Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 12

Control- and data-flow analysis

Several Kotlin features such as variable initialization analysis and smart casting analysis require performing control- and data-flow analyses. This section describes them and their applications.

12.1 Control flow graph

We define all control-flow analyses for Kotlin on a classic model called a control-flow graph (CFG). A CFG of a program is a graph which loosely defines all feasible paths the flow of a particular program can take during execution. All CFGs given in this section are *intraprocedural*, meaning that they describe the flow inside a *single* function, not taking function calls into account. CFG may, however, include multiple function bodies if said functions are *declared* inside each other (as is the case for lambdas).

The following sections describe CFG *fragments* associated with a particular Kotlin code construct. These fragments are introduced using visual notation rather than relational notation to simplify the understanding of the graph structure. To represent intermediate values created during computation, we use *implicit registers*, denoted `$1`, `$2`, `$3`, etc. These are considered to be unique in each CFG fragment (assigning the same register twice in the same CFG may only occur in unrelated program paths) and in the complete CFG, too. The numbers given are only notational.

We introduce special `eval` nodes, represented in *dashed lines*, to connect CFG fragments into bigger fragments. `eval x` here means that this node must be replaced with the whole CFG fragment associated with `x`. When this replacement is performed, the value produced by `eval` is the same value that the meta-register

`$result` holds in the corresponding fragment. All incoming edges of a fragment are connected to the incoming edges of the `eval` node, while all outgoing edges of a fragment are connected to the outgoing edges of the `eval` node. It is important, however, that, if such edges are absent either in the fragment or in the `eval` node, they (edges) are removed from the CFG.

We also use the `eval b` notation where `b` is not a single statement, but rather a control structure body. The fragment for a control structure body is the sequence of fragments for its statements, connected in the program order.

Some of the fragments have two kinds of outgoing edges, labeled `t` and `f` on the pictures. In a similar fashion, some `eval` nodes have two outgoing edges with the same labels. If such a fragment is inserted into such a node, only edges with matching labels are merged into each other. If either the fragment or the node have only unlabeled outgoing edges, the process is performed same as above.

For some types of analyses, it is important which boolean conditions hold on a control flow path. We use special `assume` nodes to introduce these conditions. `assume x` means that boolean condition `x` is always `true` when program flow passes through this particular node.

Some nodes are *labeled*, similarly to how statements may be labeled in Kotlin. Labeled nodes are considered CFG-unique and are handled as follows: if a fragment mentions a particular labeled node, this node is the same as any other node with this label in the complete CFG (i.e., a singular actual node is shared between all its labeled references). This is important when building graphs representing loops.

There are two other special kinds of nodes: `unreachable` nodes, signifying unreachable code, and `backedge` nodes, important for some kinds of analyses.

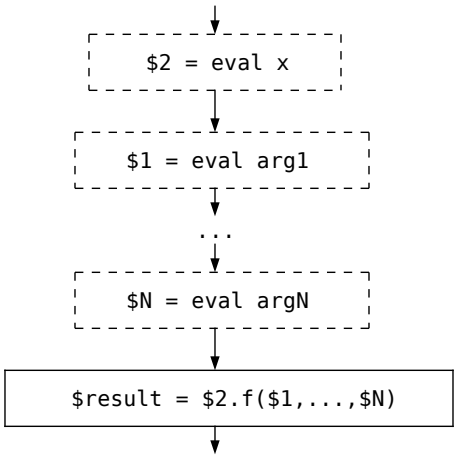
12.1.1 Expressions

Simple expressions, like literals and references, do not affect the control-flow of the program in any way and are irrelevant w.r.t. CFG.

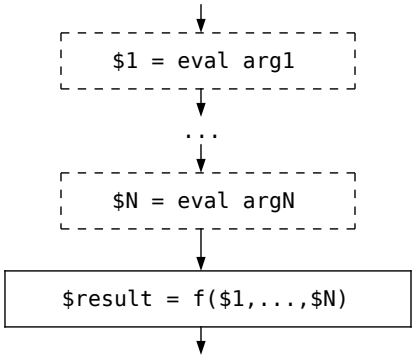
Function calls and operators

> Note: we do not consider operator calls as being different from function calls, as they are just special types of function calls. Henceforth, they are not treated separately.

```
x.f(arg1,..., argN)
```



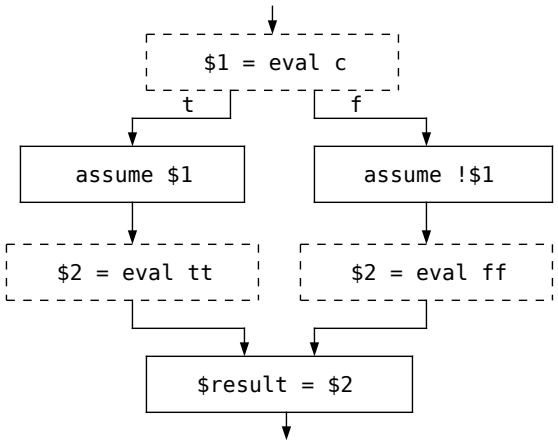

```
f(arg1,..., argN)
```



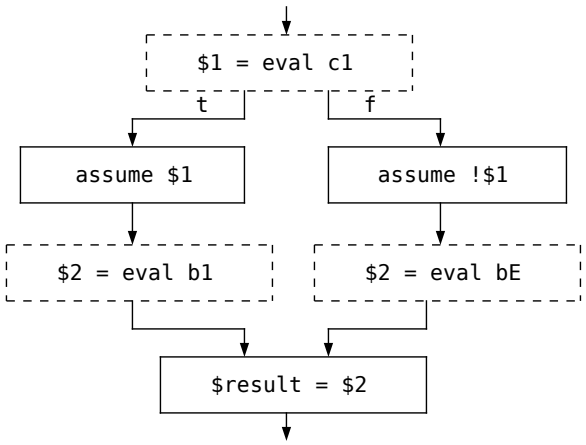
Conditional expressions

> Note: to simplify the notation, we consider only `if`-expressions with both branches present. Any `if`-statement in Kotlin may be trivially turned into such an expression by replacing the missing `else` branch with a `kotlin.Unit` object expression.

```
if(c) tt else ff
```




```
when {
    c1 -> b1
    else -> bE
}
```



