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

Dorel Lucanu, Traian Florin Şerbănuţă

TR 12-03, June 2012 (Version 1)
CinK – an exercise on how to think in K

Dorel Lucanu and Traian Florin Șerbănuță

Alexandru Ioan Cuza University of Iași, Romania
Chapter 1

Introduction

CinK is a kernel of the C++ language we used to experiment with K. The language is used an example for teaching classes and is referred in several research papers.

In this report we briefly present the K definition of CinK. The most part of the text is automatically generated with the 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 K. We assume the reader is already familiar with the K Framework and the K tool and here we try to share our experience in defining a languages with some specific features, as C++ is. Such features includes a clear distinction between l-values and r-values, declaration of aliases, and various parameters passing mechanisms. We also extend the definition with a small property language, including LTL formulas, and we show how the K tool is used together with Maude system for analyzing CinK programs.

The K definition of CinK is continuously evolving, so this report presents the status of this definition at the publication date.
Chapter 2

Iteration #1: Basic Constructs

This iteration defines the basic constructs of the language. The definition can be tested with the online tool: http://fmse.info.uaic.ro/tools/K/?tree=examples/cink/basic/cink.k

Imported Modules

Two modules are imported: CINK-BASIC-SYNTAX and CINK-BASIC-SEMANTICS, including the syntax and the semantics, respectively, of the basic constructions. These modules will be imported in the extensions of the language, as well.

2.1 Syntax

The syntax of CinK is written using the latest version at the moment of the K tool. Since the language is relatively small, we use the facilities the K tool offers for defining the syntax. This is given using a BNF-like notation enriched with annotations and priority specifications. In this way, the rules giving semantics are written using the CinK syntax.

MODULE CINK-BASIC-SYNTAX

2.1.1 Declarations

The declarations are used for declaring variables and function names together with their return types. We consider only integer and boolean variables. A return type of a function could be an integer, a boolean, or void.

SYNTAX PrimType ::= int
  | bool
  | void

SYNTAX Type ::= PrimType

SYNTAXDecl ::= Type Exp

2.1.2 Values

The constants of the builtin types are primitive constructs and therefore they should be values. We prefer to declare cin and cout as values rather than identifiers; the reason for this decision will be explained later.
2.1.3 Expressions

We included in CinK a small subset of C++ language, the missing operators can be easily added in a similar way. Recall that the most important (from the semantic point of view) thing we have to mention for operators is the attribute \texttt{strict}, that specifies the evaluation order of the operands. For instance, the arithmetic binary operators are strict in both arguments, hence the behavior of some programs could be undefined or non-deterministic because the evaluation of the arguments could have side-effects. The assignment operator is strict only in the second argument because the first argument must be evaluated to an l-value. These features are common to many languages. 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 it is depending on the binding mechanism of the corresponding argument: this can be call-by-value or call-by-reference. These two mechanisms will be explained later.

\begin{verbatim}
SYNTAX Exp ::= Id
             | Val
             | (Exp)[bracket]
             | Exp(Exp)[strict(1)]
             | Exp * Exp [strict, multiply]
             | Exp / Exp [strict, divide]
             | Exp\%Exp [strict, modulo]
             | Exp + Exp [strict, plus]
             | Exp - Exp [strict, minus]
             | Exp < Exp [strict, lessthan]
             | Exp > Exp [strict, greatthan]
             | Exp \leq Exp [strict, lessequal]
             | Exp == Exp [strict, equality]
             | ! Exp [strict, negation]
             | Exp \&\& Exp [strict(1), conjunction]
             | Exp || Exp [strict(1), disjunction]
             | Exp \ll Exp [seqstrict, write]
             | Exp \gg Exp [read]
             | Exp = Exp [strict(2), assign]
             | endl
\end{verbatim}

2.1.4 Statements

For now, we include in CinK only (a subset) of the imperative statements: expression statement, bloc, sequential composition, while, and conditionals. We also added a minimal support for threads, similar to IMPPP, but using a C++ syntax and semantics.

\begin{verbatim}
SYNTAX Stmt ::= Exp ; [strict]
             | #include <iostream>
             | using namespace std;
             | Decl ; [klabel(declStmt)]
             | {}.
\end{verbatim}
while (Exp) Stmt
return Exps ; [strict]
Decl(Decls) { Stmts }
if (Exp) Stmt else Stmt [strict(1)]
if (Exp) Stmt

A program is a sequence of statements:

SYNTAX  Pgm ::= Stmts
SYNTAX  Stmts ::= Stmt
              | Stmts Stmts

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

SYNTAX  Exps ::= List{Exp," ","} [strict]
SYNTAX  Decls ::= List{Decl," ","} [strict]

END MODULE

2.2 Semantics

MODULE CINK-BASIC-SEMANTICS

2.2.1 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, e.g., for the heating
and cooling rules. We already defined the subset of values that is part of the syntax, namely the values given by
the primitive types. Here we extend the set of 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 K examples (IMP, IMPPP, SIMPLE), the functions are stored similar to
variables and therefore their definitions are seen as values. As usual, all values must be subsorted to the KResult
sort.

