::= "accum" "(" Decl "," Decl ")" [strict(1)]
  rule <k> accum(bindings(Env:Map), D:Decl) ⇒
           reset-env(Env', bindings-union(bindings(Env), D)) ...</k>
       <env> Env':Map ⇒ Env'[Env] </env>
```

The auxiliary operation `reset-env(M,K)` preserves the result of `K` when resetting the current environment to `M`:

```
module RESET-ENV
  syntax K ::= "reset-env" "(" Map "," K ")" [strict(2)]
  rule <k> reset-env(Env:Map, V':KResult) ⇒ V' ...</k>
       <env> _:Map ⇒ Env </env>
```

The `K` argument could be of sort `Expr`, `Decl` or `Comm`. Since we do not use `reset-env` directly in the translation of CinK to funcons, the fact that `reset-env(M,K)` is (semantically) of the same sort as `K` is irrelevant.

4.3 Commands

```
module COMM  imports SKIP
  syntax Comm ::= Skip
  syntax KResult ::= Skip
```

In contrast to the usual style in K specifications, commands compute the unique value `skip:Skip` on normal termination, rather than dissolving. However, this difference does not affect the translation of programs to funcons.

```
module SEQ-DECL  imports ...
  syntax Decl ::= "seq" "(" Comm "," Decl ")" [strict(1)]
  rule seq(skip, D:Decl) ⇒ D
```

As with `if-true`, the funcon `seq(C,X)` is essentially generic in `X`, but its syntax needs to be specified separately for each sort of `X`. In contrast, the sort of `effect(X)` is independent of the sort of `X`, and we can specify it generically:

```
module EFFECT  imports COMM
  syntax Comm ::= "effect" "(" K ")" [strict]
  rule effect(_:KResult) ⇒ skip
```

The specification of `while-true` illustrates reuse *between* funcon specifications:

```
module WHILE-TRUE
  imports COMM
  imports EXPR
  imports IF-TRUE-COMM
  imports SEQ-COMM
  syntax Comm ::= "while-true" "(" Expr "," Comm ")"
  rule while-true(E:Expr, C:Comm) ⇒
         if-true(E, seq(C, while-true(E, C)), skip)
```

4.4 Variables

Variables are themselves treated as values:

```
module VARIABLES  imports ...
  syntax Variables ::= "no-variable"
  syntax Values ::= Variables
```

The specifications of the funcons for allocating, assigning to, and inspecting the values stored in variables are much as usual. For example, the funcon `assign-giving-variable` assigns a value to a variable and then returns the variable:

```
module ASSIGN-GIVING-VARIABLE  imports ...
  syntax Expr ::=
      "assign-giving-variable" "(" Expr "," Expr ")" [strict]
  rule <k> assign-giving-variable(Var:Variables, V:Values) ⇒
              Var ...</k>
       <store>... Var |-> ( _ ⇒ V ) ...</store>
```

4.5 Vector Allocation

The funcon `allocate-vector` serves to allocate a vector of variables. It uses the `allocate` funcon for allocation of element variables.

```
module ALLOCATE-VECTOR  imports ...
  syntax Expr ::= "allocate-vector" "(" Expr ")" [strict]
  rule allocate-vector(vectors(1, T:Types):Types) ⇒
          vector-prefix(allocate-vector(T), vector(.))
  rule allocate-vector(vectors(I:Int, T:Types):Types) ⇒
          vector-prefix(allocate-vector(T),
                        allocate-vector(vectors(int-minus(I, 1), T)))
       when I >Int 1
  rule allocate-vector(T:Types) ⇒ allocate(variables(T))
    when is-vector-type(T) =/=K true
```

4.6 Functions

```
module FUNCTIONS imports ...
  syntax Functions ::= "abstraction" "(" Expr ")"
  syntax Values ::= Functions
```

The operation `abstraction(E)` constructs a value from an *unevaluated* expression `E`. It can then be closed to obtain static bindings for identifiers in `E` (the K specification of the funcon `close(E)` is unsurprising, and omitted here).

```
module APPLY imports ...
  syntax Expr ::= "apply" "(" Expr "," Expr ")" [strict]
  rule apply(abstraction(E:Expr), V:Values) ⇒ supply(V, E)
```

The funcon `supply(E1,E2)` makes the value of `E1` available as '`given`' in the evaluation of `E2`:

```
module SUPPLY-EXPR imports ...
  syntax Expr ::= "supply" "(" Expr "," Expr ")" [strict(1)]
  rule <k> supply(V:Values, E:Expr) ⇒ reset-given(V', E) ...</k>
       <given> V' ⇒ V </given>

module GIVEN imports ...
  syntax Expr ::= "given"
  rule <k> given ⇒ V:Values ...</k> <given> V </given>
```

The specifications of the funcons `throw` and `catch` assume that all cells used to represent the current context of a computation are grouped under a unique context cell. This gives improved modularity: the specification remains the same when further contextual cells are required. In other respects, the specification follows the usual style in the K literature, using a stack of exception handlers:

```
module THROW imports ...
  syntax Comm ::= "throw" "(" Expr ")" [strict]
  rule <k> (throw(V':Values) ~> _) ⇒ (apply(F, V') ~> K) </k>
       <xstack> (F:Functions, K:K, B:Bag) ⇒ . ...</xstack>
       <context> _ ⇒ B </context>

module CATCH imports ...
  syntax Expr ::= "catch" "(" Comm "," Expr ")" [strict(2)]
  rule <k> (catch(C:Comm, F:Functions) ⇒ (C ~> popx)) ~> K </k>
       <xstack> . ⇒ (F, K, B) ...</xstack>
       <context> B:Bag </context>
  syntax K ::= "popx"
  rule <k> popx ⇒ . </k> <xstack> _:ListItem ⇒ . ...</xstack>
```

Funcons `throw` and `catch` have the most complicated definitions of all, yet they are still modest in size and complexity.

5 Related Work

The work in this paper was inspired by a basic specification of the IMP example language in funcons using K by Roșu. IMP contains arithmetic and boolean

expressions, variables, if- and while-statements, and blocks. The translation to funcons is specified directly using K rewrite rules without defining sorted translation functions. The example can be found in the stable K distribution.[4]

CinK, the sublanguage of C++ that we use as a case study in this paper, is taken from a technical report by Lucanu and Șerbănuță [8]. We have limited ourselves to the same subset of C++.

SIMPLE [12] is another K example language which is fairly similar to CinK. The language is presented in two variants: an untyped and a typed one. The definition of typed SIMPLE uses a different syntax and only specifies static semantics. With the component-based approach, we specify a single translation of language constructs to funcons. The MSOS of the funcons defines separate relations for typing and evaluation; in K, it seems we would need to provide a separate static semantics module for each funcon, since the strictness annotations and the computation rules differ.

K specifications scale up to real-world languages, as illustrated by Ellison's semantics of C [4]. The PLanCompS project is currently carrying out major case studies (C#, Java) to examine how the funcon-based approach scales up to large languages, and to test the reusability of the funcon specifications.

Specification of individual language constructs in separate K modules was proposed by Hills and Roșu [6] and further developed by Hills [5, Chap. 5]. They obtained reusable rules by inferring the transformations needed for the rules to match the overall K configuration. The reusability of their modules was limited by their dependence on language syntax, and by the fact that the semantics of individual language constructs is generally more complicated than that of individual funcons.

6 Conclusion

We have given a component-based specification of CinK, using K to define the translation of CinK to funcons as well as the (dynamic) semantics of the funcons themselves. This experiment confirms the feasibility of integrating component-based semantics with the K Framework.

The K specification of each funcon is an independent module. Funcons are significantly simpler than constructs of languages such as CinK, and it was pleasantly straightforward to specify their K rules. However, we would have preferred the K configurations for combination of funcons to be generated automatically.

Many of the funcons used here for CinK were introduced in the component-based specification of Caml Light [3], demonstrating their reusability. The names of the funcons are suggestive of their intended interpretation, so the translation specification alone should convey a first impression of the CinK semantics. Readers are invited to browse the complete K specifications of our funcons online, then compare our translation of CinK to funcons with its direct specification in K [8].

In the future, we are aiming to define the static semantics of funcons in K, so our translation would induce a static semantics for CinK.

[4] http://www.kframework.org

References

1. Chalub, F., Braga, C.: Maude MSOS tool. In: WRLA 2006, ENTCS, vol. 176, pp. 133–146. Elsevier (2007)
2. Churchill, M., Mosses, P.D.: Modular bisimulation theory for computations and values. In: Pfenning, F. (ed.) FOSSACS 2013 (ETAPS 2013). LNCS, vol. 7794, pp. 97–112. Springer, Heidelberg (2013)
3. Churchill, M., Mosses, P.D., Torrini, P.: Reusable components of semantic specifications. In: Proceedings of the 13th International Conference on Modularity, MODULARITY '14, pp. 145–156. ACM, New York (2014)
4. Ellison, C., Roșu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, pp. 533–544. ACM, New York (2012)
5. Hills, M.: A Modular Rewriting Approach to Language Design, Evolution and Analysis. Ph.D. thesis, University of Illinois at Urbana-Champaign (2009)
6. Hills, M., Roșu, G.: Towards a module system for K. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 187–205. Springer, Heidelberg (2009)
7. ISO International Standard ISO/IEC 14882:2011(E) – Programming Language C++ (2011). http://isocpp.org/std/the-standard
8. Lucanu, D., Șerbănuță, T.F.: CinK – an exercise on how to think in K. Technical report TR 12-03 (v2), Faculty of Computer Science, A. I. Cuza University, December 2013. https://fmse.info.uaic.ro/publications/181/
9. Lucanu, D., Șerbănuță, T.F., Roșu, G.: $\mathbb{K}$ framework distilled. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 31–53. Springer, Heidelberg (2012)
10. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. **60–61**, 195–228 (2004)
11. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60–61**, 17–139 (2004)
12. Roșu, G., Șerbănuță, T.F.: K overview and SIMPLE case study. In: Proceedings of the Second International Workshop on the K Framework and Its Applications (K 2011), ENTCS, vol. 304, pp. 3–56. Elsevier (2014)
13. Șerbănuță, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Roșu, G.: The K primer (v3.3). In: Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011), ENTCS, vol. 304, pp. 57–80. Elsevier (2014)