Important: we only consider `when` expressions having exactly two branches for simplicity. A `when` expression with more than two branches may be trivially desugared into a series of nested when expression as follows:

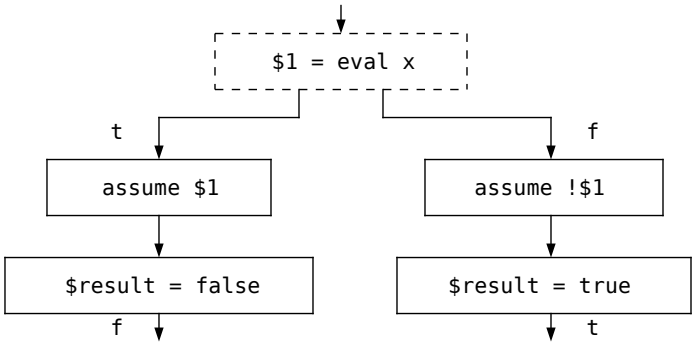
```
when {
    <entry1>
    <entries...>
    else -> bE
}
```

is the same as

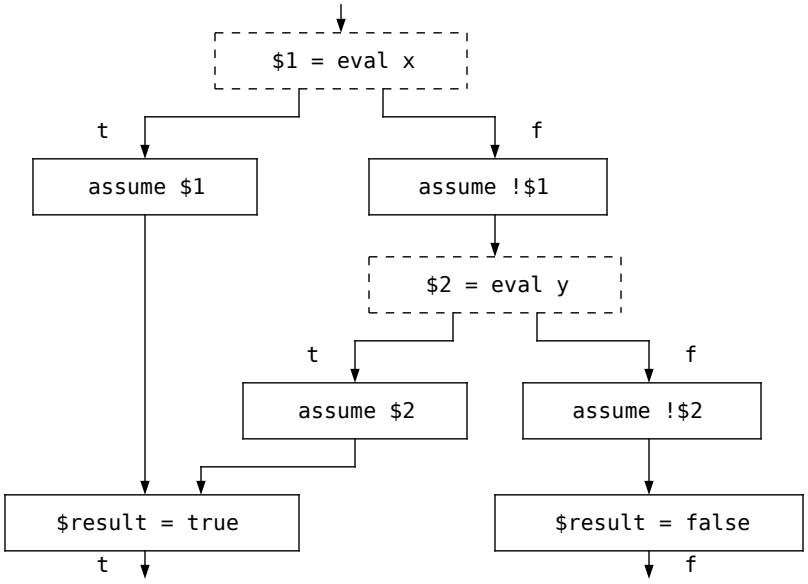
```
when {
    <entry1>
    else -> {
        when {
            <entries...>
            else -> bE
        }
    }
}
```

