University "Alexandru Ioan Cuza" of Iaşi
Faculty of Computer Science

TECHNICAL REPORT

CinK – an exercise on how to think in $\mathbb{K}$

Dorel Lucanu, Traian Florin Şerbănuţă

TR 12-03, December 2013, (Version 2)

ISSN 1224-9327

Universitatea "Alexandru Ioan Cuza" Iaşi
Facultatea de Informatică
Str. Berthelot 16, 6600-Iaşi, Romania
Tel. +40-32-201090, email: bibl@infoiasi.ro

CinK – an exercise on how to think in $\mathbb{K}$

Dorel Lucanu and Traian Florin Şerbănuţă

Alexandru Ioan Cuza University of Iaşi, Romania

Chapter 1

Introduction

CinK is a kernel of the C++ language we used to experiment with $\mathbb{K}$. The language is used as an working example for teaching classes and is refered in several research papers.

We assume the reader is already familiar with the $\mathbb{K}$ Framework and the $\mathbb{K}$ tool; we here share our experiece in defining languages with some specific features, such as C++. In giving semantics to the language we keep as close as possible to the C++ manual (version 2011). Therefore, for many constructs, the semantics rules are preceded by a quoted text including the semantics description given in the manual.

CinK is defined in several iterations, each iteration being a incremental contribution. Each iteration can be compiled and executes as a stand-alone definition.

The definition can be downloaded from the github site: https://github.com/kframework/cink-semantics and tested with the online tool: http://fmse.info.uaic.ro/tools/K/?tree=examples/cink-semantics/README.md.

1.1 C++ Concepts

In this section we briefly present some C++ concepts that are essential in defining the semantics of CinK. Some of them are quite challenging for the formal semantics.

1.1.1 Memory Model

From C++ manual (1.7):

> The fundamental storage unit in the C++ memory model is the byte. A *byte* is at least large enough to contain any member of the basic execution character set (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address. [Note: The representation of types is described in 3.9. –end note]
>
> A *memory location* is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [Note: Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. –end note]
>
> Two or more threads of execution (1.10) can update and access separate memory locations without interfering with each other.

Our formal semantics uses a more abstract memory model consisting of symbolic cells, each location being either a scalar, or a function definition, or an object, ... The memory locations are defined in the basic iteration and then extended in the iteration defining arrays.

1.1.2 Object Model

From C++ manual (1.8):

> The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. –end note] An object is created by a definition (3.1), by a new-expression (5.3.4) or by the implementation (12.2) when needed. The properties of an object

are determined when the object is created. An object can have a *name* (Clause 3). An object has a *storage duration* (3.7) which influences its *lifetime* (3.8). An object has a *type* (3.9). The term *object type* refers to the type with which the object is created. Some objects are *polymorphic* (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the expressions (Clause 5) used to access them.
Objects can contain other objects, called subobjects. A subobject can be a member subobject (9.2), a base class subobject (Clause 10), or an array element. An object that is not a subobject of any other object is called a complete object.
For every object x, there is some object called the complete object of x, determined as follows:

- If x is a complete object, then x is the complete object of x.
- Otherwise, the complete object of x is the complete object of the (unique) object that contains x.

The objects will be defined later in the iteration specifying the semantics of the classes.

1.1.3 Side Effects

Side effects are very important in giving semantics to C++ constructs, e.g., expressions, and therefore it is necessary to know exactly what a side effect means. Here is the definition from the C++ manual:

> Accessing an object designated by a volatile glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. Evaluation of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a volatile object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the volatile access may not have completed yet.

1.1.4 Sequenced before

"Sequenced before" is specific to languages where the evaluation of the expressions has side effects. Obviously, this is the case for C and C++ languages. "Sequenced before" defines a partial order relation over the evaluation of the subexpressions and over the side effects generated by these evaluations. Here is the definition for sequenced before from the C++ manual (1.9):

> *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are unsequenced. [Note: The execution of unsequenced evaluations can overlap. –end note] Evaluations A and B are indeterminately sequenced when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [Note: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. –end note]
> Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.
> Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced.
> [Note: In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. –end note]
> The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.

Sources for the side effects in C++ are are the assignment operators, postfix increment and decrement operators. Sequenced before is eesential in defining the behavior of programs.

1.1.5 Types

From the C++ manual (3.9):

> [Note: 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (1.8), references (8.3.2), or functions (8.3.5). –end note]
> ...
> The object representation of an object of type T is the sequence of N unsigned char objects taken up by the object of type T, where N equals sizeof(T). The value representation of an object is the set of bits that hold the value of type T.
> ... Arithmetic types (3.9.1), enumeration types, pointer types, pointer to member types (3.9.2), std::nullptr_-t, and cv-qualified versions of these types (3.9.3) are collectively called *scalar types.*
> ...

Fundamental types

> Objects declared as *characters (char)* shall be large enough to store any member of the implementation's basic character set. ...
> There are five standard signed *integer types*: "signed char", "short int", "int", "long int", and "long long int". ...
> Values of type *bool* are either true or false.
> ...
> There are three *floating point* types: float, double, and long double. The type double provides at least as much precision as float, and the type long double provides at least as much precision as double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double. The value representation of floating-point types is implementation-defined. Integral and floating types are collectively called arithmetic types. ...
> The *void* type has an empty set of values. The void type is an incomplete type that cannot be completed. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type cv void (5.4). An expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of ?: (5.16), as the operand of typeid or decltype, as the expression in a return statement (6.6.3) for a function with the return type void, or as the operand of an explicit conversion to type cv void.

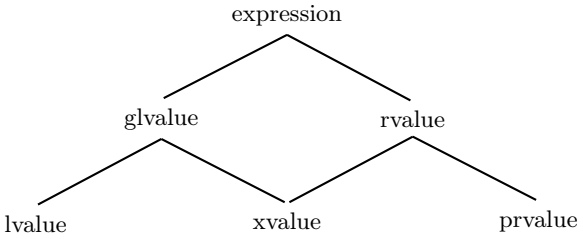
Compound types

> Compound types can be constructed in the following ways:
>
> - *arrays* of objects of a given type, 8.3.4;
> - *functions*, which have parameters of given types and return void or references or objects of a given type, 8.3.5;
> - *pointers* to void or objects or functions (including static members of classes) of a given type, 8.3.1;
> - *references* to objects or functions of a given type, 8.3.2. There are two types of references:
>   - lvalue reference
>   - rvalue reference
> - *classes* containing a sequence of objects of various types (Clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities(Clause 11);
> - *unions*, which are classes capable of containing objects of different types at different times, 9.5;
> - *enumerations*, which comprise a set of named constant values. Each distinct enumeration constitutes a different enumerated type, 7.2;
> - *pointers* to non-static class members, which identify members of a given type within objects of a given class, 8.3.3.

In CinK we consider only a part of the above types, but we try to keep the taxonomy and terminology.

1.1.6 Lvalues and Rvalues

In C++ the expressions are splitted in several categories:



From the C++ manual (3.10):

– An lvalue (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. –end example]
– An xvalue (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references (8.3.2). [Example: The result of calling a function whose return type is an rvalue reference is an xvalue. –end example]
– A glvalue ("generalized" lvalue) is an lvalue or an xvalue.
– An rvalue (so called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.
– A prvalue ("pure" rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue. –end example]

In order to keep the definition as simple as possible, we consider only two of these categories: *lvalues* and *rvalues*. This classification of expressions is essential in defining their semantics since the evaluation of an lvalue expression is different from a rvalue one.

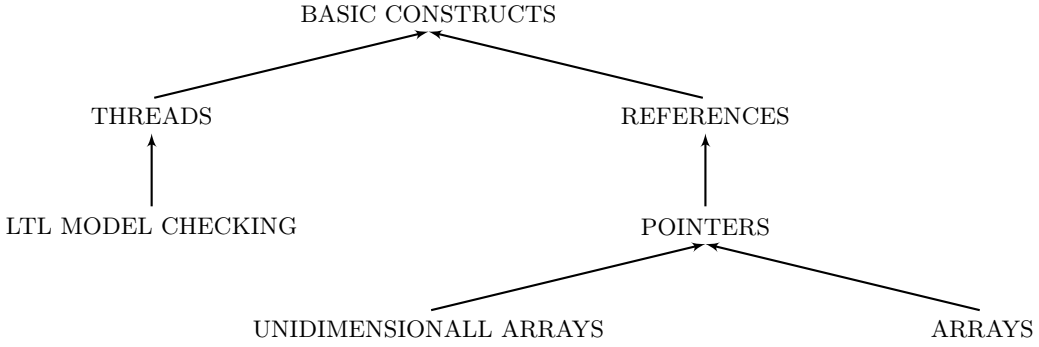
1.2 Structure of the Report

In this report we briefly present the $\mathbb{K}$ definition of CinK. The most part of the text is automatically generated with the $\mathbb{K}$ tool, therefore there are some differences between the source code and the pdf version. For instance, the terminal syntax declarations are enclosed into quotes but in the pdf version these are not displayed. In the pdf version, for ease of reading reasons, there are used different fonts in order to distinguish between different syntactic categories.

The report is not intended to be an introduction to $\mathbb{K}$ or a complete formal semantics of C++. Instead we are focusing on some challenging features of this language. Such features include a clear distinction between l-values and r-values, declaration of aliases, and parameters passing mechanisms, arrays that can be symbolically defined. We also extend the definition with a small property language, including LTL formulas, and we show how the $\mathbb{K}$ tool is used together with Maude system for analizing CinK programs. We exemplify how the $\mathbb{K}$ tool can be used to execute programs on symbolic input data.

The $\mathbb{K}$ definition of CinK is continuously evolving, so this report presents the status of this definition at the publication/revision date.

The iterations included in this version have the following hierarchy:



Here is a brief description of the iterations:

Iteration	Features included
Basic	`int` and `bool` types, local and global variables, expressions, side effects, sequenced before, lvalues and rvalues, functions, call by value, symbolic execution
Threads	minimal support for threads, it follows to be extended
LTL Model Checking	LTL formulas expressing properties of programs, use of the LTL model checker
References	reference types, declaration of aliases, & operator, call by reference
Pointers	pointer type, indirection operator, relation with & operator (they are inverse each other), `new` and `delete` operators
Unidimensional Arrays	declarations, representation, uses of 1D arrays as parameters, 1D arrays and pointers
Arrays	declarations, representation, uses of arrays as parameters, arrays and pointers, symbolic execution

Chapter 2

Iteration #1: Basic Constructs

This definition includes the starting iteration, where the basic constructs of CinK are defined. The main implemented features include:

- `bool` and `int` types
- aritmetic and Boolean expressions, including increment and decrement operators, the comma operator, function call
- lvalues and rvalues expressions
- sequence points
- imperative statements
- function definitions
- local and global variables
- a call-by-value mechanism for binding the function call arguments

Restrictions comparing with C++:

- local variables can be declared only in the main body of the function. The support for local variables defined inside a block will be added later. The main reason we did not it here is because we want to use the LTL model checker for program analysis in a subsequent iteration. The use of the local variables in blocks could lead to an infinite state space.
- not all operators on `int` and b`bool` are defined; since their specifications is similar to the presented ones, they are let as exercises. The same is true for the imperative statements.

Among the examples included with this definition we mention:

`nondet.cink` - a defined program with non-determinsitic behavior

`undefined.cink` - a program with undefined behavior

`sumn.cink` - a program with simbolic input

The programs in the subfolder `tests` are only for testing purposes.

2.1 Syntax

MODULE CINK-BASIC-SYNTAX

2.1.1 Types

In this iteration we consider only a subset of the fundamental types:

SYNTAX *FundType* ::= `int`
| `bool`
| `void`

Any fundamental type is a scalar type:

SYNTAX *ScalType* ::= *FundType*

Any scalar type is a type:

SYNTAX *Type* ::= *ScalType*

2.1.2 Declarations

Declarations are used to introduce names for variables and function names together with their return types.

SYNTAX *Decl* ::= *Type* *Exp*

2.1.3 Expressions

We included in CinK a small subset of operators from C++ language, the missing ones could being easily added in a similar way.

