Verifying Duff's device

A simple compositional denotational semantics for Goto and computed jumps

Hendrik Tews

Institut für Theoretische Informatik, TU Dresden, Germany
http://wwwtcs.inf.tu-dresden.de/∼tews

Abstract. This paper presents a compositional denotational semantics for goto jumps. The semantics interacts well with the idioms of structural programming, like if-statements and while-loops. It faithfully models jumps into and out of nested blocks and it can also handle computed jumps (in the form of C's switch-statement). The underlying mathematics is remarkable simple: It only uses total functions of ordinary set theory and disjoint unions. Neither domain theory nor continuous functions are required. The paper demonstrates the semantics for a simple while language containing assignments, goto jumps and switch-statements. The semantics is used to verify Duff's device, a famous example of the (ab-)use of computed jumps in structural programming.
The semantics presented here is a generalisation of Huismann and Jacobs semantics for abrupt termination in Java [1].

1 Introduction

Nowadays most programmers follow the rules of structured programming unless some reason forces them not to do so. There are several such possible reasons. For instance a few forward jumps `goto success` might lead to much clearer code than the equivalent deeply nested if-statements. Another reason might be that the right idioms of structured programming are not available. For instance in C or C++ one can not `break` or `continue` with respect to an outer loop. Sometimes the programmer avoids the structured programming idioms because they are too expensive. For instance the C++ source code of the Fiasco [2] microkernel uses setjmp and longjmp.[1] Alternatively exceptions could have been used. However, with `gcc` exceptions require a big runtime library, which the developer of Fiasco wanted to avoid. We can conclude that unstructured idioms are still present in newly written code [3].

The described situation poses a challenge for theoretical computer science. In the fields of semantics and verification it is necessary to also discuss and provide solutions for unstructured and "dirty" idioms like goto jumps or type casts. The point here is not to provide support for badly written programs (although this would be an interesting challenge in itself). The point is to provide semantics and verification support to well written programs, in which occasionally unstructured or "dirty" idioms *cannot be avoided*.

This paper investigates the denotational semantics of goto statements in an otherwise high-level programming language, like C++ or Ada. There is a long tradition in denotational semantics, see the overview articles [4, 5] and the citations therein. Over the years solutions for

[1] In essence setjmp saves the program counter and the stack pointer and longjmp simply restores them. A longjmp never returns, instead one leaves setjmp a second time.

more and more complex programming idioms have been developed. The underlying mathematics is fairly complex. Complete partial orders and continuous functions are always required as a basis. Often solutions for recursive domain equations are needed.

This paper presents a denotational semantics for a subset of C++ that contains basic expressions without side effects, assignments to integer variables and integer arrays and the statements `break`, `goto`, `if`, `while` and `switch`. The subset is called GotoWhile here. Its semantics requires only basic set theory. Therefore I would claim that it is simple enough to be comprehensible to every programmer, implementer, and language designer. This claim is in stark contrast to R. D. Tennents statement [4, page 176] saying that the mathematics necessary for rigorous semantic descriptions is too challenging.

Features like functions, local variables, objects and methods have only been excluded from GotoWhile for reasons of space and originality. The semantics presented here can easily be extended with standard solutions for these features. In fact, this has been done: GotoWhile forms a part of my lecture on denotational semantics and verification [6]. For this lecture I implemented the semantics (in the form of a shallow embedding) in the interactive theorem prover PVS [7]. I used this embedding for a number of example verifications, most notably to verify Duff's device (see Section 5). The semantic equations, the definitions and the theorems in the following sections are striped-down versions of definitions and theorems in the PVS source code. The complete PVS material is available in the World Wide Web at wwwtcs.inf.tu-dresden.de/∼tews/Goto.

The background work of this paper is the VFiasco project. The aim of this project is to apply source code verification to the *unmodified* sources of the Fiasco microkernel operating system. Fiasco is written almost entirely in C++ and it *does contain* a few goto jumps, setjmp/longjmp and many type casts (which are unavoidable in an operating system). The semantics for goto jumps of this paper forms a part of the semantics of C++ that we are developing in the VFiasco project (see also [8] for the VFiasco approach to data types and type casts).

In the next section I give the abstract syntax and intended semantics of GotoWhile. Section 3 defines its denotational semantics. With the exception of loops (constructed with `while` or `goto`) this semantics is operational and works nicely in proofs (at least in PVS). Section 4 proves a Hoare-like total correctness theorem for while loops, which forms the core of the total correctness proof of Duff's device in Section 5.