Boolean operators

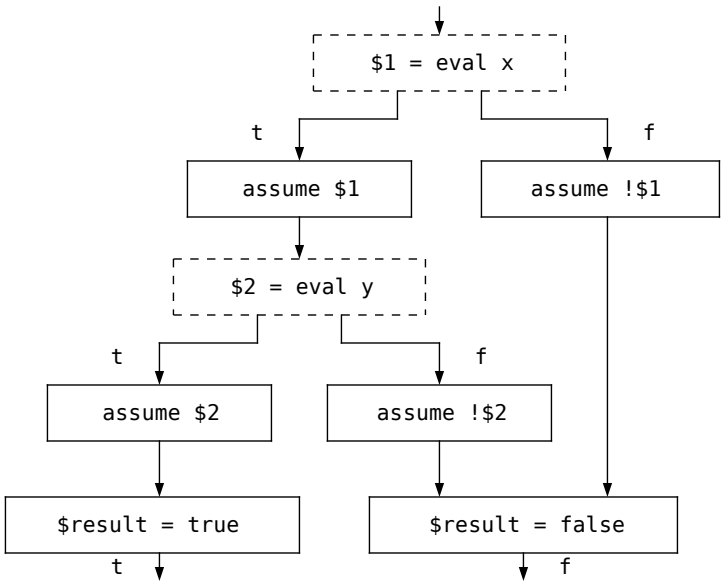
`!x`



`x || y`

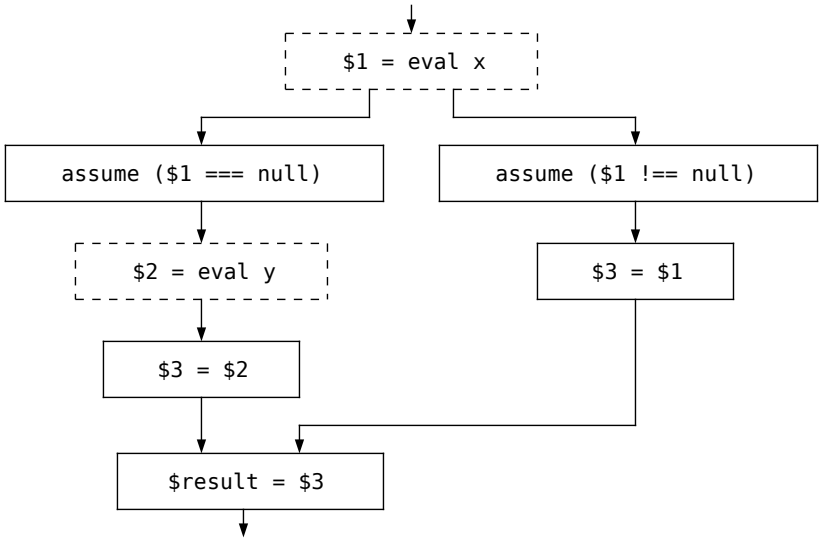


```
x && y
```

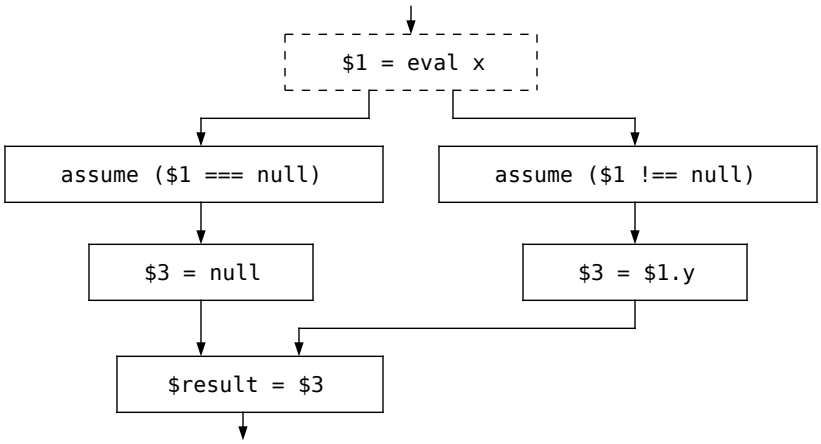


Other expressions

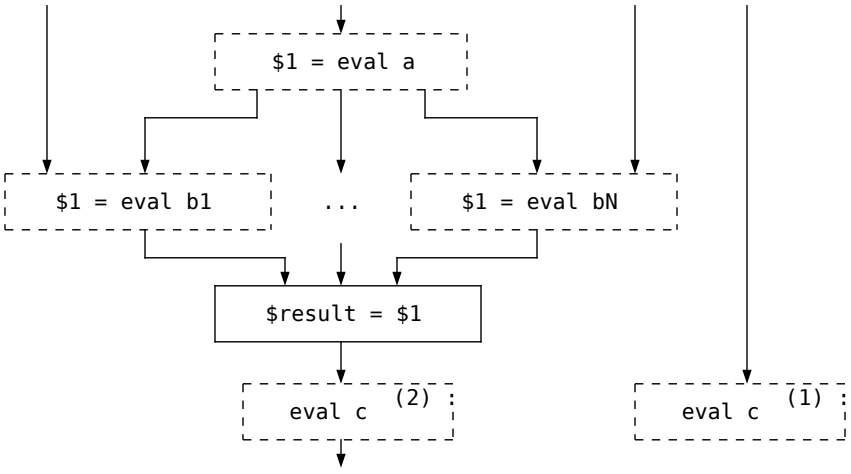
```
x ?: y
```




```
x?.y
```

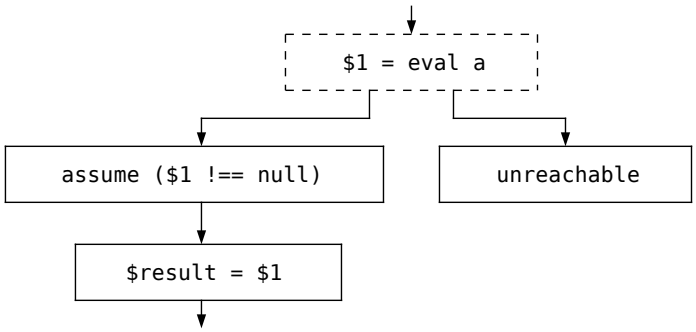



```
try { a... }
catch (e1: T1) { b1... }
...
catch (eN: TN) { bN... }
finally { c... }
```

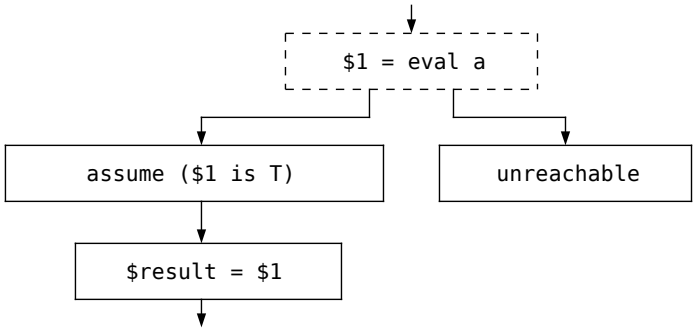


Important: in this diagram we consider `finally` block *twice*. The (1) block is used when handling the `finally` block and its body. The (2) block is used when considering the `finally` block w.r.t. rest of the CFG.

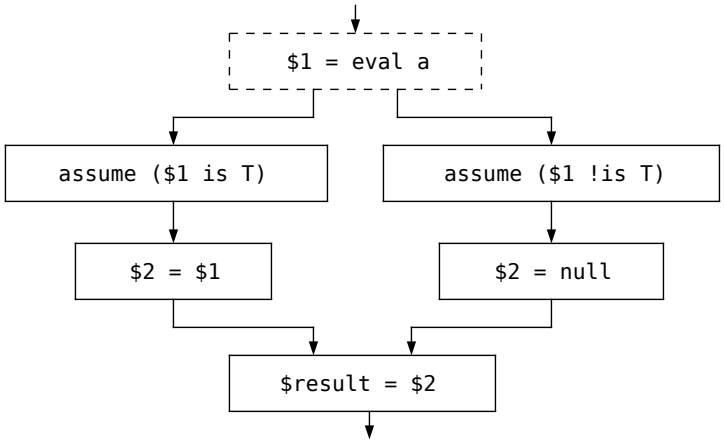
```
a!!
```



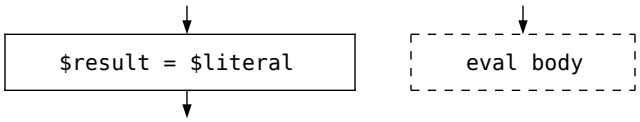

```
a as T
```



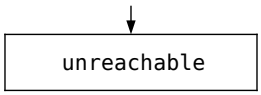

```
a as? T
```



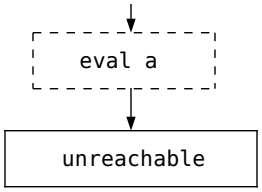

```
{ a: T ... -> body... }
```



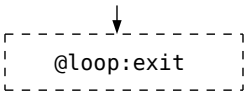

```
return
return@label
```



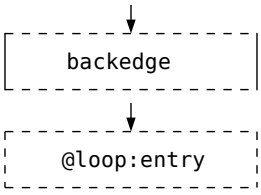

```
return a
return@label a
throw a
```




```
break@loop
```