A major feature of the C++ expressions is that given by the relation "sequenced before", which defines a partial order over the evaluation of subexpressions. This can be easily expressed in $\mathbb{K}$ using the `strict` attribute to specify the evaluation order for the operands. If the operands of an operator are unsequenced, then the operator will have the attribute `strict`, meaning that they are evaluated in strict nondeterministic order. If an operand A is sequenced before B then the evaluation order is expressed as argument of the `strict` attribute. For instance, since the operands of the binary operators are unsequenced, these operators are strict in both arguments. Hence the behavior of some programs could be undefined or non-determinsitic because the evaluation of the arguments could have side-effects.

The second major feature is given by the classification of the expression in rvalues and lvalues. For instance, the arguments of a binary operator are evaluated as rvalues and their result is a rvalue, too, whilst the argument of the prefix increment/decrement and their result are lvalues. Therefore the value of the strict attribute for such operators is a list of two sub-attributes: a `context` sub-attribute having as value the label `rvalue` and a a `result` sub-attribute having as value the result type of the argument(s). The expression are considered lvalues by default. This will be explained later in the module defining semantics, when we define the values of the language.

In contrast to the other examples, the function call expression is strict only in the first argument (the function name) because the evaluation of the arguments is depending on the binding mechanism of the corresponding argument: this can be call-by-value or call-by-reference. In this iteration we consider only the former one, the latter will be implemented together with the references in a next iteration.

The primitive constructs for expressions can be variable names, which are identifiers, and values modelled by the sort `Val`. A subset of values, modelled by the sort `ScalVal`, is given by the constants of the scalar types.

We prefer to declare the standard input/output streams `cin` and `cout` as being values; the reason for this decision will be explained later, when we give semantics to the reading/writting operators.

SYNTAX *ScalVal* ::= *Bool*
| *Int*
| *String*

SYNTAX *Val* ::= `cout`
| `cin`
| *ScalVal*

SYNTAX *Exp* ::= *Id*
| *Val*
| (*Exp*) [bracket]
| *Exp*(*Exps*) [strict(1(context(rvalue))), funcall]
| `++` *Exp* [strict(all(result(Val))), prefinc]
| `-` *Exp* [strict(all(result(Val))), prefdec]
| *Exp* `++` [strict(all(result(Val))), postinc]

| *Exp* `-` [strict(all(result(Val))), postdec]
| *Exp* `*` *Exp* [strict(all(context(rvalue))), multiply]
| *Exp* `/` *Exp* [strict(all(context(rvalue))), divide]
| *Exp*%*Exp* [strict(all(context(rvalue))), modulo]
| *Exp* `+` *Exp* [strict(all(context(rvalue))), plus]
| *Exp* `-` *Exp* [strict(all(context(rvalue))), minus]
| *Exp* `<` *Exp* [strict(all(context(rvalue), result(ScalVal))), lessthan]
| *Exp* `>` *Exp* [strict(all(context(rvalue), result(ScalVal))), greatthan]
| *Exp* ≤ *Exp* [strict(all(context(rvalue), result(ScalVal))), lessequal]
| *Exp* ≠ *Exp* [strict(all(context(rvalue), result(ScalVal))), notequal]
| *Exp* `==` *Exp* [strict(all(context(rvalue), result(ScalVal))), equality]
| `!` *Exp* [strict(all(context(rvalue), result(Bool))), negation]
| *Exp* && *Exp* [strict(1(context(rvalue), result(Bool))), conjunction]
| *Exp* `||` *Exp* [strict(1(context(rvalue), result(Bool))), disjunction]
| *Exp* `«` *Exp* [strict(1, 2(context(rvalue))), write]
| *Exp* `»` *Exp* [strict, read]
| *Exp* `=` *Exp* [strict(1, 2(context(rvalue))), assign]
| `endl`
| *Exp*, *Exp* [strict(1), comma]

2.1.4 Statements

For now, we include in CinK only a subset of the imperative statements: expresion statement, bloc, sequential composition, while, and conditionals. Other imperative statements, like `for` or `do-until` are easy to define and we let them as exercises.

SYNTAX *Stmt* ::= *Decl* `;` [klabel(declStmt)]
| *Exps* `;` [strict]
| `#include <iostream>`
| `using namespace std;`
| {}
| {*Stmts*}
| `while` (*Exp*)*Stmt*
| `return` *Exp* `;` [strict(all(context(rvalue)))]
| *Decl*(*Decls*){*Stmts*}
| `if` (*Exp*)*Stmt* `else` *Stmt* [strict(1(context(rvalue)))]
| `if` (*Exp*)*Stmt*

A program is a sequence of statements:

SYNTAX *Pgm* ::= *Stmts*

SYNTAX *Stmts* ::= *Stmt*
| *Stmts* *Stmts*

2.1.5 List of Declarations

The above definitions are using lists of declarations, which ais declared as follows:

SYNTAX *Decls* ::= *List*{*Decl*,“,”}

2.1.6 List of Expressions

The definition for the list of expressions is more tricky because of the comma operator. According to the C++ manual (5.18), the expression lists are special comma expressions:

> In contexts where comma is given a special meaning, [Example: in lists of arguments to functions (5.2.2) and lists of initializers (8.5) -end example] the comma operator as described in Clause 5 can appear only in parentheses.

The main difference between the expression lists and comma expressions is given by the empty list. Therefore we have to separately define it. Mainly, the following declaration says 1) the empty list `.Exps` is an expression list (denoted by the sort `Exps`), and 2) an expression, in particular any comma expression, is an expression lists.

SYNTAX *Exps* ::= [onlyLabel, klabel('.Exps)]
| *Exp*

SYNTAX *Exp* ::= `.Exps`

An expression can be converted into an expression list by a rule of the form

```
E => append(E, .Exps)
```

The customised syntax for symbolic variables:

SYNTAX *Variable* ::= *Token*{"[*A* − *Z*][*a* − *zA* − *Z*0 − 9] ∗ [\ :][*a* − *zA* − *Z*] + "} [variable, onlyLabel]

SYNTAX *Int* ::= *Variable*

END MODULE

2.2 Semantics

MODULE CINK-BASIC-SEMANTICS

2.2.1 Memory Model

We use an abstract memory model consisting of (automatically generated) symbolic values for locations. The locations are splitted in two categories: scalar locations, which can store scala values, and compound locations, which can store values of compound types. A location is identified with its address. These sort for locations is `Loc` and that for scalar locations is `ScalLoc`. We use a special constant `noLoc` for non-defined locations. The locations are used as values for pointer types and therefore they are subsodrted to `ScalVal`.

SYNTAX *ScalLoc* ::= `noLoc`

SYNTAX *Loc* ::= *ScalLoc*

SYNTAX *ScalVal* ::= *Loc*

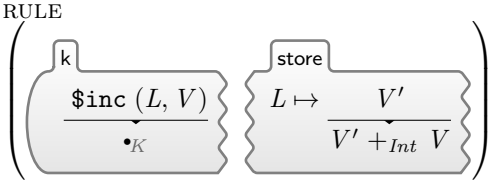
Memmory operations and side effects

The memory operations and the side effects are described by internal operations whose semantics is self explained.

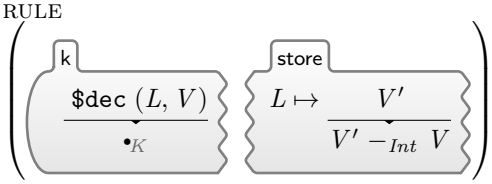
SYNTAX *Exp* ::= `$inc` (*K*, *ScalVal*)
| `$dec` (*K*, *ScalVal*)
| `$update` (*K*, *Val*) [strict(1(result(Loc)))]

| `$lookup` (K)

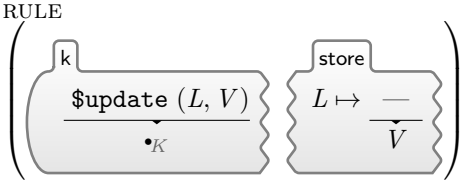
RULE



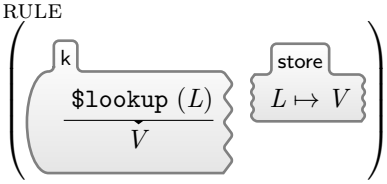
RULE



RULE



RULE



2.2.2 Values.

The values are a very important syntactic category in the definition of any language; with them, we are able to know when the evaluation of an expression is finished. This piece of information is crucial for the heating and cooling rules.

In order to keep the definition as simple as possible, we let the expressions be evaluated always to lvalues and then use the context `rvalue` to convert them into rvalues whenever this is needed. To be more precise, we consider the example of the expression `++x + y`. According to C++ semantics, the evaluation of the operator `++` returns an lvalue. When the operator `+` is evaluated this must be converted into an rvalue. Therefore the strict attribute of `+` has two sub-attributes: `context(rvalue)` and `result(ScalVal)`. To understand better this mechanism, we present the heating/cooling rules generated by strict attribute for the `+` operator:

```
E1:Exp + E2:Exp => rvalue(E1) ~> HOLE + E2 when notBool isScalVal(E1)
E1:Exp + E2:Exp => rvalue(E2) ~> E2 + HOLE when notBool isScalVal(E2)
I1:ScalVal ~> HOLE + E2 => I1 + E2
I2:ScalVal ~> E1 + HOLE => E1 + I2
```

For instance, the above expression will be derived into the following sequence of computations (note that `rvalue` is strict, too):

```
++x + y => rvalue(y) ~> ++x + HOLE => y ~> rvalue(HOLE) ~> ++x + HOLE
```

After the variable `y` is evaluated to its lvalue, it must be converted to an rvalue. Therefore we use a label `lval(`l`)` saying that the location l plays the role of lvalue and the conversion is realised by a rule of the form `rvalue(lval(l)) =>` *the value stored at the location* `l`.

We already defined the subset of values that is part of the syntax, namely the values given by the scalar types. Here we extend the set o values with intermediate constructs needed to execute programs. Such a value is`noVal`, which "paradoxically" denotes in fact "no value". The lambda abstractions are used for storing the functions. Similar to other $\mathbb{K}$ examples (IMP, IMP++, SIMPLE), the functions are stored similar to variables and therefore their definitions are seen as values. Since they cannot split into sub-values, we define them as scalar values.

SYNTAX $ScalVal ::=$ `noVal`
| $\lambda\, Decls \bullet Stmts$

SYNTAX $Val ::= \texttt{lval}\ (Loc)$

The empty expression is a value as well:

RULE

$$\frac{isVal(\ \texttt{.Exps})}{\text{true}}$$

Any value is a K result:

SYNTAX $KResult ::= Val$

The definition of the strict context `rvalue`:

SYNTAX $Exp ::= \texttt{rvalue}\ (Exp)$ [context(result(Val)), strict, klabel(rvalue)]

When an lvalue is in an rvalue context, evaluate it (this can be seen as a conversio from lvalues to rvalues):

RULE

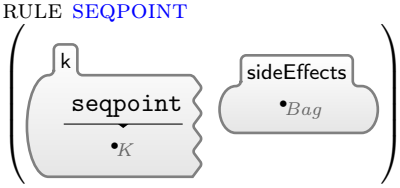
$$\texttt{rvalue}\ (\ \frac{\texttt{lval}\ (L)}{\texttt{\$lookup}\ (L)}\ )$$

2.2.3 Seqpoints

The semantics of a sequence point consists of executing all the side effects collected in the evaluation of the current expression. This is required, e.g., by the complete evaluation of a full expression and the comma operator. So, the semantical statement "seqpoint" has the role to check that the all side effects are executed, i.e. the cell sideEffects is empty.

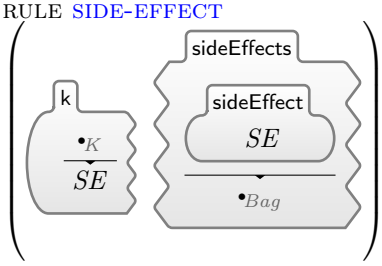
SYNTAX $K ::= \texttt{seqpoint}$

RULE SEQPOINT



The side effects can be also executed concurrently with the other evaluation steps of the current expression:

RULE SIDE-EFFECT



2.2.4 Auxiliary constructs.

`execute` is used to start the computation of a program; and `noname` for the initial name of a thread.