SYNTAX  Val ::= noVal
          | \ Decls • Stmts
SYNTAX  Vals ::= List{Val," ","}
SYNTAX  KResult ::= Val
SYNTAX  Exp ::= undefined

2.2.2 Lvalues and Rvalues
In C++ the values are split in several categories:
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 of these categories: lvalues (LVal) and rvalues (RVal). The sort Val for values will be extended in the module describing the semantics.

We first deal only with lvalues and prvalues. In order to keep the definition as simple as possible, we let the the prvalue as the default category for expressions and we make explicitly only the category of lvalue expressions:

**SYNTAX**  \[ K ::= \text{lvalue} (K) \]

As its definition says, the main operator emphasizing the two categories is the assignment. We use a context declaration to say that the left-hand side of the assignment is an lvalue:

**CONTEXT**

\[
\square = \text{lvalue} (\square)
\]

The above declaration is equivalent with the following to heating/cooling rules:

\[
E_1 = E_2 \Rightarrow \text{lvalue}(E_1) \xrightarrow{\text{HOLE}} E_2 \quad (2.1)
\]

\[
\text{lvalue}(L_1) \xrightarrow{\text{HOLE}} E_2 \Rightarrow L_1 = E_2 \quad (2.2)
\]

This assume that the locations must be defined as values:

### 2.2.3 Memory Locations

We use (automatically generated) symbolic values for locations. These are of sort Loc. The locations are by definition lvalues.

**SYNTAX**  \[ Loc ::= \text{noLoc} \]

**SYNTAX**  \[ LVal ::= Loc \]

### 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**  \[ K ::= \text{execute} \]

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

**SYNTAX**  \[ ListItem ::= (\text{List}, K) \]

\[ \text{List} \]

\[ \text{Map} \]
2.2.5 Configuration.

Since CinK is developed in iterations, each such iteration will have its own configuration. In order to have a modular definition, each module adding semantics for several feature will include a minimal configuration needed to define these features. The following cells are used to give semantics for the basic constructs:

\[
\begin{align*}
\text{configuration:} & \\
& \begin{cases}
  (P_{GM} \rightarrow \text{execute}) \\
  \text{Map env} \\
  \text{List fstack} \\
  \text{Map genv} \\
  \text{Map store} \\
  \text{List in} \\
  \text{List out}
\end{cases}
\end{align*}
\]

In order to have a minimal set of rules, some syntactic constructs are desugared. The desugaring can be done using the structural rules.

The desugaring rule for the if-then statement:

**Rule**
\[
\text{if} \ (B) \ St \\
\text{if} \ (B) \ St \ 	ext{else} \ \{\}
\]

The desugaring rule for statements declaring multiple variables.

**Rule**
\[
T \ E, E', Es; \\
T \ E; T \ E'; T \ Es;
\]

Desugaring rule for variable declarations with initialization. The alias declarations are defined later, together with the pointers.

**Rule**
\[
T \ E1 = E2; \\
T \ E1; E1 = E2; \\
\text{when} \ \text{isAliasExp} \ (E1) \neq K \ true()
\]

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**
\[
\begin{align*}
& (\text{Decl} (XI) \{Sts\}) \\
& \text{Map env} \\
& \text{Map store}
\end{align*}
\]

when fresh \ (L)

[fun-decl, structural]
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.

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.

The following expressions are strict.
RULE
\[ \frac{I_1 / I_2}{I_1 \div \text{Int} \ I_2} \]
when \( I_2 /= \text{Int} \ 0() \)
[structural]

RULE
\[ \frac{I_1 \% I_2}{I_1 \% \text{Int} \ I_2} \]
when \( I_2 /= \text{Int} \ 0() \)
[structural]

RULE
\[ \frac{I_1 < I_2}{(I_1 < \text{Int} \ I_2)} \]
[structural]

RULE
\[ \frac{I_1 > I_2}{(I_1 > \text{Int} \ I_2)} \]
[structural]

RULE
\[ \frac{I_1 \leq I_2}{(I_1 \leq \text{Int} \ I_2)} \]
[structural]

RULE
\[ \frac{I_1 == I_2}{(I_1 == \text{Int} \ I_2)} \]
[structural]

Here are the rules for the expressions having the strict attribute declared in the syntax.