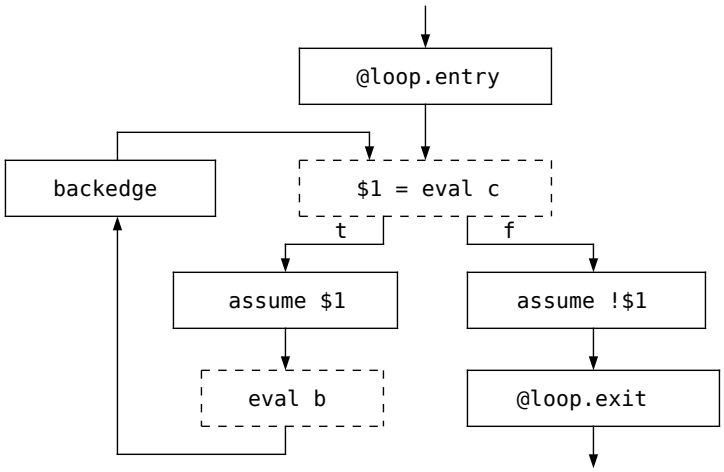

```
continue@loop
```



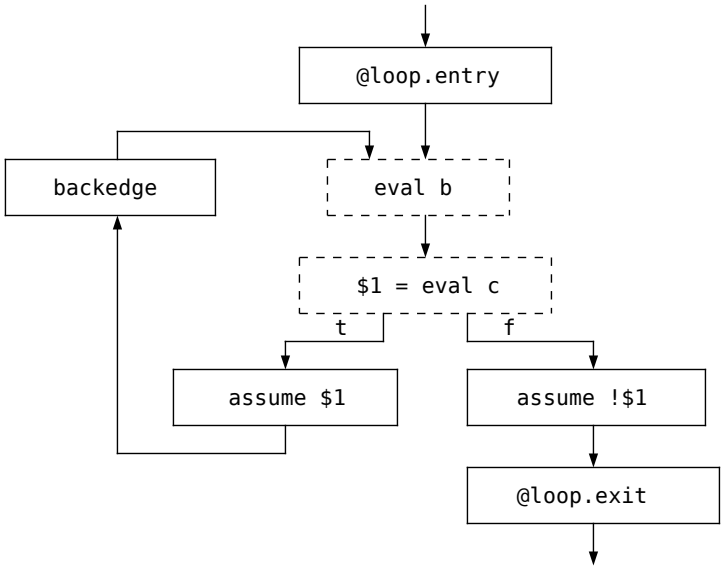
12.1.2 Statements

> Note: to simplify the notation, we consider only labeled loops, as unlabeled loops may be trivially turned into labeled ones by assigning them a unique label.

```
loop@ while(c) { b... }
```

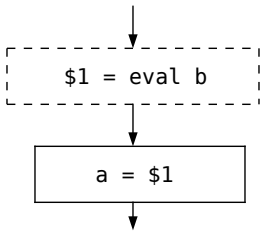



```
loop@ do { b... } while(c)
```

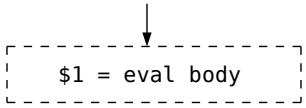


12.1.3 Declarations

```
var a = b
var a by b
val a = b
val a by b
```

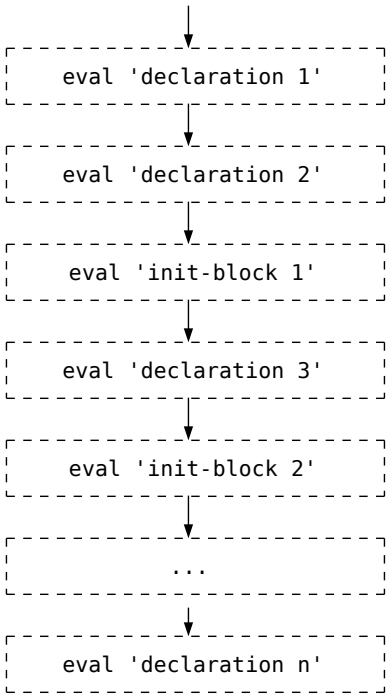



```
fun f() { body... }
```



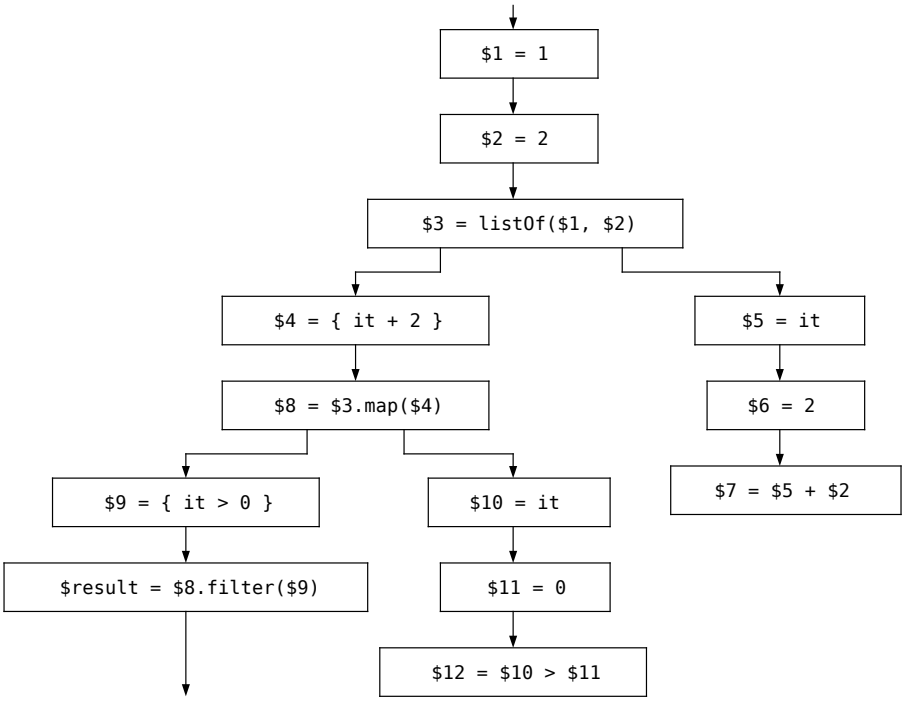

```
class A (...) {
    'declaration 1'
    'declaration 2'
    'init-block 1'
    'declaration 3'
    'init-block 2'
    ...
}
```

For every declaration and init block in a class body, the control flow is propagated through every element in the order of their appearance. Here we give a simplified example.