SYNTAX $K ::= \texttt{execute}$

The next two constructors are used for storing the environment and the rest of the computation in the call stack (`fstack`).

SYNTAX $\quad \mathit{ListItem} ::= (\mathit{List}, K)$
$\qquad\qquad\qquad\quad \mid\ [\mathit{Map}]$

2.2.5 Desugaring

In order to have a minimal set of rules, some syntactic constructs are desugared.

The desugaring rule for the if-then statement:

RULE

$$\left(\frac{\texttt{if}\ (B)\mathit{St}}{\texttt{if}\ (B)\mathit{St}\ \texttt{else}\ \{\}}\right)$$

The desugaring rule for statements declaring multiple variables.

RULE

$$\left(\frac{T\ \ (E, E')\ \texttt{;}}{T\ \ E\ \texttt{;}\ \ T\ \ E'\ \texttt{;}}\right)$$

Desugaring rule for variable declarations with initialization. In this iterartion only variables of fundamental types can be initialized.

RULE

$$\left(\frac{T\ \ \mathit{E1}\ \texttt{=}\ \mathit{E2}\ \texttt{;}}{T\ \ \mathit{E1}\ \texttt{;}\ \ \texttt{getName}\,(\mathit{E1})\ \texttt{=}\ \mathit{E2}\ \texttt{;}}\right)$$

2.2.6 Declarations.

Function declaration: a function is a stored similar to a variable, where the value stored in the associated location is the lambda abstraction of the function.

RULE

$$\left(\langle \frac{(\mathit{Decl}(\mathit{Xl})\{\mathit{Sts}\})}{\bullet_K} \cdots \rangle_{\mathsf{k}} \ \ \langle \cdots \frac{\bullet_{\mathit{Map}}}{(\ \texttt{getName}\ (\mathit{Decl}) \mapsto L)} \cdots \rangle_{\mathsf{env}} \ \ \langle \cdots \frac{\bullet_{\mathit{Map}}}{(L \mapsto \lambda\, \mathit{Xl} \bullet \mathit{Sts})} \cdots \rangle_{\mathsf{store}} \right)$$

when $\texttt{fresh}\,(L)$

The rules for variable declarations for the fundamental types:
From the C++ manual (6.7):

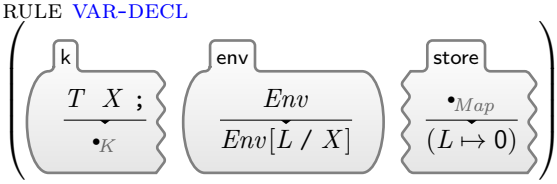
> The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place.

From the C++ manual (8.5):

> To zero-initialize an object or reference of type T means:
> - if T is a scalar type (3.9), the object is set to the value 0 (zero), taken as an integral constant expression, converted to T;

In this iteration the values and expressions are not typed, so the conversion cannot be defined.

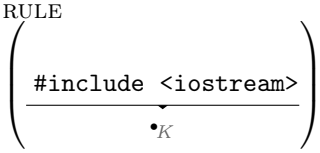
RULE VAR-DECL



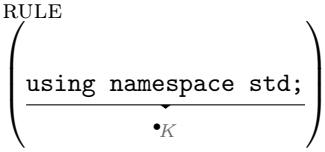
when `fresh` (L)

The following two constructs have no semantics yet; they are used now only for having a full compatibility with C++, e.g., the CinK programs can be compiled with a C++ compiler.

RULE

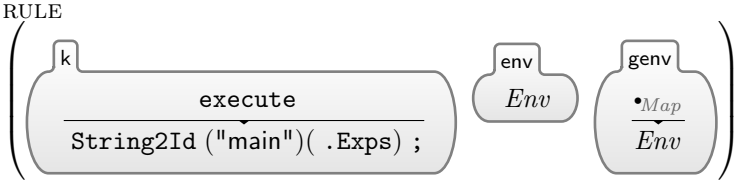


RULE



The auxiliary construct `execute` is used to initialize the execution of a program, which for the case of CinK consists of the call of the main function.

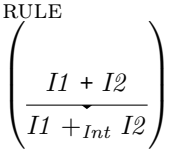
RULE



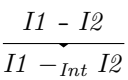
2.2.7 Expressions Evaluation.

The following expressions are strict and therefore their semantics is given only for the case the operands are values.

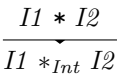
RULE



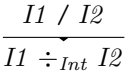
RULE



RULE

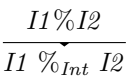


RULE



when $I2 =/=_{Int} 0$

RULE



when $I2 =/=_{Int} 0$

RULE

$$\frac{I1 < I2}{(I1 <_{Int} I2)}$$

RULE

$$\frac{I1 > I2}{(I1 >_{Int} I2)}$$

RULE

$$\frac{I1 \leq I2}{(I1 \leq_{Int} I2)}$$

RULE

$$\frac{I1 \neq I2}{(I1 =/=_{Int} I2)}$$

RULE

$$\frac{I1 == I2}{(I1 ==_{Int} I2)}$$

RULE

$$\frac{\texttt{true} \;\&\&\; B}{B}$$

RULE

$$\frac{\texttt{false} \;\&\&\; B}{\texttt{false}}$$

RULE

$$\frac{\texttt{true} \;||\; B}{\texttt{true}}$$

RULE

$$\frac{\texttt{false} \;||\; B}{B}$$

RULE

$$\frac{\texttt{!}\; \texttt{false}}{\texttt{true}}$$

RULE

$$\frac{\texttt{!}\; \texttt{true}}{\texttt{false}}$$

RULE

$$\frac{\texttt{endl}}{\texttt{"\textbackslash n"}}$$

The prefix increment/decrement operator

Here is the description of the the prefix increment from the C++ manual:

> The operand of prefix ++ is modified by adding 1, or set to true if it is bool (this use is deprecated). The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a completely-defined object type. The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. If x is not of type bool, the expression ++x is equivalent to x+=1 [Note: See the discussions of addition (5.7) and assignment operators (5.17) for information on conversions.]

The semantics consists of executing the corresponding side effect and returning the involved location:

RULE

$$\frac{\texttt{++ lval}\; (L)}{\texttt{\$inc}\; (L, 1) \curvearrowright \texttt{lval}\; (L)}$$

RULE

$$\frac{\texttt{-}\ \texttt{lval}\ (L)}{\texttt{\$dec}\ (L, 1) \curvearrowright\ \texttt{lval}\ (L)}$$

The postfix increment/decrement operator

Here is the description of this operator from the manual of C++ 2011 (5.2.6):

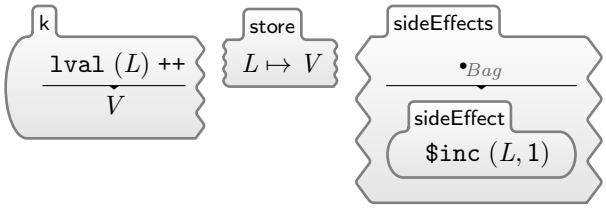
> The value of a postfix ++ expression is the value of its operand. [Note: the value obtained is a copy of the original value -end note] The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a complete object type. The value of the operand object is modified by adding 1 to it, unless the object is of type bool, in which case it is set to true. [Note: this use is deprecated, see Annex D. -end note] The value computation of the ++ expression is sequenced before the modification of the operand object. ... The result is a prvalue.

From the Appendix D:

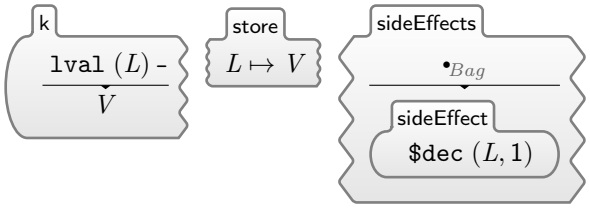
> D.1 The use of an operand of type bool with the ++ operator is deprecated (see 5.3.2 and 5.2.6)."

We add a new cell that collects the side-effects during the evaluation of a full-expression. Therefore the result value is the value stored at the computed location and its side-effect is added to the cell with the side-effects. Note that we have a kind of lvalue to rvalue conversion here.

RULE POSTINC

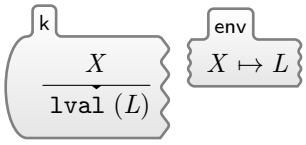


RULE POSTDEC



The evaluation of a variable name as an lvalue:

RULE



The memory update is given by the assignment operator. An assignemnt, after the reduction of the arguments, has in the lef-hand side a location (= the lvalue designated by the lhs.) and in the right-hand side an rvalue. From C++ manual:

> The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand. .... In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression.

The first requirement is solved by the strict attribute and the second one by the next rule. The semantics of compound assignment operators is let as exercises.

RULE ASSIGN

$$\frac{\texttt{lval}\ (L)\ \texttt{=}\ V}{\texttt{\$update}\ (L, V) \curvearrowright \texttt{lval}\ (L)}$$

The comma operator. From C++ manual:

> A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded value expression (Clause 5). Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression. The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, ... If the value of the right operand is a temporary (12.2), the result is that temporary.

The sequenced before requirement is accomplished by adding a seqpoint:

RULE

$$\frac{V, E}{\texttt{seqpoint}\curvearrowright E}$$

2.2.8 Control Statements.

As usual, the `while` statement is desugared using the `if-then-else` statement.

RULE

$$\left(\frac{\texttt{while}\ (B) St}{\texttt{if}\ (B)\{St\ \ \texttt{while}\ (B) St\}\ \texttt{else}\ \{\}}\right)$$

Since `if` is strict in the first argument, which is a boolean expression, proceed by case-analysis on the result values:

RULE

$$\left(\frac{\texttt{if}\ (\mathsf{false})\text{—}\ \texttt{else}\ St}{St}\right)$$

RULE

$$\left(\frac{\texttt{if}\ (\mathsf{true}) St\ \texttt{else}\ \text{—}}{St}\right)$$

The semantics of the expression statement consists of removing the value obtained by evaluating the expression. Recall that the statement is strict.
From the C++ manual:

> Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated

We use a seqpoint in order to sequence the side effects.

RULE

$$\left(\frac{V\ \texttt{;}}{\texttt{seqpoint}}\right)$$

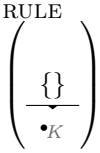
Block. Note that in this iteration we assume that the blocks do not include variable declarations. This will be added in a future iteration.
– the case of non-empty block

RULE

$$\left(\frac{\{Sts\}}{Sts}\right)$$

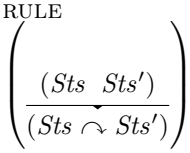
– the case of the empty block:

RULE



The sequential composition is just a sequence of computations:

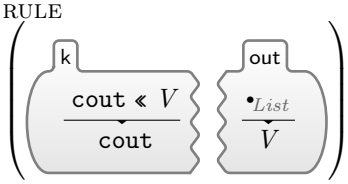
RULE



2.2.9 Input/Output Statements.

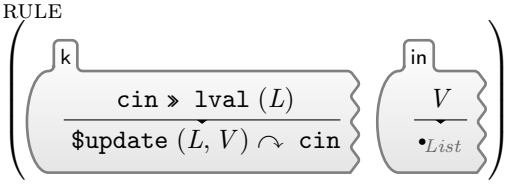
Writting in the the standard stream `cout`:

RULE



In order to read from the standard stream `cin`, the expression from the right-hand side must be evaluated to an l-value:

RULE



2.2.10 The function call expression

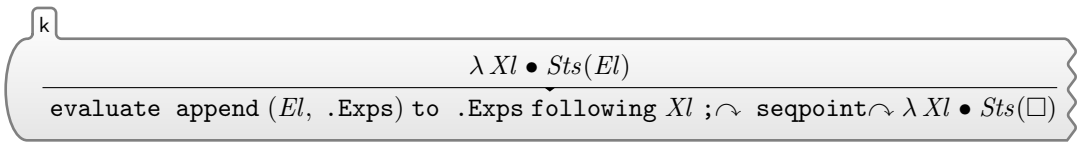
The function name is evaluated to its value, which is a lambda abstraction: *Xl* is the list of parameters, *Sts* is body of the function. The FUNCTION-CALL rule pushes the calling context, i.e., the remainder of the computation K and environment stack (including the current environment) on top of the function stack, while the RETURN rule uses the information there to restore the environment and computation of the caller. Since the evaluation strategy for the second argument is depending on the binding specification in the function signature, the function call expression is declared strict only in its first argument. Note that in this iteration we consider only the call-by-value binding mechanism. The call-by-reference mechanism will be considered together with references in the next iteration.