RULE
\[ \frac{\text{true()} \ &\& \ B}{B} \]
[structural]

RULE
\[ \frac{\text{false()} \ &\& \ B}{\text{false()}} \]
[structural]

RULE
\[ \frac{\text{true()} \ |\ | \ B}{\text{true()}} \]
[structural]

RULE
\[ \frac{\text{false()} \ |\ | \ B}{B} \]
[structural]
2.2.8 Memory Operations.

The evaluation of a variable name as an r-value:

\[
\text{rule}\!
\frac{! \text{false}()}{! \text{true}()}\!
\text{[structural]}
\]

2.2.8 Memory Operations.

The evaluation of a variable name as an r-value:

\[
\text{rule}\!
\frac{! \text{true}()}{! \text{false}()}\!
\text{[structural]}
\]

2.2.8 Memory Operations.

The evaluation of a variable name as an r-value:

\[
\text{rule}\!
\frac{\text{endl}}{\text{\"\n\"()}}\!
\text{[structural]}
\]

2.2.8 Memory Operations.

The evaluation of a variable name as an r-value:

\[
\text{rule}\!
\frac{X \rightarrow L}{L \rightarrow V}\!
\text{[kripke(mem-lookup)]}
\]

An identifier seen as a lvalue is evaluated to its location:

\[
\text{rule}\!
\frac{\text{lvalue}(X)}{X \rightarrow L}\!
\text{[lookup]}
\]

The memory update is given by the assignment operator. An assignment, after the reduction of the arguments, has in the left-hand side a location (= the lvalue designated by the lhs.)

\[
\text{rule}\!
\frac{L = V}{L \rightarrow V}\!
\text{[update]}
\]

/*

2.2.9 Control Statements.

As usual, the while statement is desugared using the if-then-else statement.
Since if is strict in the first argument, which is a boolean expression, proceed by case-analysis on the result values:

RULE
\[
\begin{align*}
\text{if} (\text{false}()) &\rightarrow \text{else } \text{St} \\
\text{St} &
\end{align*}
\]
[if-false]

RULE
\[
\begin{align*}
\text{if} (\text{true}()) \text{St} \text{else} &\rightarrow \\
\text{St} &
\end{align*}
\]
[if-true]

The semantics of the expression statement consists of removing the value obtained by evaluating the expression. Recall that the statement is strict.

RULE
\[
\begin{align*}
(V ; \leftarrow K) &\rightarrow K \\
\text{when } K &\neq K \cdot K
\end{align*}
\]
[instr-expr]

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
\[
\begin{align*}
\{\text{Sts}\} &\rightarrow \text{Sts} \\
\text{Sts} &
\end{align*}
\]
[block, structural]

– the case of the empty block:

RULE
\[
\begin{align*}
\{} &\rightarrow *K \\
*K &
\end{align*}
\]
[block-empty, structural]

The sequential composition is just a sequence of computations:

RULE
\[
\begin{align*}
(\text{Sts} \text{ Sts}') &\rightarrow (\text{Sts} \leftarrow \text{Sts}') \\
(\text{Sts} \leftarrow \text{Sts}') &
\end{align*}
\]
[seq-comp, structural]

2.2.10 Input/Output Statements.

Writing in the the standard stream \texttt{cout}:
In order to read from the standard stream `cin`, the expression from the right-hand side must be evaluated to an l-value:

\[
\text{context} \\
\text{cin} \gg \square \\
\text{rule} \\
\text{lvalue}(\square)
\]

**2.2.11 The function call expression**

The function name is evaluated to its value, which is a lambda abstraction: \( \lambda Xl \) 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 in a future 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 ] ...

When a function is called, the parameters that have object type shall have completely-defined object type. ...

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 the initialization 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)... ]

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 mimicking the heating-cooling mechanism (the first rule is a heating-like one, and the second a cooling-like one).
To evaluate actual parameters according to their declared strategy we will make use of the power of $K$ evaluation contexts. The actual parameters must be evaluated using the `evaluate` construct and following the list of formal parameters.