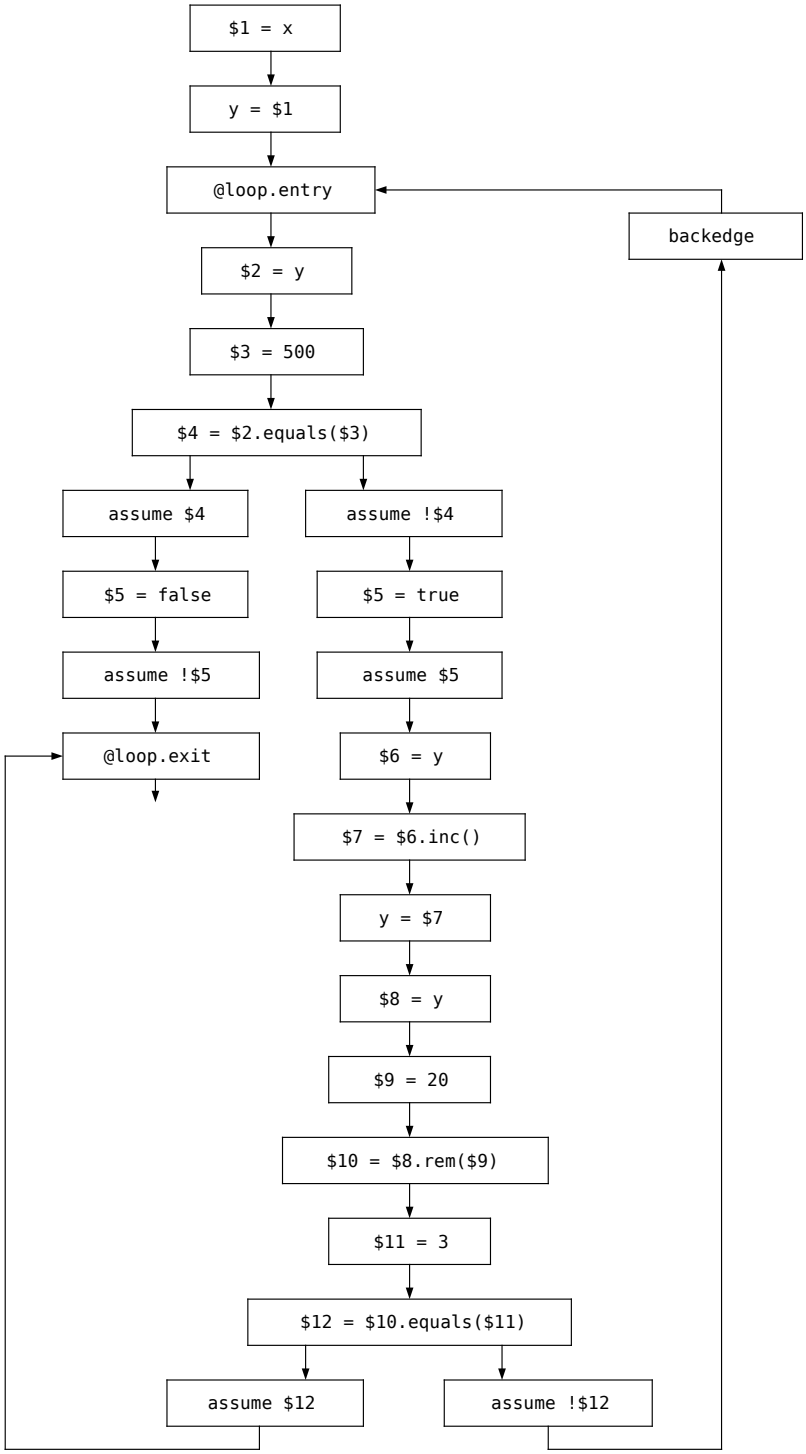


12.1.4 Examples

```
fun f() = listOf(1, 2).map { it + 2 }.filter { it > 0 }
```




```
fun f(x: Int) {
    var y = x
    loop@ while(y != 500) {
        y++
        if(y % 20 == 3) break@loop
    }
}
```


$1 = x
y = $1
@loop.entry
backedge
$2 = y
$3 = 500
$4 = $2.equals($3)
assume $4
assume !$4
$5 = false
$5 = true
assume !$5
assume $5
@loop.exit
$6 = y
$7 = $6.inc()
y = $7
$8 = y
$9 = 20
$10 = $8.rem($9)
$11 = 3
$12 = $10.equals($11)
assume $12
assume !$12

12.1.5 `kotlin.Nothing` and its influence on the CFG

As discussed in the type system section of this specification, `kotlin.Nothing` is an uninhabited type, meaning an instance of this type can never exist at runtime. For the purposes of control-flow graph (and related analyses) this means, as soon as an expression is known statically to have `kotlin.Nothing` type, all subsequent code is **unreachable**.

> Important: each specific analysis may decide to either use this information or ignore it for a given program. If unreachability from `kotlin.Nothing` is used, it can be represented in different ways, e.g., by changing the CFG structure or via *killDataFlow* instructions.

12.2 Performing analyses on the control-flow graph

The analyses defined in this document follow the pattern of analyses based on monotone frameworks, which work by modeling abstract program states as elements of lattices and joining these states using standard lattice operations. Such analyses may achieve limited path sensitivity via the analysis of conditions used in the `assume` nodes.