From the C++ manual:

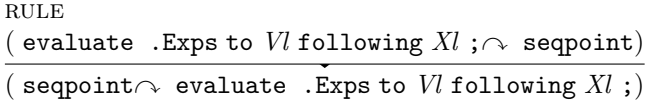
> When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. [Note: Such initializations are indeterminately sequenced with respect to each other (1.9) -end note] ...
> During the initialization of a parameter, an implementation may avoid the construction of extra temporaries by combining the conversions on the associated argument and/or the construction of temporaries with thevinitialization of the parameter (see 12.2). The lifetime of a parameter ends when the function in which it is defined returns. The initialization and destruction of each parameter occurs within the context of the calling function. ...
> [Note: a function can change the values of its non-const parameters, but these changes cannot affect the values of the arguments except where a parameter is of a reference type (8.3.2)...-end note]

The rule defining the evaluation of a function call expression evaluates first the actual parameters, and then binds the values to the formal parameters and executes the body, while saving the calling context in case of an abrupt return. This is done by mimic the heating-cooling mechanism (the first rule is a heating-like one, and the second a cooling-like one).

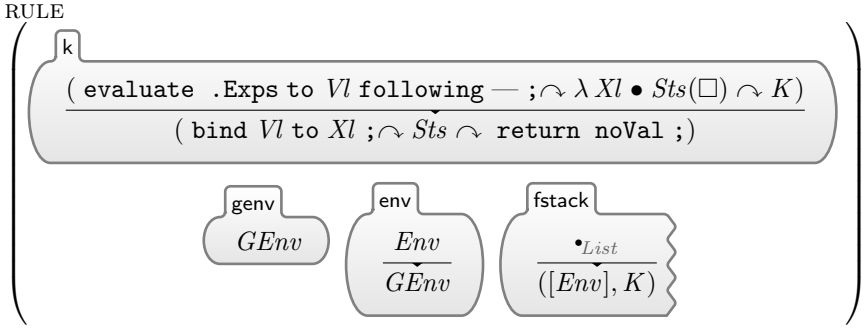
RULE



RULE



RULE

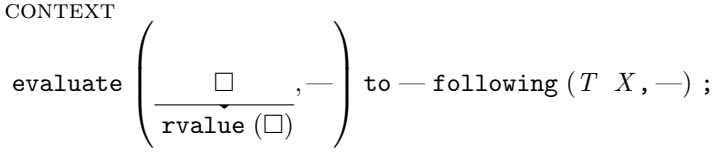


To evaluate actual parameters according to their declared strategy we will make use of the power of $\mathbb{K}$ evaluation contexts. The actual parameters must be evaluated using the `evaluate` construct and `following` the list of formal parameters.

SYNTAX $K ::=$ `evaluate` K `to` K `following` $Decls$ `;` [evaluate]

For a formal parameter declared with the call-by-value mechanism, the corresponding argument expression must be evaluated to an rvalue as specified by the following contextual declaration:

CONTEXT

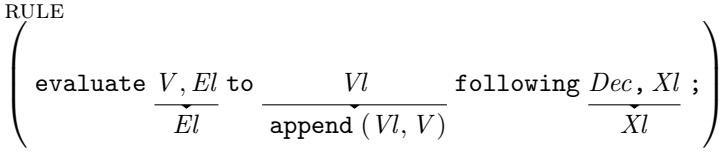


result(ScalVal)

This second context uses again the special type of context used above for `evaluate`, by requesting that the expression on position □ be evaluated as an lvalue.

The following two rules, together with the strict evaluation strategy for the comma operator complete the semantics of `evaluate` by recursing into the lists:

RULE



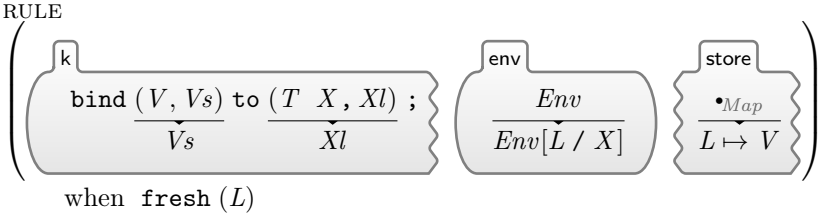
Binding mechanisms

Similarly to the evaluation rules, the binding rules are also different for the two parameter passing styles. As we have already seen, the binding is performed using an auxiliary construction:

SYNTAX $K ::=$ `bind` K `to` $Decls$ `;`

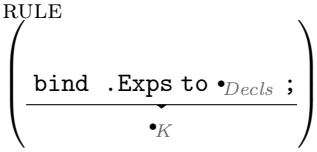
For call-by-value, the passed value V is stored into a new memory location which is bound to the formal parameter:

RULE



Finally, once all parameters have been bound, the binding construct dissolves:

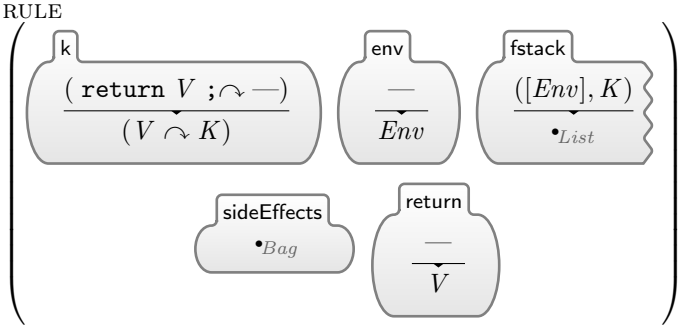
RULE



2.2.11 Function Return.

The environment and the code are restored before returning back to the calling code. The side effects are sequenced before; this ensured by the fact the cell sideEffects is empty.

RULE



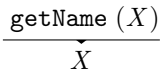
2.2.12 Auxiliary Functions and Rules.

Here we include the semantics for the auxiliary constructs, used to give semnatics for CinK.

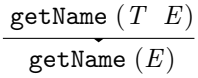
The next operator returns the variable name from an expression occurring in a declaration:

SYNTAX $Id ::=$ `getName` (K) [function]

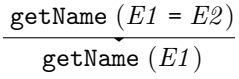
RULE



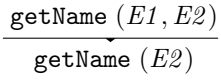
RULE



RULE



RULE



Append function:

SYNTAX $Exp ::= \texttt{append}\ (Exp, Exp)$ [function]

RULE

$$\frac{\texttt{append}\ ((E, El), E')}{E,\ \texttt{append}\ (El, E')}$$

RULE

$$\frac{\texttt{append}\ (\ \texttt{.Exps},\ \texttt{.Exps})}{\texttt{.Exps}}$$

RULE

$$\frac{\texttt{append}\ (\ \texttt{.Exps}, E)}{E,\ \texttt{.Exps}}$$

when $E \neq_K \texttt{.Exps}$

RULE

$$\frac{\texttt{append}\ (E, E')}{E, E'}$$

END MODULE

2.3 The Main Module

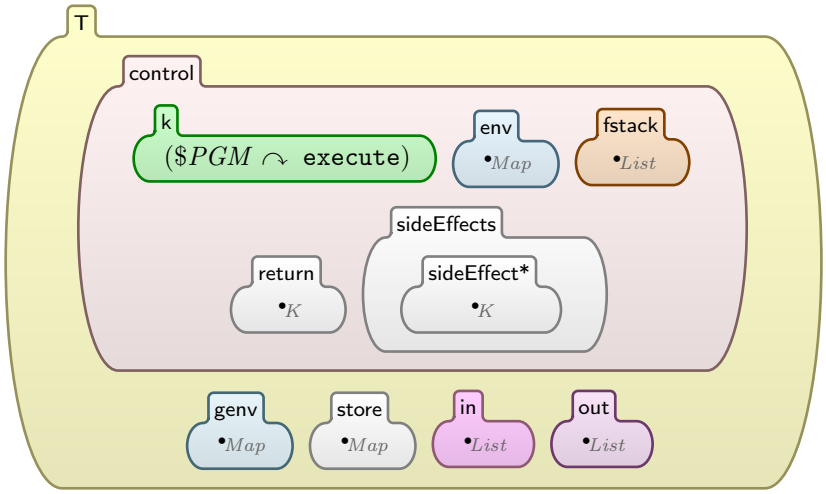
Includes the modules defining the syntax and semantics, and defines the main configuration of the language.

MODULE CINK

2.3.1 Configuration.

The configuration is standard for such languages (compare it to that of IMPPP and SIMPLE).

CONFIGURATION:



END MODULE

Chapter 3

Iteration #2: Threads

This iteration includes the extension of the starting iteration with a minimal support for threads. Fom the C++ manual:

> A thread of execution (also known as a thread) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread.
> [Note: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. âĂŤend note]
> Every thread in a program can potentially access every object and function in a program.
> Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads.
> [Note: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. âĂŤend note]
> Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.
> Implementations should ensure that all unblocked threads eventually make progress.
> [Note: Standard library functions may silently block on I/O or locks. Factors in the execution environment, including externally-imposed thread priorities, may prevent an implementation from making certain guarantees of forward progress. âĂŤend note]
> The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below.
> [Note: In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. âĂŤend note]

Imported Modules

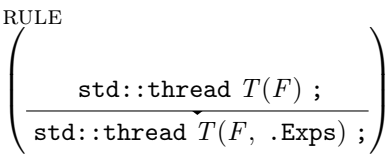
Two modules are imported: `CINK-BASIC-SYNTAX` and `CINK-BASIC-SEMANTICS`, including the syntax and the semantics, respectively, of the basic constructions.

3.1 The New Modules

MODULE CINK-THREADS-SYNTAX

SYNTAX *Stmt* ::= `std::thread` *Id*(*Id*, *Exps*) `;`
| `std::thread` *Id*(*Id*) `;`

RULE



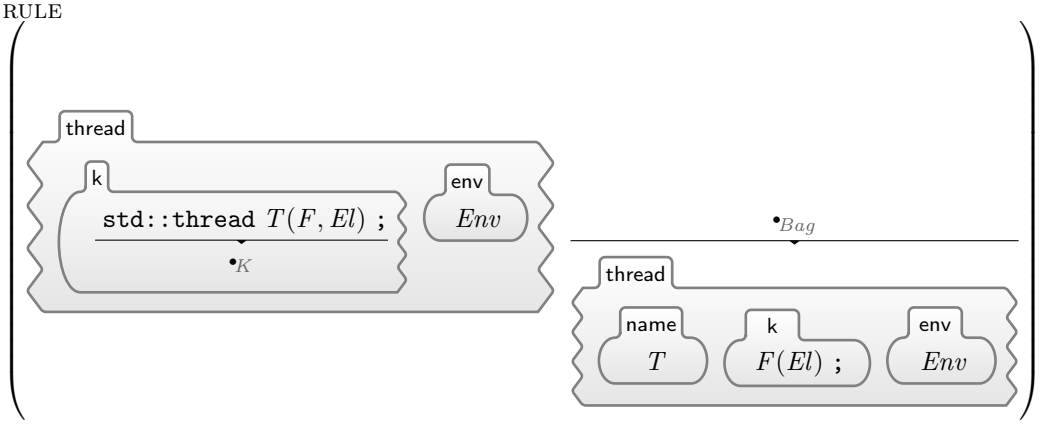
END MODULE

MODULE CINK-THREADS-SEMANTICS

SYNTAX $K ::= \texttt{noName}$

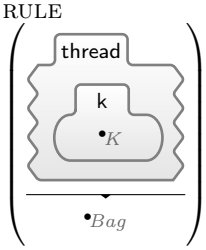
For now, CinK includes a minimal support for threads, namely the creation of a thread and the ending of a thread. The statement for creating a thread specifies the name of the thread T, the name of a function F, and the arguments El of the function. The rule giving semantics to this statement, creates a new cell `thread`, where the computation from the cell `k` of the new thread is the function call expression given as arguments, and the environment of the new thread is current environment of the current thread.