**Syntax**

```plaintext
Exps ::= evaluate Exps to Vals following Decls;
```

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

**Context**

```plaintext
evaluate (□, —) to — following (T X, —);
```

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

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

**Rule**

```plaintext
evaluate V, El to Vl following Dec, Xl;
```

```plaintext
\[
\text{append}(Vl, V) \quad \text{append}(Vl, V)
\]
```

**[Structural]**

**Rule**

```plaintext
evaluate *Exps to Vl following *Decls;
```

**[Structural]**

### 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**

```plaintext
K ::= bind Vals 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**

```plaintext
bind (V, Vs) to (T X, XL);
```

```plaintext
\[
\text{store}(\text{Map}(L \mapsto V))
\]
```

**[Structural]**

when $\text{fresh}(L)$
Finally, once all parameters have been bound, the binding construct dissolves:

RULE
bind \cdot_{\text{vals}} \text{ to } \cdot_{\text{decls}} ;
\quad \star_{K}
[\text{structural}]

2.2.12 Function Return.

Similar to C++, the argument of a return statement is a list of expressions (note that Cink does not have yet a distinguished definition for the comma operator).

Similar to the comma operator, the evaluation of the return argument is given from left to right. Since the expression lists are strict, we get this feature for free provided the declaration for return is strict. Remains only to write rules keeping the last value from the list.

RULE
return \_, V, Vs ;
\begin{align*}
\text{return } V, Vs ;
\end{align*}

2.2.13 Auxiliary Functions and Rules.

Here we include the semantics for the auxiliary constructs, used to give semantics for CinK. The "alias" are used for declaring aliases for the variables; their semantics is given in the iteration that adds the pointers.

SYNTAX
\begin{align*}
\text{Bool} & := \, \text{isAliasExp} \, (\text{Exp}) \ \text{[function]}
\end{align*}

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

SYNTAX
\begin{align*}
\text{Id} & := \, \text{getName} \, (K) \ \text{[function]}
\end{align*}

RULE
\begin{align*}
\text{getName} (X)
\end{align*}

RULE
\begin{align*}
\text{getName} (T \ E)
\quad \text{getName} (E)
\end{align*}

RULE
\begin{align*}
\text{getName} (E1 = E2)
\quad \text{getName} (E1)
\end{align*}

Append function:

SYNTAX
\begin{align*}
\text{Vals} & := \, \text{append} \, (\text{Vals}, \text{Val}) \ \text{[function]}
\end{align*}
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. The definition can be tested with the online tool:

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(Exps) ;

END MODULE

MODULE CINK-THREADS-SEMANTICS

SYNTAX K ::= noName

CONFIGURATION:

3.1.1 Threads.

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.

From 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. End note]

Every thread in a program can potentially access every object and function in a program.10 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. End 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. End 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. End note]

```c++
rule
std::thread T(F, El);
• Bag

[k thread]