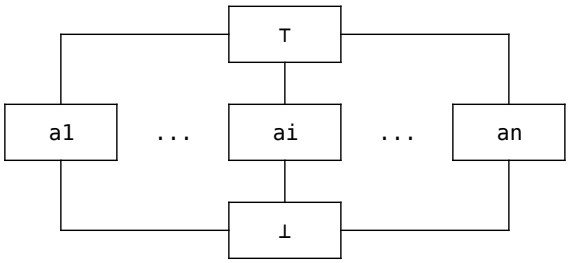
In short, an analysis is defined on the CFG by introducing:

- A lattice **S** (a partially ordered set that has both a greatest lower bound and a least upper bound defined for every pair of its elements) of values, called *abstract states*;
- A *transfer function* for mapping CFG nodes to the elements of **S**, essentially a set of rules on how to calculate an abstract state for each node of the CFG either directly or by using abstract states of other nodes.

The result of an analysis is a *fixed point* of the transfer function for each node of the given CFG, i.e., an abstract state for each node such that the transfer function maps the state to itself. For the particular shapes of the transfer function used in program analyses, given a finite **S**, the fixed point always exists, although the details of how this works go out of scope of this document.

12.2.1 Types of lattices

- Flat lattice over set $A = \{a1, \ldots, ai, \ldots, an\}$ of *incomparable* elements is built by adding a top element $\top$, which is *greater* than other elements, and a bottom element $\bot$, which is *less* than other elements. This forms the following lattice structure.



The flat lattice is usually used for analyses interested in *exact* facts, such as definite (un)assignment or constant propagation, as the fixed point results are either exact elements from the set A, or top/bottom elements.

- Map lattice of a set $A = \{a_1, \ldots, a_n\}$ to a lattice L is a lattice with sets of functions from A to L as its elements.

$$A \to L = \{[a_1 \to l_1, \ldots, a_n \to l_n] \,|\forall i : a_i \in A, l_i \in L\}$$
$$f \leq g \Leftrightarrow \forall a_i \in A : f(a_i) \leq g(a_i), \text{ where } f, g \in A \to L$$

The map lattice is usually used as the "top-level" lattice for bootstrapping the monotone framework analysis, by providing a way to represent the mapping from program entities (e.g., variables or expressions) to interesting facts (e.g., their initialization or availability) as a lattice.

12.2.2 Preliminary analysis and *killDataFlow* instruction

Some analyses described further in this document are based on special instruction called *killDataFlow*(v) where v is a program variable. These are not present in the graph representation described above and need to be inferred before such analyses may actually take place.

killDataFlow inference is based on a standard control-flow analysis with the lattice of natural numbers over "min" and "max" operations. That is, for every assignable property x an element of this lattice is a natural number N, with the least upper bound of two numbers defined as maximum function and the greatest lower bound as minimum function.

Note: such lattice has 0 as its bottom element and does not have a top element.

We assume the following transfer functions for our analysis.

$$\begin{aligned}
[\![\texttt{x = y}]\!]\,(s) &= s[x \to s(x) + 1] \\
[\![\texttt{backedge}]\!]\,(s) &= \{\star \to 0\} \\
[\![l]\!]\,(s) &= \bigsqcup_{p \in predecessor(l)} [\![p]\!]\,(s)
\end{aligned}$$

After running this analysis, for every backedge b and every variable x present in s, if $\exists b_p, b_s : b_p \in predecessors(b) \wedge b_s \in successors(b) \wedge [\![b_p]\!]\,(x) > [\![b_s]\!]\,(x)$, a $killDataFlow(x)$ instruction must be inserted after b.

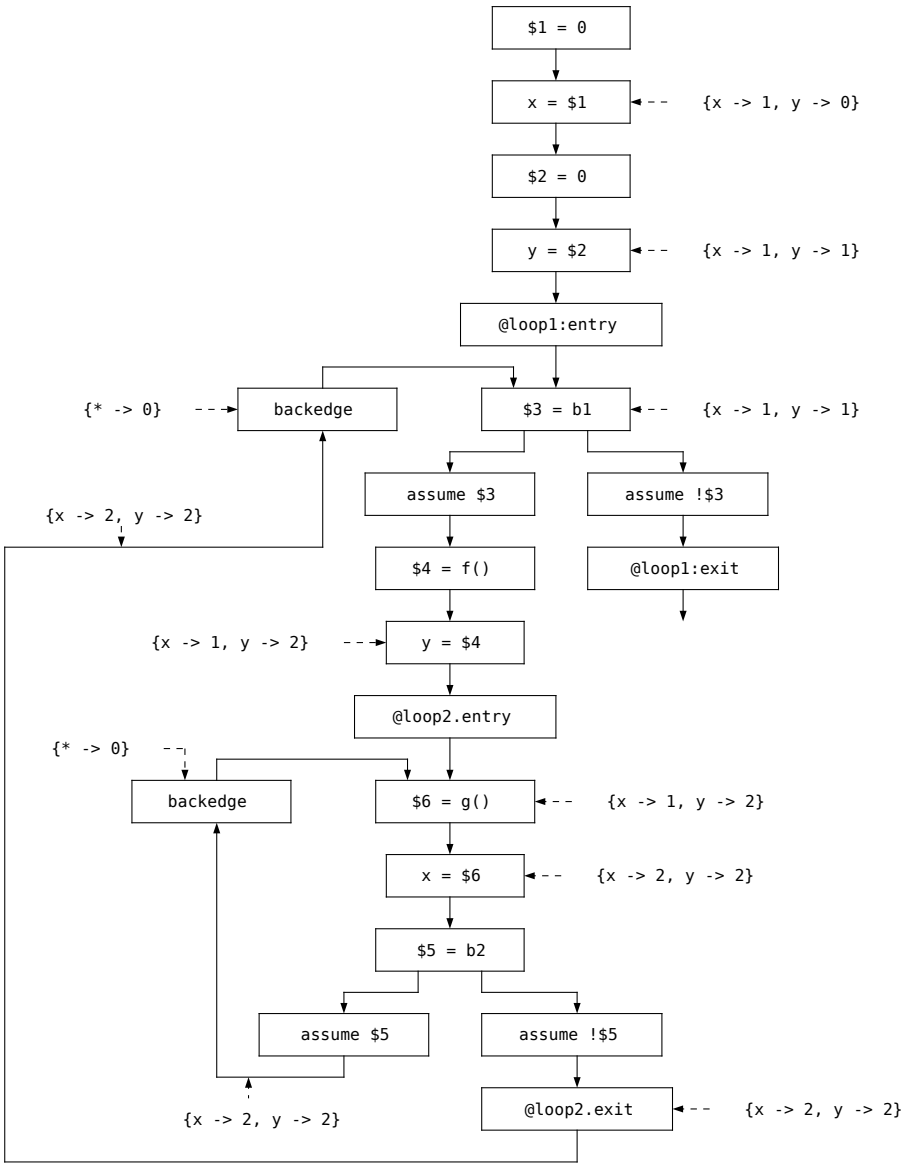
> Informally: this somewhat complicated condition matches variables which have been assigned to in the loop body w.r.t. this loop's backedge.