Acknowledgements I thank Michael Hohmuth for constantly pushing me against the edge of well-known denotational semantics and for continuous discussions about the subject. This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through DFG grant Re 874/2–3.

Notation I use the standard notation of set theory with the following exceptions: The function space between two sets X and Y is denoted with $X \Rightarrow Y \overset{\text{def}}{=} \{f \mid f : X \longrightarrow Y\}$. Function update is written as $f[a := b]$. The unit is $\mathbf{1} \overset{\text{def}}{=} \{*\}$. For disjoint unions with many components I prefer to have meaningful names for the injections instead of the standard κ_i. The notation $\overset{inx:}{X} \uplus \overset{iny:}{Y} \overset{\text{def}}{=} (\{0\} \times X) \cup (\{1\} \times Y)$ describes the disjoint union of X and Y with associated injections $inx : X \longrightarrow X \uplus Y$ and $iny : Y \longrightarrow X \uplus Y$. Similarly $\overset{px:}{X} \times \overset{py:}{Y}$ describes a cartesian product with the projections px and py. For $z \in \overset{inx:}{X} \uplus \overset{iny:}{Y}$ I write $z = inx(-)$ as shorthand of $\exists x \in X \,.\, z = inx(x)$. Some bits of the semantics are instances of well-known abstract notions such as monads. I generally avoid these abstract notions here because they do not contribute any insight in this work.

$$
\begin{array}{rcl}
\textit{expr} & ::= & n \mid \textit{ivar} \mid \textit{avar}[\textit{expr}] \mid \textit{expr}\ \textit{op}\ \textit{expr} \mid !\textit{expr} \\
\textit{op} & ::= & + \mid * \mid / \mid \% \mid == \\
\\
\textit{labeled-statement} & ::= & \textit{statement} \\
& \mid & \texttt{case}\ i\text{: }\textit{statment} \qquad\qquad (i \in \mathbb{Z}) \\
& \mid & \texttt{default}\text{: }\textit{statment} \\
& \mid & \textit{string-label}\text{: }\textit{statment} \\
\\
\textit{statement} & ::= & \texttt{nop} \\
& \mid & \textit{ivar} = \textit{expr} \\
& \mid & \textit{avar}[\textit{expr}] = \textit{expr} \\
& \mid & \texttt{break} \\
& \mid & \texttt{goto}\ \textit{label} \\
& \mid & \texttt{if}(\textit{expr})\ \textit{labeled-statement}\ \texttt{else}\ \textit{labeled-statement} \\
& \mid & \texttt{while}(\textit{expr})\ \textit{labeled-statement} \\
& \mid & \texttt{switch}(\textit{expr})\ \textit{labeled-statement} \\
& \mid & \texttt{new\_array}\ \textit{avar}\ \textit{expr}\ \textit{expr} \\
& \mid & \{\ \textit{labeled-statement};\ \ldots\ \} \\
\\
\textit{program} & ::= & \textit{labeled-statement}
\end{array}
$$

Fig. 1. A simple while language with goto statements

2 A simple `while` language

In this section I describe the programming language for which I develop a denotational semantics in the following section. The language is called GotoWhile here. Figure 1 shows its abstract syntax. With two exceptions (`nop` and `new_array`) GotoWhile is a fragment of C or C++. The intended semantics that I describe in the following is precisely the C++ semantics [9].

For simplicity, GotoWhile has very simple expressions and there are no advanced features like procedures, functions or local variables. These features can be added without problem, using one of the standard semantics from [10, 5, 4]. The aim here is not to present a complete or nice programming language. Instead I chose a language fragment that contains `goto` and `switch` and that is rich enough to express Duff's device.

Expressions of GotoWhile are integers (denoted with n), integer variables ($ivar$), integer array expressions ($avar[expr]$, where $avar$ stands for an array variable), arithmetic expressions with addition ($+$), multiplication ($*$), integer division ($/$) and modulus ($\%$) and boolean expressions with equality ($==$) and negation ($!$). The integer arrays are not especially interesting here, I only need them to formulate Duff's device in GotoWhile in Section 5. The expressions behave as expected. However, note that the array lookup might fail if the index is out of the array bounds.