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

```
configuration:

END MODULE
Chapter 4

Iteration #3: LTL Model-Checking

This file includes an extension of the CinK-threads iteration with a small property language, including LTL formulas, and we show how the \( \mathcal{K} \) tool is used together with Maude system for analyzing CinK programs.

The definition must be parsed with the command

\[
\text{kompile cink -transition="kripke"}
\]

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

The definition can be tested with the online tool:

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)

\[
\text{SYNTAX } \quad \text{Pgm ::= LtlFormula}
\]

This module combines the syntax of the CinK language with that of the LTL formulas. The module MODEL-CHECKER-HOOKS is a \( \mathcal{K} \) 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 is an example of such proposition, whose intended semantics is to test whether the value of a variable in the configuration is equal to a given value. The semantics for this proposition will be given later, in the main module.

\[
\text{SYNTAX } \quad \text{Prop ::= eqTo (Id, Val)}
\]

\[
\mid \text{lt (Id, Val)}
\]

\[
\mid \text{leq (Id, Val)}
\]

\[
\mid \text{gt (Id, Val)}
\]

\[
\mid \text{geq (Id, Val)}
\]

\[
\mid \text{neqTo (Id, Val)}
\]

END MODULE

MODULE CINK-LTLMC-SEMANTICS

CONFIGURATION:
This is the main module that 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 \textit{Bag}:

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

\begin{verbatim}
SYNTAX   Int ::= val (Bag, Id)  [function]
RULE
\end{verbatim}

\begin{verbatim}
val (generatedTop, X), L, I

\texttt{val} (B, X)

\hline
\end{verbatim}

We are ready now to give the semantics for \texttt{eqTo}(X, I): it is satisfied by a configuration (state) $B$ iff the value of $X$ in $B$ is equal to $I$:

\begin{verbatim}
RULE
B \models_{Ltl} eqTo (X, I)  
  \texttt{true}()  
  \text{when } \texttt{val} (B, X) =_K I  
  \textit{[anywhere]}

RULE
B \models_{Ltl} neqTo (X, I)  
  \texttt{true}()  
  \text{when } \texttt{val} (B, X) \neq_K I  
  \textit{[anywhere]}

RULE
B \models_{Ltl} lt (X, I)  
  \texttt{true}()  
  \text{when } \texttt{val} (B, X) <_{\textit{Int}} I  
  \textit{[anywhere]}

RULE
B \models_{Ltl} leq (X, I)  
  \texttt{true}()  
  \text{when } \texttt{val} (B, X) \leq_{\textit{Int}} I  
  \textit{[anywhere]}
\end{verbatim}
4.2 The Main Module

4.3 Semantics

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

module CINK

configuration:

END MODULE
Chapter 5

Iteration #4: 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.

Related to the pointers is the ampersand (&) operator:

The result of the unary & operator is a pointer to its operand. The operand shall be an lvalue or a qualified id. If the operand is a qualified-id naming a non-static member m of some class C with type T, the result has type "pointer to member of class C of type T" and is a prvalue designating C::m. Otherwise, if the type of the expression is T, the result has type "pointer to T" and is a prvalue that is the address of the designated object (1.7) or a pointer to the designated function.

The definition can be tested with the online tool:

The Imported Modules

The list of imported modules imports those from the basic iteration, CINK-BASIC-SYNTAX,CINK-BASIC-SEMANTICS.

5.1 The New Modules

MODULE CINK-POINTERS-SYNTAX

SYNTAX Exp ::= * Exp [indirect]
| & Exp [ampersand]

END MODULE

MODULE CINK-POINTERS-SEMANTICS

Pointer type:

SYNTAX PtrType ::= pointer to Type

SYNTAX Type ::= PtrType

Extend the declarations with "pointer to type" type
Desugaring the type of a declaration

RULE

\[ T \ast X ; \]
\[ X \text{ of pointer to } T ; \]

[macro]

RULE

\[ * X \text{ of } PT ; \]
\[ X \text{ of pointer to } PT ; \]

[macro]

RULE

\[ \begin{array}{c}
  X \text{ of } T ; \\
  \ast \\
\end{array} \]
\[ \begin{array}{c}
  Env \\
  Env[L / X] \\
\end{array} \]

when \( \text{fresh}(L) \)

[ptr-decl, structural]

Declaration of an alias:

RULE

\[ \text{isAliasExp} ( \& E) \]
\[ \text{true}() \]

CONTEXT

\[ \& = \quad \square \quad ; \]
\[ \text{lvalue}(\square) \]

RULE

\[ \begin{array}{c}
  T \ast X = L ; \\
  \ast \\
\end{array} \]
\[ \begin{array}{c}
  Env \\
  Env[L / X] \\
\end{array} \]

The indirect operator

CONTEXT

\[ \ast \quad \square \quad \text{lvalue}(\square) \]

CONTEXT

\[ \text{lvalue}(\ast \square) \]

RULE

\[ \begin{array}{c}
  \text{lvalue}((\ast L)) \\
\end{array} \]
\[ \begin{array}{c}
  L' \\
\end{array} \]

RULE

\[ \begin{array}{c}
  \ast L \\
\end{array} \]
\[ \begin{array}{c}
  V \\
\end{array} \]

\[ L \mapsto L' \quad L' \mapsto V \]

\[ \text{store} \]
A pointer to a location is an \( lvalue \)

\[
\begin{align*}
\text{lvalue (} & \& \text{)} \\
\rule & \text{lvalue (□)}
\end{align*}
\]

The type of "\& L" is "pointer to the type of L", hence the type of "\* & L" is "type of L". \( L \) is evaluated as an \( rvalue \) that implies that the result is the value stored at \( L \).

\[
\begin{align*}
\text{rule} & \quad \text{isLVal (} & \& \text{L)} \\
\text{true()} & \\
\text{rule} & \quad \text{\* & L} \\
\text{store} & \quad \text{L} \mapsto \text{V}
\end{align*}
\]

5.2 The Main Module

Module CINK

Configuration:

End Module
Chapter 6

Iteration #5: Call-by-reference

This iteration extends the initial iteration with the call-by-reference mechanism. The definition can be tested with the online tool:

Imported Modules

The list of imported modules imports the basic ones, CINK-BASIC-SYNTAX, CINK-BASIC-SEMANTICS, CINK-POINTERS-SYNTAX, and CINK-POINTERS-SEMANTICS.

6.1 The New Module

MODULE CINK-CALL-BY-REF-SEMANTICS

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 \( \begin{array}{c}
\square, - \\
\text{lvalue}(\square)
\end{array} \) to \( - \) following \( (T & X), - \);

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

\[
\begin{array}{c}
\text{bind}(L, Vs) \text{ to } (T & X, Xl); \\
Vs \\
Xl
\end{array} \\
\]

END MODULE

6.2 The Main Module

MODULE CINK
CONFIGURATION:

END MODULE