> Note: this analysis does involve a possibly **infinite** lattice (a lattice of natural numbers) and may seem to diverge on some graphs. However, if we assume that every backedge in an arbitrary CFG is marked with a `backedge` instruction, it is trivial to prove that no number in the lattice will ever exceed the number of assignments (which is **finite**) in the analyzed program as any loop in the graph will contain at least one backedge.

As an example, consider the following Kotlin code:

```
var x: Int = 0
var y: Int = 0
while (b1) {
    y = f()
    do {
        x = g()
    } while (b2)
}
```

which results in the following CFG diagram (annotated with the analysis results where it is important):



There are two backedges: one for the inner loop (the inner backedge) and one for the outer loop (the outer backedge). The inner backedge has one predecessor with state $\{\mathtt{x} \to 2, \mathtt{y} \to 2\}$ and one successor with state $\{\mathtt{x} \to 1, \mathtt{y} \to 2\}$ with the value for `x` being less in the successor, meaning that we need to insert *killDataFlow*(`x`) after the backedge. The outer backedge has one predecessor with state $\{\mathtt{x} \to 2, \mathtt{y} \to 2\}$ and one successor with state $\{\mathtt{x} \to 1, \mathtt{y} \to 1\}$ with values for both variables being less in the successor, meaning we need to insert *killDataFlow*(`x`) and *killDataFlow*(`y`) after the backedge.

12.2.3 Variable initialization analysis

Kotlin allows non-delegated properties to not have initializers in their declaration as long as the property is *definitely assigned* before its first usage. This property is checked by the variable initialization analysis (VIA). VIA operates on abstract values from the *assignedness* lattice, which is a flat lattice constructed over the set $\{Assigned, Unassigned\}$. The analysis itself uses abstract values from a map lattice of all property declarations to their abstract states based on the assignedness lattice. The abstract states are propagated in a forward manner using the standard join operation to merge states from different paths.

The CFG nodes relevant to VIA include only property declarations and direct property assignments. Every property declaration adds itself to the domain by setting the *Unassigned* value to itself. Every direct property assignment changes the value for this property to *Assigned*.

The results of the analysis are interpreted as follows. For every property, any usage of the said property in any statement is a compile-time error unless the abstract state of this property at this statement is *Assigned*. For every read-only property (declared using `val` keyword), any assignment to this property is a compile-time error unless the abstract state of this property is *Unassigned*.

As an example, consider the following Kotlin code:

```
/*  1 */ val x: Int     //{x → Unassigned, ⋆ → ⊥}
/*  2 */ var y: Int     //{x → Unassigned, y → Unassigned, ⋆ → ⊥}
/*  3 */ if (c) {        //
/*  4 */     x = 40     //{x → Assigned, y → Unassigned, ⋆ → ⊥}
/*  5 */     y = 4      //{x → Assigned, y → Assigned, ⋆ → ⊥}
/*  6 */ } else {       //
/*  7 */     x = 20     //{x → Assigned, y → Unassigned, ⋆ → ⊥}
/*  8 */ }             //{x → Assigned, y → ⊤, ⋆ → ⊥}
/*  9 */ y = 5         //{x → Assigned, y → Assigned, ⋆ → ⊥}
/* 10 */ val z = x + y //{x → Assigned, y → Assigned, z → Assigned}
```

There are no incorrect operations in this example, so the code does not produce any compile-time errors.

Let us consider another example:

```
/* 1 */ val x: Int     //{x → Unassigned, ⋆ → ⊥}
/* 2 */ var y: Int     //{x → Unassigned, y → Unassigned, ⋆ → ⊥}
/* 3 */ while (c) {     //{x → ⊤, y → ⊤, ⋆ → ⊥} Error!
/* 4 */     x = 40     //{x → ⊤, y → ⊤, ⋆ → ⊥}
/* 5 */     y = 4      //{x → ⊤, y → ⊤, ⋆ → ⊥}
/* 6 */ }              //
/* 7 */ val z = x + y //{x → ⊤, y → ⊤, ⋆ → ⊥} More errors!
```

In this example, the state of both properties at line 3 is $\top$, as it is the least upper bound of the states from lines 5 and 2 (from the `while` loop), which is derived to be $\top$. This leads to a compile-time error at line 4 for `x`, because one cannot reassign a read-only property.

At line 7 there is another compile-time error when both properties are used, as there are paths in the CFG which reach line 7 when the properties have not been assigned (i.e., the case when the `while` loop body was skipped).

12.2.4 Smart casting analysis

See the corresponding section for details.

12.2.5 Function contracts

> Note: as of Kotlin 1.9, contracts for user-defined functions are an experimental feature and, thus, not described here

Some standard-library functions in Kotlin are defined in such a way that they adhere to a specific *call contract* that affects the way calls to such functions are analyzed from the perspective of the caller's control flow graph. A function's call contract consists of one or more *effects*.

There are several kinds of effects:

- Calls-in-place effect for a function-type parameter of the function;
- Returns-implies-condition effect for a boolean parameter of the function;
- Particular implementations may introduce other types of effects.