RULE



A thread is finished (and deleted) when the content of its `k` cell is empty:

RULE



END MODULE

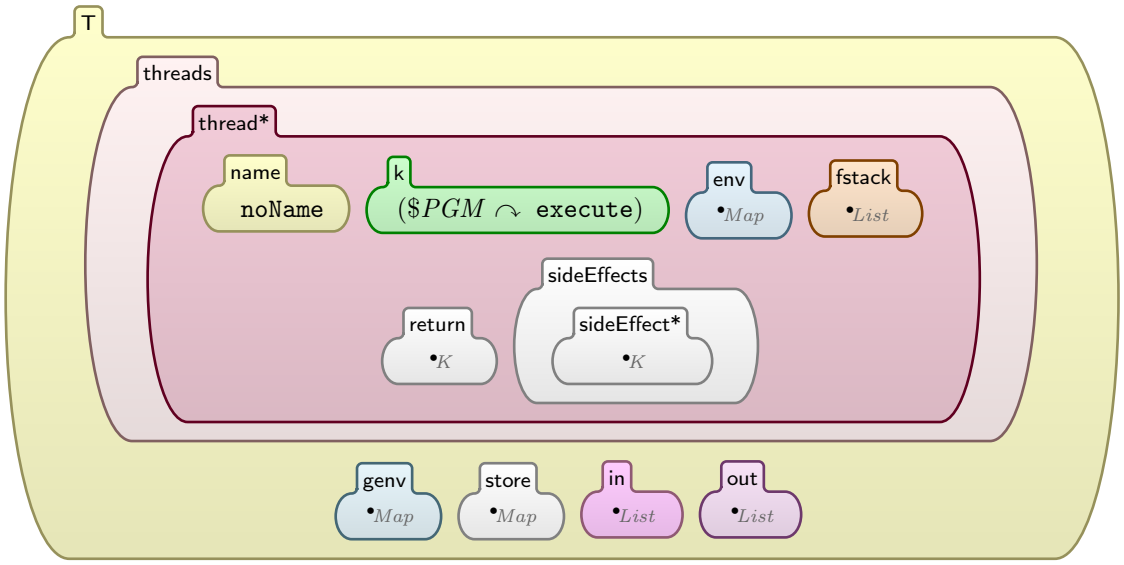
3.2 The Main Module

MODULE CINK

3.2.1 Configuration.

The threads are included in a cell named `threads`, where each cell `thread`, representing the current of a thread, includes a cell `k` for the local computations, a cell `env` for the local environment.

CONFIGURATION:


T
threads
thread*
name
noName
k
($PGM ↷ execute)
env
•Map
fstack
•List
sideEffects
return
•K
sideEffect*
•K
genv
•Map
store
•Map
in
•List
out
•List

END MODULE

Chapter 4

Iteration #3: LTL Model-Checking

This iteration includes an extension of the CinK-threads iteration with a small property language based on LTL formulas, and shows how the $\mathbb{K}$ tool is used together with Maude system for analazing CinK programs.

The definition must be parsed with the command

kompile cink –transition="kripke"

in order to use the model-checker. The model against to the LTL formulas are checked is the transitional system whose transitions are given by the rules annotated with the tag "kripke".

Imported Modules

Two modules are imported: `CINK-BASIC-SYNTAX`, `CINK-BASIC-SEMANTICS`, `CINK-THREADS-SYNTAX`, `CINK-THREADS-SEMANTICS`.

4.1 The New Modules

MODULE CINK-LTLMC-SYNTAX

A input program is a sequence of statements or a LTL Formula (this is needed for parsing purposes)

SYNTAX $Pgm ::= LtlFormula$

We extend the language with labelled statements in order to express with LTL formulas when a statement is reached.

SYNTAX $Stmt ::= Id : Stmt$

This module combines the syntax of the CinK language with that of the LTL formulas. The imported module `MODEL-CHECKER-HOOKS` is a $\mathbb{K}$ interface to the Maude module defining the syntax for the model-checker. In addition to this interface, we have to define the atomic propositions. Here we define a mall set of such propositions. The semantics for these proposition will be given in the next module.

SYNTAX $Prop ::= Id$
| `eqToSum` (Id, Id)
| `eqTo` (Id, Int)
| `logInv` (Id, Id, Id)
| `lt` (Id, Val)
| `leq` (Id, Val)
| `gt` (Id, Val)
| `geq` (Id, Val)
| `neqTo` (Id, Val)

| `eq` (*Id*, *Id*)

SYNTAX *Exp* ::= `step`

END MODULE

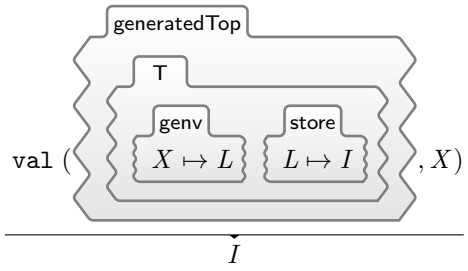
MODULE CINK-LTLMC-SEMANTICS

This module combines the semantics of CinK with the interface to the model-checker, given by the module `LTL-HOOKS`. The states of the transition system to be model-checked are given by the configurations of CinK programs, which are of sort `Bag`.

In order to give semantics to the proposition `eqTo`, we use an auxiliary function `val` that returns the value of a given variable name in a given configuration:

SYNTAX *Int* ::= `val` (*Bag*, *Id*) [function]

RULE



Some additional domain functions:

SYNTAX *Int* ::= `sumFirstN` (*Int*) [function]

RULE

$$\left(\frac{\texttt{sumFirstN}\ (0)}{0}\right)$$

RULE

$$\left(\frac{\texttt{sumFirstN}\ (N)}{\texttt{sumFirstN}\ (N -_{Int} 1) +_{Int} N}\right) \quad \text{when } N >_{Int} 0$$

SYNTAX *Int* ::= `pow` (*Int*, *Int*) [function]

RULE

$$\frac{\texttt{pow}\ (X, 0)}{1}$$

RULE

$$\frac{\texttt{pow}\ (X, Y)}{X *_{Int} \texttt{pow}\ (X, Y -_{Int} 1)} \quad \text{when } Y >_{Int} 0$$

We are ready now to give the semantics for atomic propositions, which is self explained:

RULE

$$\left(\frac{B \models_{Ltl} \texttt{eqTo}\,(X, I)}{\textsf{true}}\right)$$

when $\texttt{val}\,(B, X) =_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{neqTo}\,(X, I)}{\textsf{true}}\right)$$

when $\texttt{val}\,(B, X) \neq_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{lt}\,(X, I)}{\textsf{true}}\right)$$

when $\texttt{val}\,(B, X) <_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{leq}\,(X, I)}{\textsf{true}}\right)$$

when $\texttt{val}\,(B, X) \leq_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{gt}\,(X, I)}{\textsf{true}}\right)$$

when $\texttt{val}\,(B, X) >_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{geq}\,(X, I)}{\textsf{true}}\right)$$

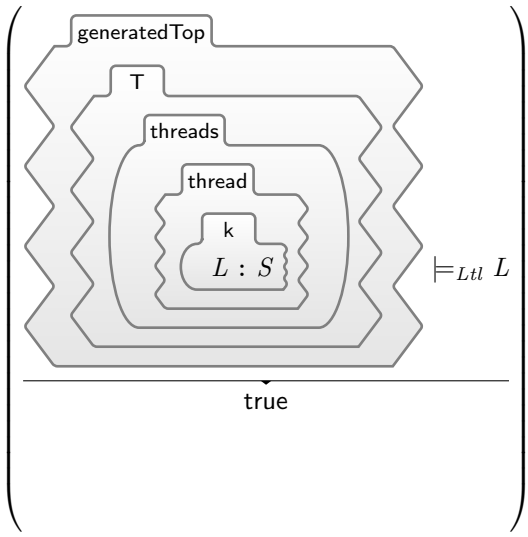
when $\texttt{val}\,(B, X) \geq_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{eq}\,(X, Y)}{\textsf{true}}\right)$$

when $\texttt{val}\,(B, X) ==_{Int} \texttt{val}\,(B, Y)$

RULE



RULE

$$\left(\frac{B \models_{Ltl} \texttt{eqToSum}\ (X, Y)}{\text{true}}\right)$$

when $\texttt{val}\ (B, X) ==_{Int} \texttt{sumFirstN}\ (\texttt{val}\ (B, Y))$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{eqTo}\ (X, I)}{\text{true}}\right)$$

when $\texttt{val}\ (B, X) ==_{Int} I$

RULE

$$\left(\frac{B \models_{Ltl} \texttt{logInv}\ (A, X, K)}{\text{true}}\right)$$

when $(\texttt{val}\ (B, X) *_{Int} \texttt{pow}\ (2, \texttt{val}\ (B, K)) \leq_{Int} \texttt{val}\ (B, A)) \wedge_{Bool} (\texttt{val}\ (B, A) <_{Int} (\texttt{val}\ (B, X) +_{Int} 1) *_{Int} \texttt{pow}\ (2, \texttt{val}\ (B, K)))$

RULE STEP

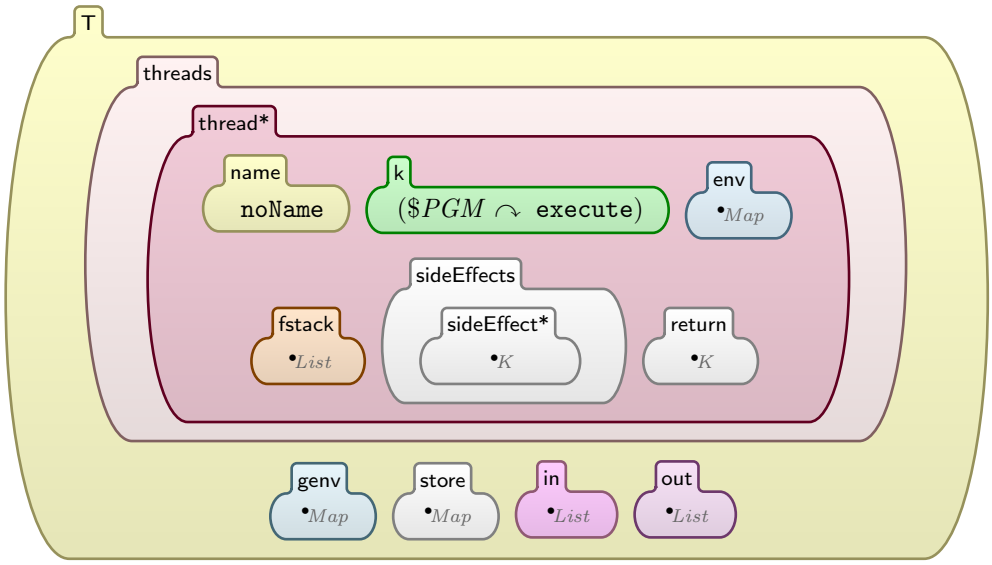
$$\left(\frac{\texttt{step}\ ;}{\bullet_K}\right)$$

END MODULE

4.2 The Main Module

MODULE CINK

CONFIGURATION:


T
threads
thread*
name
noName
k
($PGM ↷ execute)
env
•Map
sideEffects
sideEffect*
•K
fstack
•List
return
•K
genv
•Map
store
•Map
in
•List
out
•List

END MODULE

Chapter 5

Iteration #4: References

This iteration extends the basic iteration with references, declaration of aliases, and the call by reference mechanism.