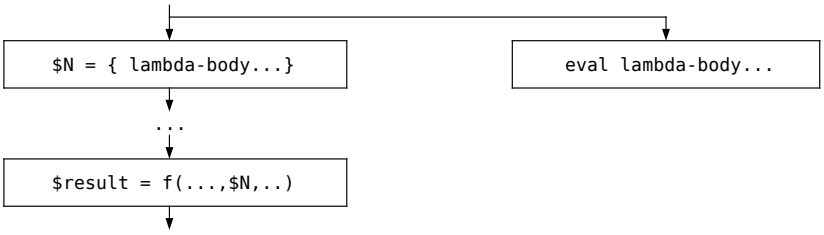
Calls-in-place effect of function F for a function-type parameter P specifies that for every call of F parameter P will be also invoked as a function. This effect may also have one of the three invocation types:

- *At-least-once*, meaning that P will be invoked at least once;
- *Exactly-once*, meaning that P will be invoked exactly once;
- *At-most-once*, meaning that P will be invoked at most once.

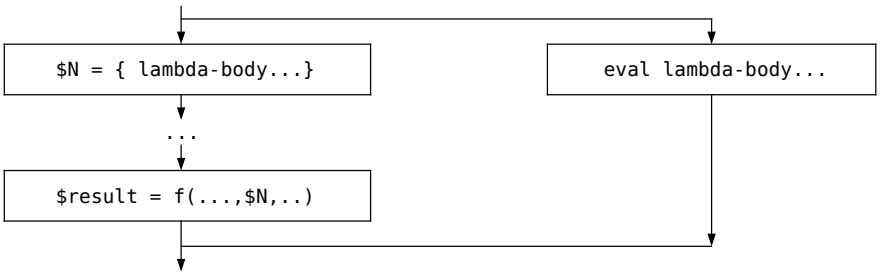
These effects change the call graph that is produced for a function call of F when supplied a lambda-expression parameter for P. Without any effect, the graph looks like this:

For a function call

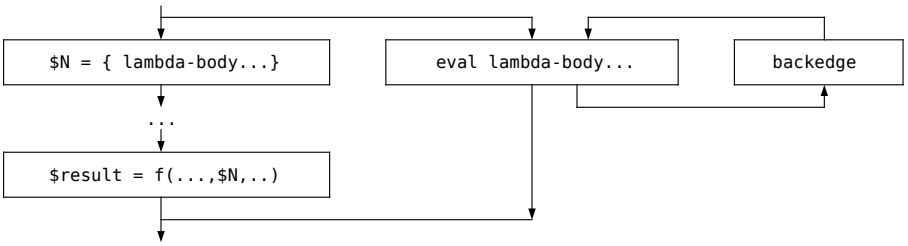
```
f(..., { lambda-body... }, ...)
```



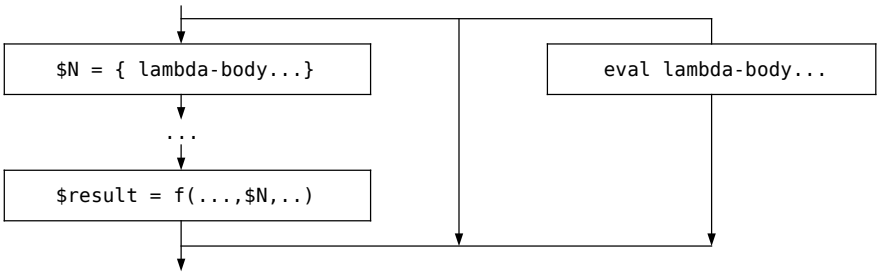
Please note that control flow information is passed inside the lambda body, but no information is extracted from it. If the corresponding parameter P is introduced with *exactly-once* effect, this changes to:



If the corresponding parameter P is introduced with *at-least-once* effect, this changes to:



If the corresponding parameter P is introduced with *at-most-once* effect, this changes to:



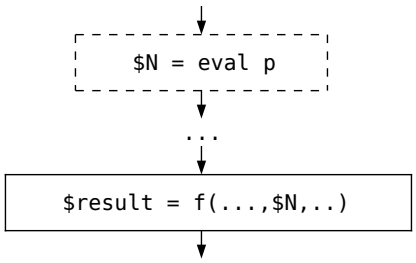
This allows the control-flow information to be extracted from lambda expression

according to the policy of its invocation.

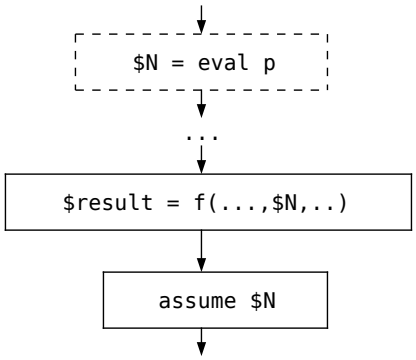
Returns-implies-condition effect of function F for a boolean parameter P specifies that if, when invoked normally, a call to F returns, P is assumed to be true. For a function call

```
f(..., p, ...)
```

this changes normal call graph that looks like this:



to look like this:



The following standard library functions have contracts with the following effects:

- `kotlin.run`, `kotlin.with`, `kotlin.let`, `kotlin.apply`, `kotlin.also` (all overloads): calls-in-place effect with invocation kind "exactly-once" for its functional argument;
- `kotlin.check`, `kotlin.require` (all overloads): returns-implies-condition effect on the boolean parameter.

Examples:

This code would result in a initialized variable analysis violation if `run` was not a standard function with corresponding contract:

```
val x: Int
run { // run invokes its argument exactly once
```

```
    x = 4
}
// could be error: x is not initialized
// but is ok
println(x)
```

Several examples of contract-introduced smart-cast:

```
val x: Any = ...
check(x is Int)
// x is known to be Int thanks to assume introduced by
// the contract of check
val y = x + 4 // would be illegal without contract
```

```
val x: Int? = ...
// x is known to be non-null thanks to assume introduced by
// the contract of require
require(x != null)
val y = x + 4 // would be illegal without contract
```

References

1. Frances E. Allen. “Control flow analysis.” ACM SIGPLAN Notices, 1970.
2. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. “Principles of program analysis.” Springer, 2015.
3. Kam, John B., and Jeffrey D. Ullman. “Monotone data flow analysis frameworks.” Acta informatica 7.3 (1977): 305-317.
4. Anders Moller, and Michael I. Schwartzbach. “Static Program Analysis.” 2018 (https://cs.au.dk/~amoeller/spa/)