From C++ manual (8.3.2):

> In a declaration T D where D has either of the forms
> & $attribute - specifier - seq_{opt} D1$
> && $attribute - specifier - seq_{opt} D1$
> and the type of the identifier in the declaration T D1 is derived-declarator-type-list T, then the type of the identifier of D is derived-declarator-type-list reference to T. The optional attribute-specifier-seq appertains to the reference type.. . .
> [Note: A reference can be thought of as a name of an object. –end note]
> A declarator that specifies the type reference to cv void is ill-formed.
> A reference type that is declared using & is called an lvalue reference, and a reference type that is declared using && is called an rvalue reference. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.
>
> It is unspecified whether or not a reference requires storage (3.7). There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an initializer (8.5.3) except when the declaration contains an explicit extern specifier (7.1.1), is a class member (9.2) declaration within a class definition, or is the declaration of a parameter or a return type (8.3.5); see 3.1. A reference shall be initialized to refer to a valid object or function.
> [Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the object obtained by dereferencing a null pointer, which causes undefined behavior. As described in 9.6, a reference cannot be bound directly to a bit-field. –end note]

Imported Modules

Two modules are imported: `CINK-BASIC-SYNTAX` and `CINK-BASIC-SEMANTICS`, including the syntax and the semantics, respectively, of the basic iteration.

5.1 The New Modules

MODULE CINK-REFERENCES-SYNTAX

From the C++ manual (5.3.1):

> The result of the unary & operator is a pointer to its operand. The operand shall be an lvalue or a qualifiedid. If the operand is a qualified-id naming a non-static member m of some class C with type

T, the result has type âĂIJpointer to member of class C of type TâĂİ and is a prvalue designating C::m. Otherwise, if the type of the expression is T, the result has type âĂIJpointer to TâĂİ and is a prvalue that is the address of the designated object (1.7) or a pointer to the designated function.

So, the `strict` is just what we need for the ampersand operator:

SYNTAX *Exp* ::= `&` *Exp* [strict, ampersand]

END MODULE

MODULE CINK-REFERENCES-SEMANTICS

Extension of the getName function:

RULE

$$\frac{\texttt{getName}\ (\ \texttt{\&}\ E)}{\texttt{getName}\ (E)}$$

Reference type:

SYNTAX *RefType* ::= `reference to` *Type*

SYNTAX *Type* ::= *RefType*

Extend the declarations with "reference to type" type:

SYNTAX *Decl* ::= *Exp* `of` *RefType*

SYNTAX *Stmt* ::= *Exp* `of` *RefType* `=` *Exp* `;`

Desugaring the type of a declaration:

RULE

$$\left(\frac{T\ \ \texttt{\&}\ X}{X\ \texttt{of reference to}\ T}\right)$$

Declaration of an alias:

CONTEXT

— `of reference to` — `;`↷ — `=` □ `;`

RULE

k: $\dfrac{X\ \texttt{of reference to}\ \text{—}\ \texttt{;}\curvearrowright X\ \texttt{=}\ \texttt{lval}\ (L)\ \texttt{;}}{\bullet_K}$ env: $\dfrac{Env}{Env[L\ /\ X]}$

The evaluation of the ampersand operator returns the lvalue computed by the heating/cooling rules. Similar to the case of potsfix increment/decrement operators, the evaluation can also be seen as a conversion from lvalue to rvalue since the computed value can be used only as a rvalue.

RULE

$$\frac{\texttt{\&}\ \texttt{lval}\ (L)}{L}$$

5.1.1 Call by Reference

For a formal parameter declared with the call-by-reference mechanism, the corresponding argument expression must be evaluated to an lvalue. From the C++ manual:

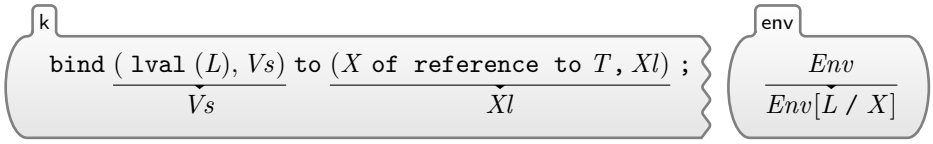
a function can change the values of its non-const parameters, but these changes cannot affect the values of the arguments except where a parameter is of a reference type (8.3.2)...

CONTEXT

`evaluate` (□, —) `to` — `following` (— `of reference to` —, —) `;`

For call-by-reference, the location pointed to by the lvalue is directly bound to the formal parameter. This is achieved by exactly one rule:

RULE

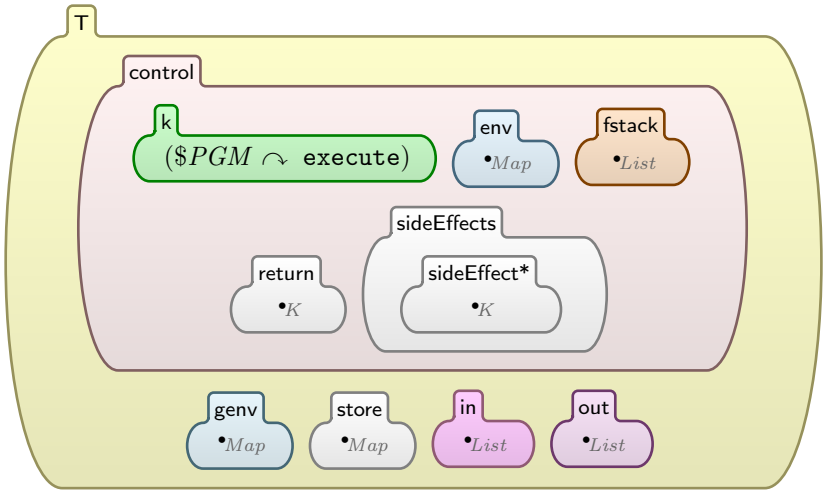


END MODULE

5.2 The Main Module

MODULE CINK

CONFIGURATION:



END MODULE

Chapter 6

Iteration #5: Pointers

This iteration extends the basic iteration with the mechanisms able to handle pointers.

Here is the definition of the indirect operator from the C++ 2011 manual:

> The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is *pointer to T*, the type of the result is *T*.

The Imported Modules

The list of imported modules includes those from the basic iteration, `CINK-BASIC-SYNTAX`,`CINK-BASIC-SEMANTICS`, and that from the references iteration, `CINK-REFERENCES-SYNTAX` and `CINK-REFERENCES-SEMANTICS`.

6.1 The New Modules

MODULE CINK-POINTERS-SYNTAX

The only operand of the indirect operator is an rvalue.

SYNTAX *Exp* ::= `*` *Exp* [strict(all(context(rvalue), result(Val))), indirect]

Extend syntax wit type specifiers needed for the new operator:

SYNTAX *TypeSpec* ::= *FundType*
| *TypeSpec* `*`

The new and delete operators:

SYNTAX *Exp* ::= `new` *TypeSpec* [new]
| `delete` *Exp* [strict, delete]

END MODULE

MODULE CINK-POINTERS-SEMANTICS

Extension of the getName function:

RULE

$$\frac{\texttt{getName}\ (\ \texttt{*}\ E)}{\texttt{getName}\ (E)}$$

Pointer type:

SYNTAX $\mathit{PtrType} ::= \texttt{pointer to}\ \mathit{Type}$

SYNTAX $\mathit{ScalType} ::= \mathit{PtrType}$

Extend the declarations with "pointer to type" type:

SYNTAX $\mathit{Stmt} ::= \mathit{Exp}\ \texttt{of}\ \mathit{PtrType}\ \texttt{;}$

Desugaring the type of a declaration:

RULE

$$\left(\frac{T\ \ \texttt{*}\ X\ \texttt{;}}{X\ \texttt{of pointer to}\ T\ \texttt{;}}\right)$$

RULE

$$\left(\frac{\texttt{*}\ X\ \texttt{of}\ PT\ \texttt{;}}{X\ \texttt{of pointer to}\ PT\ \texttt{;}}\right)$$

Declaration of a pointer variable:

RULE

$$\left(\begin{array}{c}\boxed{\text{k}\ \ \dfrac{X\ \texttt{of}\ T\ \texttt{;}}{\bullet_K}}\ \ \boxed{\text{env}\ \ \dfrac{\mathit{Env}}{\mathit{Env}[L\ /\ X]}}\ \ \boxed{\text{store}\ \ \dfrac{\bullet_{\mathit{Map}}}{(L \mapsto 0)}}\end{array}\right)$$

when $\texttt{fresh}\ (L)$

The indirect operator.
Its semantics is just the inverse of the & operator!!!

RULE

$$\frac{\texttt{*}\ L}{\texttt{lval}\ (L)}$$

Pointers as parameters:

CONTEXT

$$\texttt{evaluate}\ \left(\frac{\square}{\texttt{rvalue}\ (\square)}\ \texttt{,}\ \texttt{—}\right)\ \texttt{to}\ \texttt{—}\ \texttt{following}\ (T\ \ \texttt{*}\ \texttt{—}\ \texttt{,}\ \texttt{—})\ \texttt{;}$$

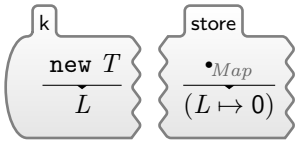
RULE

$$\boxed{\text{k}\ \ \texttt{bind}\ \left(\frac{V}{Vs}\ \texttt{,}\ \frac{Vs}{Xl}\right)\ \texttt{to}\ \left(\frac{T\ \ \texttt{*}\ E}{X}\ \texttt{,}\ \frac{X}{Xl}\right)\ \texttt{;}}\ \ \boxed{\text{env}\ \ \frac{\mathit{Env}}{\mathit{Env}[L\ /\ \texttt{getName}\ (E)]}}\ \ \boxed{\text{store}\ \ \frac{\bullet_{\mathit{Map}}}{L \mapsto V}}$$

when $\texttt{fresh}\ (L)$

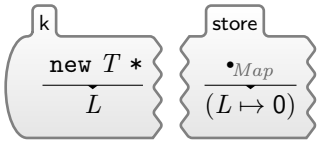
We consider a simplified form of the new operator:

RULE



when `fresh` (L)

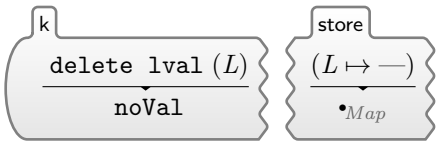
RULE



when `fresh` (L)

The same for delete:

RULE

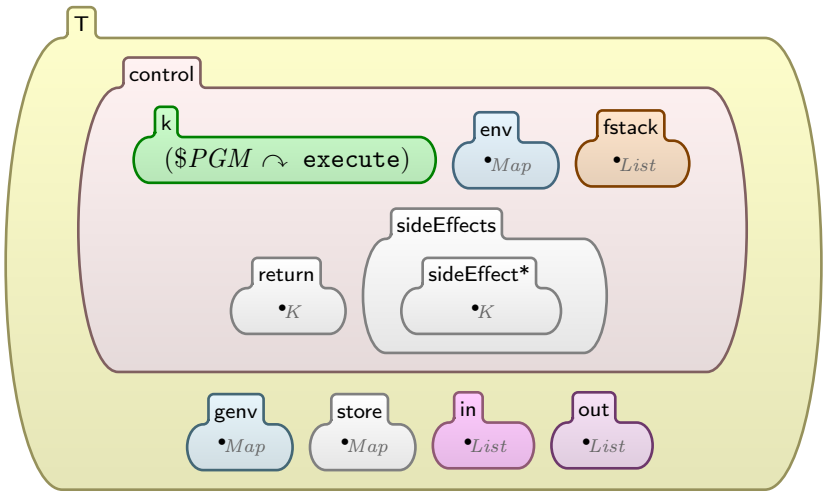


END MODULE

6.2 The Main Module

MODULE CINK

CONFIGURATION:



END MODULE

Chapter 7

Iteration #6: Unidimensional Arrays

This iteration extends the pointers iteration with unidimensional arrays.

From C++ 2011 Manual:

> In a declaration T D where D has the form
> D1 [constant-expressionopt] attribute-specifier-seqopt
> and the type of the identifier in the declaration T D1 is "derived-declarator-type-list T", then the type of the identifier of D is an array type...
> Example:
>
> ```
> typedef int A[5], AA[2][3];
> typedef const A CA; // type is "array of 5 const int"
> typedef const AA CAA; // type is "array of 2 array of 3 const int"
> ```
>
> ...
> An array can be constructed from one of the fundamental types (except void), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array.
> When several "array of" specifications are adjacent, a multidimensional array is created; only the first of the constant expressions that specify the bounds of the arrays may be omitted. In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (8.3.5). An array bound may also be omitted when the declarator is followed by an initializer (8.5). In this case the bound is calculated from the number of initial elements (say, N) supplied (8.5.1), and the type of the identifier of D is "array of N T.

We extend the memory model with contiguous sequences of locations. Such sequences are represented by using the theory of arrays. We assume the reader familiar with this theory. Here is its definition included in `include/builtins/array.k`:

```
syntax Array ::= "store"       "(" Array "," Int "," K ")"
               | "const-array" "(" Int "," K ")"

syntax K ::= "select" "(" Array "," Int ")"
syntax Int ::= "size-of-array" "(" Array ")"

rule select(store(_:Array, I:Int, V:K), J:Int)  => V
     requires I ==Int J

rule select(store(A:Array, I:Int, _:K), J:Int)  => select(A, J)
     requires I =/=Int J

rule select(const-array(Size:Int, V:K), I:Int)  => V
     requires (0 <=Int I) andBool (I <Int Size)
```

We explain the main idea by means of an example. A sequence of three locations storing the sequence of integers (5,3, 8) is represented by the map:

```
seq-array(3, L)
|->
```

```
store(store(store(const-array(3, 0), 0, 5), 1, 3), 2, 8)
```

where `seq-array(3, L)` is a sugar syntax for
`store(store(store(const-array(5, 0), 0, L[0]), 1, L[1]), 2, L[2])`.

The arrays can be converted to pointers. From the C++ manual:

> An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be converted to a prvalue of type "pointer to T". The result is a pointer to the first element of the array.

Therefore the pointer arithmetic is extended over array locations:
$L[I] + V = \texttt{select}(\texttt{seq-array}(N, L), I + V)$,
$++L[I] = \texttt{select}(\texttt{seq-array}(N, L), I + 1)$,
$--L[I] = \texttt{select}(\texttt{seq-array}(N, L), I - 1)$.
The lvalue of $L[I]$ is $\texttt{select}(\texttt{seq-array}(N, L), I)$, and the rvalue of $L[I]$ is $\texttt{select}(A, I)$,
where $\texttt{seq-array}(N, L), I)$ `|->` A`:Array`.
Since we did not considered yet the casting, we cannot write $A + I$ but we can write $\&A[0] + I$. The casting will be defined in a future iteration.

7.1 Imported Modules

This iteration is built over that of pointers. Therefore the list of imported modules includes the list from pointers's iteration and the two modules defined by it: `CINK-POINTERS-SYNTAX` and `CINK-POINTERS-SEMANTICS`.

7.2 New Modules

MODULE CINK-ARRAYS-SYNTAX

SYNTAX *Exp* ::= *Exp*[*Exp*] [strict(1, 2(context(rvalue), result(Int))), array]
| *Exp* `[]` [array]

END MODULE

MODULE CINK-ARRAYS-SEMANTICS

The memory model:

SYNTAX *SeqLoc* ::= `seq-array` (*Int*, *Loc*)

SYNTAX *Loc* ::= *SeqLoc*

Extending select over sequences of locations:

SYNTAX *Loc* ::= `select` (*Loc*, *Int*) [smtlib(select)]

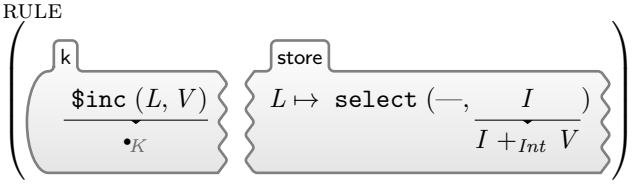
RULE

$$\frac{\texttt{lval}\ (L)[I]}{\texttt{lval}\ (\ \texttt{select}\ (L, I))}$$

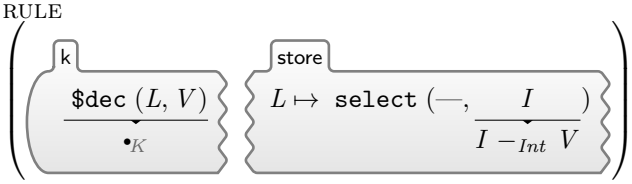
RULE

$$\frac{\texttt{select}\ (L, I)\ \texttt{+}\ J}{\texttt{select}\ (L, I +_{Int} J)}$$

RULE



RULE



We define array types following the notation convention from the manual. We call these types *explicit array types*. Note that these have a recursive definition. We also extend statements with declaration of variables (expressions) of the new type.

SYNTAX $ArrType ::= \texttt{array of}\ Int\ Type$

SYNTAX $Type ::= ArrType$

SYNTAX $Stmt ::= Exp\ \texttt{of}\ ArrType\ \texttt{;}$

An updated array is a value:

RULE

$$\frac{isVal(\ \texttt{store}\ (—,—,—))}{\texttt{true}}$$

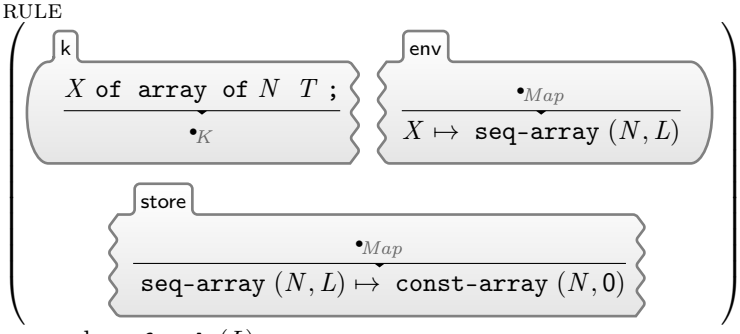
Array declarations are translated into declarations with explicit array types:

RULE

$$\frac{T\ \ X[N]\ \texttt{;}}{X\ \texttt{of array of}\ N\ \ T\ \texttt{;}}$$

Semantics of an array declaration:

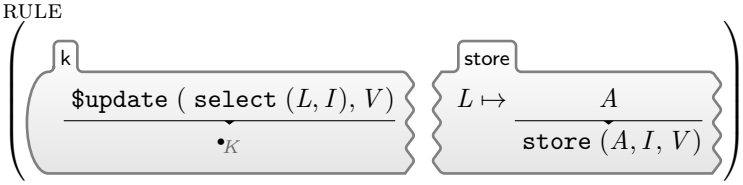
RULE



when $\texttt{fresh}\ (L)$

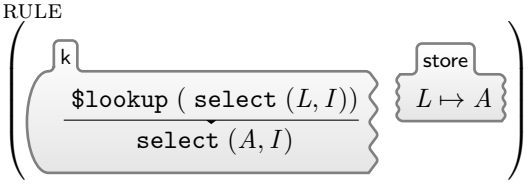
Updating an array component:

RULE



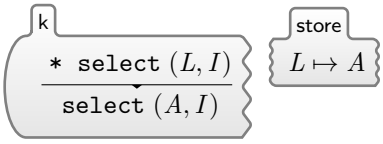
Reading an array component:

RULE



Indirection to an array component:

RULE



Arrays as parameters:

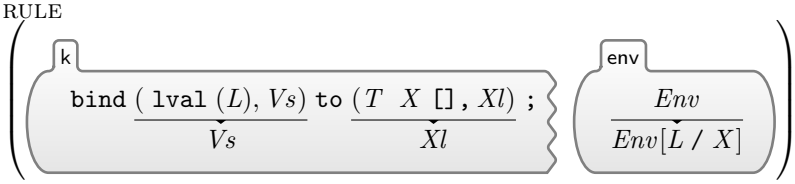
CONTEXT

`evaluate` (□, —) `to` — `following` (T —[—], —) `;`

CONTEXT

`evaluate` (□, —) `to` — `following` (T — `[]` , —) `;`

RULE



Interface with SMTLIB:

RULE

$$\frac{\texttt{K2SMTLib}\ (\ \texttt{seq-array}\ (N, L))}{\texttt{"const-array ("} +_{String}\ \texttt{K2SMTLib}\ (N) +_{String}\ \texttt{","} +_{String}\ \texttt{K2SMTLib}\ (N) +_{String}\ \texttt{")"}}$$

END MODULE

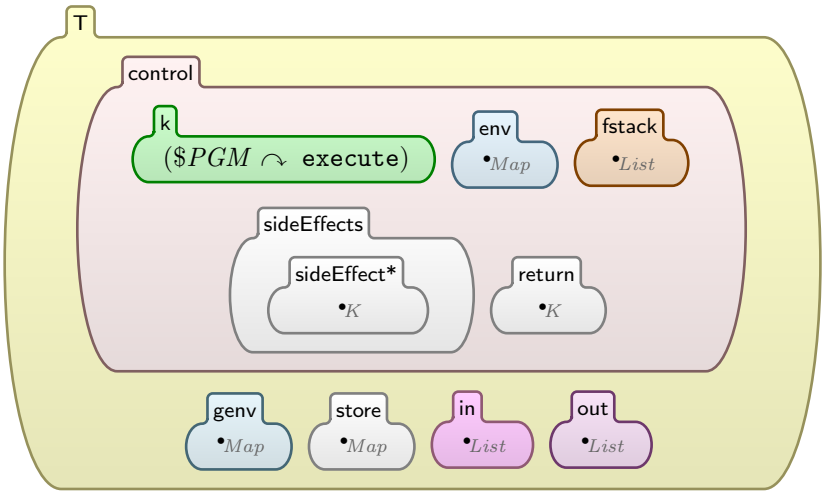
7.3 Main Module

MODULE CINK-SYNTAX

END MODULE

MODULE CINK

CONFIGURATION:


T
control
k
($PGM ↷ execute)
env
•Map
fstack
•List
sideEffects
sideEffect*
•K
return
•K
genv
•Map
store
•Map
in
•List
out
•List

END MODULE

Chapter 8

Iteration #7: Arrays

This iteration includes the extension of the pointer iteration to multidimensional arrays. From C++ 2011 Manual (8.3.4):

> In a declaration T D where D has the form
> D1 [constant-expressionopt] attribute-specifier-seqopt
> and the type of the identifier in the declaration T D1 is "derived-declarator-type-list T", then the type of the identifier of D is an array type...
> Example:
>
> ```
> typedef int A[5], AA[2][3];
> typedef const A CA; // type is "array of 5 const int"
> typedef const AA CAA; // type is "array of 2 array of 3 const int"
> ```
>
> ... "An array can be constructed from one of the fundamental types (except void), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array.
> When several "array of" specifications are adjacent, a multidimensional array is created; only the first of the constant expressions that specify the bounds of the arrays may be omitted. In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (8.3.5). An array bound may also be omitted when the declarator is followed by an initializer (8.5). In this case the bound is calculated from the number of initial elements (say, N) supplied (8.5.1), and the type of the identifier of D is "array of N T.

The memory model defined for unidimensional arrays can be extended to multidimensional arrays. A 2×3 array $\{\{2, 4, 6\}, \{3, 5, 7\}\}$ is represented as:

```
seq-array(2, L) |-> array of 2 array of 3 int

select(seq-array(2, L), 0)
|->
store(store(store(const-array(3, 0), 0, 2), 1, 4), 2, 6)

select(seq-array(2, L), 1)
|->
store(store(store(const-array(3, 0), 0, 3), 1, 5), 2, 7)
```

where `array of 3 int` is replaced first by `const-array(3, 0)` and then component-wise updated.

The main idea is as follows. We assume that the array type is `array of` N_k `array of` $N_{k-1} \ldots$. The whole array is stored in a contiguous sequence of cells starting at a distinguished address L, denoted by `seq-array`(N_k, L), where N_k is the last dimension of the array. The value stored at L is the type of the array. Since the value N_k can be "unknown" (e.g., represented by a symbolic value), we do not want to explicitly represent the array value stored at L. However, the information about the array is needed for defining the semantics of the language. Therefore the "value" stored at L is the array type, meaning that at L is stored a value of this type. The operator `select` from the theory of arrays is extended to sequences of locations. In fact, `seq-array`(N, L) can thaught as a sugar syntax for the array

$$\texttt{store}(\ldots\texttt{store}(\texttt{const-array}(N, \bot), L[0]), \ldots, L[N-1]).$$

The location of the I-th component is $\texttt{select}(\texttt{seq-array}(N, L),\ I)$. A component is represented in the cell `store` only if it is accesed. If the type of the component is a multidimensional array, then it is represented in a similar way to that of the array:

$$\texttt{select}(\texttt{seq-array}(N_k, L),\ I) \mapsto \texttt{array of}\ \ N_{k_1}\ \ \ldots$$

If the component is an array of scalars, then it is stored in the same way to that of unidimensional arrays iteration.

If all components of an array are represented in the cell `store`, then its complete value can be built in a bottom-up manner.

The pointer arithmetic can be extended with
$L[0][0] + N * I + J = \texttt{select}(\texttt{select}(\texttt{seq-array}(N, L), I), J)$ if $L[0] \mapsto \texttt{array of}\ \ N \ldots$.

8.1 Imported Modules

This iteration is built over that of pointers. Therefore the list of imported modules includes the list from pointers's iteration and the two modules defined by it: `CINK-POINTERS-SYNTAX` and `CINK-POINTERS-SEMANTICS`.

8.2 New Modules

MODULE CINK-ARRAYS-SYNTAX

The syntax of the language is extended with the syntax of arrays:

SYNTAX *Exp* ::= *Exp*[*Exp*] [strict(1, 2(context(rvalue), result(Int))), array]
| *Exp* `[]` [array]

Extend TypeSpec with arrays

SYNTAX *TypeSpec* ::= *TypeSpec*[*Exp*]

END MODULE

MODULE CINK-ARRAYS-SEMANTICS

The semantics consists of the semantics of basic constructs together with the that of arrays and the definition of the whole configuration:

The definition of the memory model for arrays:

SYNTAX *SeqLoc* ::= `seq-array` (*Int*, *Loc*)

SYNTAX *Loc* ::= *SeqLoc*

Extend select over sequences of locations

SYNTAX *Loc* ::= `select` (*Loc*, *Int*) [smtlib(select)]

RULE

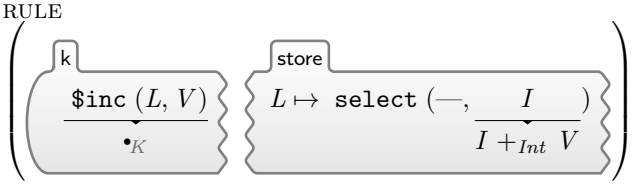
$$\frac{\texttt{lval}\ (L)[I]}{\texttt{lval}\ (\ \texttt{select}\ (L, I))}$$

Since $L[I]$, where L is the location of the array A, can be seen as a pointer to the I-th component, we have to define a pointer arithmetic over the addresses of the components. For instance, we have $L[I] + J = L[I + J]$ in this arithmetic. Recall that $L[I]$ is represented here by the select operator.

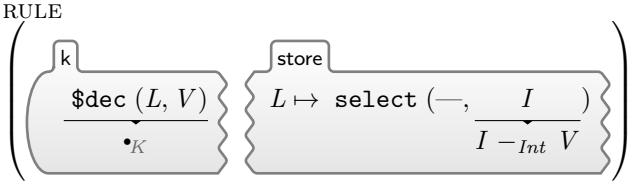
RULE

$$\frac{\texttt{select}\ (L, I)\ \texttt{+}\ J}{\texttt{select}\ (L, I +_{Int} J)}$$

RULE



RULE



Since an array is stored into contiguous sequence of locations, we have a partial order over the locations of the components:

RULE

$$\frac{\texttt{select}\ (L, I) \leq\ \texttt{select}\ (L, J)}{I \leq_{Int} J}$$

RULE

$$\frac{\texttt{select}\ (L, I)\ \texttt{<}\ \texttt{select}\ (L, J)}{I <_{Int} J}$$

RULE

$$\frac{\texttt{select}\ (L, I)\ \texttt{>}\ \texttt{select}\ (L, J)}{I >_{Int} J}$$

We define array types following the the notation convention from the manual. We call these type *explicit array types*. Note that these have a recursive definition. We also extends statements with declaration of variable (expression) of the new type.

SYNTAX $\mathit{ArrType} ::= \texttt{array of}\ \mathit{Int}\ \ \mathit{Type}$

SYNTAX $\mathit{Type} ::= \mathit{ArrType}$

SYNTAX $\mathit{Stmt} ::= \mathit{Exp}\ \texttt{of}\ \mathit{ArrType}\ \texttt{;}$

The array declaration are translated into declaration with explicit array types.

CONTEXT

$$\frac{\texttt{rvalue}\ (\square)}{\texttt{—}\ \texttt{—[}\ \square\ \texttt{]}\ \texttt{;}}$$

result(Int)

RULE

$$\frac{T\ \ X[N]\ \texttt{;}}{X\ \texttt{of array of}\ N\ \ T\ \texttt{;}}$$

RULE

$$\frac{X[N]\ \texttt{of}\ AT\ \texttt{;}}{X\ \texttt{of array of}\ N\ \ AT\ \texttt{;}}$$

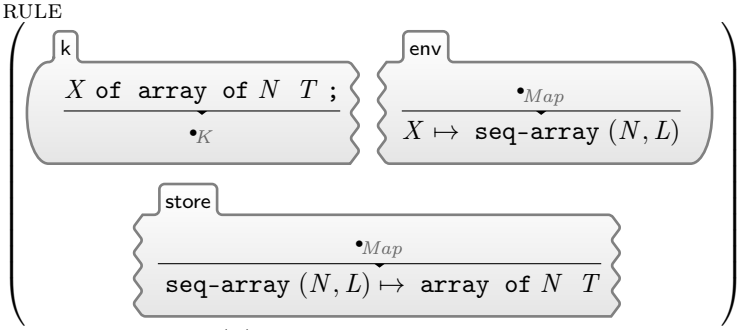
An updated array is a value:

RULE

$$\frac{\mathit{isVal}(\ \texttt{store}\ (\texttt{—}, \texttt{—}, \texttt{—}))}{\texttt{true}}$$

Declaration of a multidimensional array:

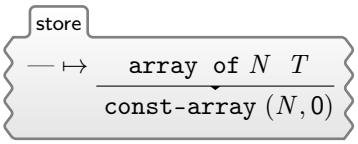
RULE



when fresh (L)

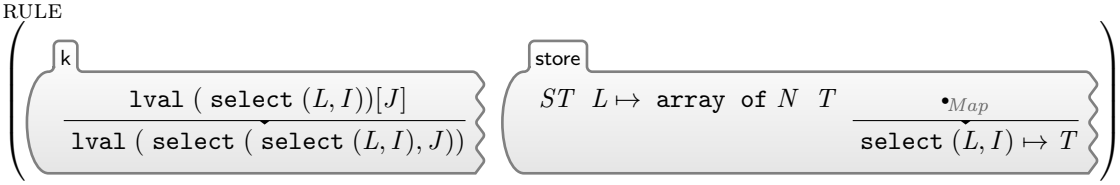
Arrays of scalar type are zero-initialized, so they are well defined:

RULE



The case of a component of a multimensional array, the first access:

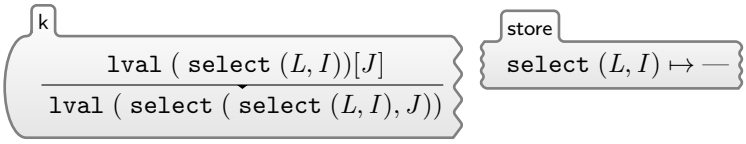
RULE



when $\neg_{Bool}$ select (L, I) in keys (ST)

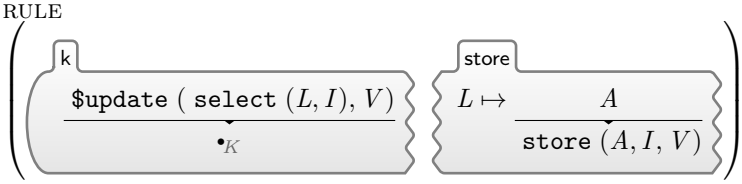
. . . and the case when the component is already allocated:

RULE



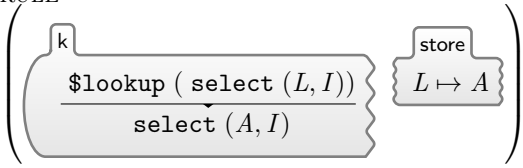
Updating an array component:

RULE



Reading an array component:

RULE



Arrays as parameters:

CONTEXT

`evaluate` $(\square, \_)$ `to` $\_$ `following` $(T\ \ \_\ \ [\,]\,, \_)$ `;`

result(Val)

CONTEXT

`evaluate` $(\square, \_)$ `to` $\_$ `following` $(T\ \ \_[\_]\,, \_)$ `;`

RULE

$$\left(\begin{array}{l} \texttt{k}: \texttt{bind}\ (\ \texttt{lval}\ (L), Vs)\ \texttt{to}\ (T\ \ X\ \ [\,]\,, Xl)\ \texttt{;} \\ \qquad\qquad\qquad\qquad\quad \downarrow Vs \qquad\qquad\quad \downarrow Xl \end{array} \quad \begin{array}{l} \texttt{env}: Env \\ \qquad\quad \downarrow Env[L\ /\ X] \end{array}\right)$$

RULE

$$\left(\frac{\texttt{bind}\ (\ \texttt{lval}\ (L), Vs)\ \texttt{to}\ (T\ \ X[\_]\,, Xl)\ \texttt{;}}{\texttt{bind}\ (\ \texttt{lval}\ (L), Vs)\ \texttt{to}\ (T\ \ X\,, Xl)\ \texttt{;}}\right)$$

RULE

$$\left(\frac{\texttt{bind}\ (\ \texttt{lval}\ (L), Vs)\ \texttt{to}\ (T\ \ X\ \ [\,]\ \ [\,]\,, Xl)\ \texttt{;}}{\texttt{bind}\ (\ \texttt{lval}\ (L), Vs)\ \texttt{to}\ (T\ \ X\ \ [\,]\,, Xl)\ \texttt{;}}\right)$$

Interface with SMTLIB:

RULE

$$\frac{\texttt{K2SMTLib}\ (\ \texttt{seq-array}\ (N, L))}{\texttt{"const-array ("} +_{String}\ \texttt{K2SMTLib}\ (N) +_{String}\ \texttt{","} +_{String}\ \texttt{K2SMTLib}\ (N) +_{String}\ \texttt{")"}}$$

END MODULE

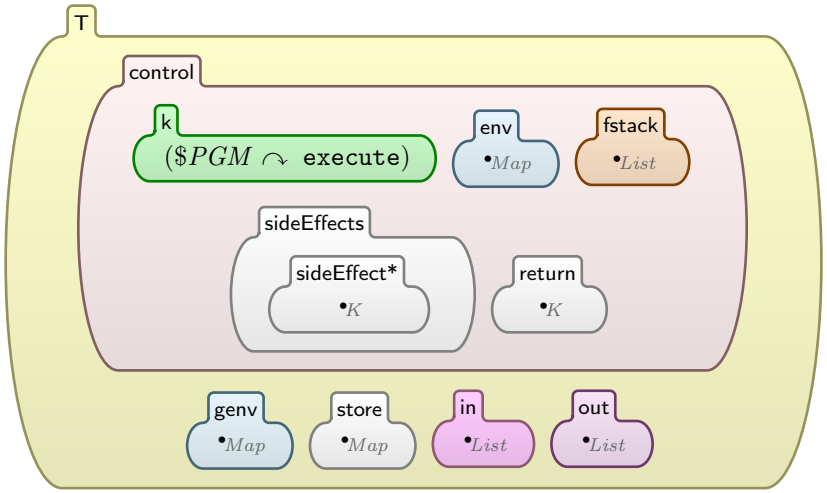
MODULE CINK-SYNTAX

END MODULE

8.3 Main Module

MODULE CINK

CONFIGURATION:



END MODULE