The labeled statements are the interesting (and challenging) part of GotoWhile. Statements can be labeled with either integers (i) or strings ($string - label$). The string labels denote targets for goto statements. Every integer-labeled statement must be enclosed by a *switch* and belongs to the closest switch surrounding it. Both integer labels and string labels can occur at an arbitrary nesting level (and not only at the top level of a program or a switch statement). Therefore the following is a correct (but nonsensical) GotoWhile program:[2]

[2] The program is legal C if enclosed with variable declarations and `main` and if `nop` is `#define`'d as the empty string.


```
if(a == 0)
  goto lab;
else
  nop;
switch(b){
  case 0: if(c == 0)
     lab:    d = 1;
          else
  case 1:    d = 2;
}
```

In this program the goto statement jumps into the middle of the body of the switch statement. Further, the case label 1 occurs in the else clause of the second if statement. So if `b` is 1 then control is transfered directly to the last assignment, without evaluating the condition of the if statement. The challenge is to come up with a semantics that can deal with this kind of programs.

The statements of GotoWhile are the usual ones with a few additions. The nop statement, corresponding to the empty statement of C, does nothing. It is included here only to make empty then and else clauses possible. Assignments are written as in C or Java. The `break` statement immediately transfers control to the statement following the enclosing `switch` or `while` statement. Break is mainly included here for a comparison with Huismann and Jacobs semantics for abrupt termination. In a real programming language one would also like to permit break statements with labels and also continue statements with labels as Java does. However, there would be nothing new in the semantics for the labels and for continue, so they are omitted for brevity.

The goto statement does the obvious. Similarly to C++, programs that contain two statements with the same label are ill formed and every label in a goto statement must label some statement. There are no restrictions on where the jump target is located.[3]

The if and the while statements behave as usual, except that it is possible to jump (either by goto or by switch) into the middle of a then or else clause or into the body of a while loop. If such a jump occurs, everything before the target label is skipped. However, beginning with the target label, the execution continues as usual. When jumping into the middle of a while loop then, after reaching the end of its body, the condition is evaluated and execution continues depending on the result of the condition. Similarly, if you jump into the then clause of an if statement you will skip the else clause (unless you directly jump into it).

The switch statement evaluates the expression and jumps to the statement that is labeled with the value of the expression. If there is no such statement, it jumps to the statement labeled with `default`. If there is no default label, it jumps to the statement following the switch. As in C++ the labels in a switch statement must be unique and there must be at most one default label in a given switch. The switch statement has the famous "fall-through" feature as known from C, C++ and Java. One has to add explicit break statements in order to leave the switch.

The statement `new_array avar size init` evaluates the size and init expressions and then initialises `avar` with an array of the given size in which all cells hold the value of `init`. The statement fails if the value of `size` is less than zero. The `new_array` statement is only included for completeness here.

[3] C++ forbids to jump into other functions. Further one must not jump over the declaration of so-called non-pod data. [9]

Composition of statements is juxtaposition and blocks can be formed with curly braces.

3 Denotational Semantics

In this section I define a compositional denotational semantics for GotoWhile. Traditionally functions of the form $St \longrightarrow St \uplus \mathbf{1}$ (or equivalently partial functions $St \rightharpoonup St$) have been taken as denotations of while programs. Here, and in the following St denotes a suitable set of program states. This traditional approach can model simple while programs. With a suitable structure on the states in St one can also deal with procedures and local variables. The technique of continuations makes it possible to model nonlocal exits, like the break-statement. However, up to my knowledge, a compositional denotational semantics for unrestricted goto jumps has not been described yet.

In their denotational semantics of Java [1], Huismann and Jacobs generalise the denotations to coalgebras of the form $St \longrightarrow \mathit{StatResult}$, where $\mathit{StatResult}$ is a disjoint union of six components. Under the right perspective $\mathit{StatResult}$ can be viewed as a monad and then the semantics of composition becomes composition in the associated Kleisli category [11]. The approach of Huismann and Jacobs only requires basic set theory. A complete partial ordering is only needed for recursive Java methods. The semantics of Huismann and Jacobs models abrupt termination via break, return or continue statements in an exceptionally simple and elegant way, without resorting to continuations.

The basic idea of Huismann and Jacobs is as follows. One of the injections of $\mathit{StatResult}$ models normal termination. Composition is defined such that a statement is only executed if the preceding statement terminates normally. The semantics of the break statement, for instance, always returns a break abnormality. Thus the composition of statements skips everything that follows a break. Special functionals that enclose (the semantics of) whole blocks transform abnormalities to normal when appropriate. Statements following a while loop are therefore executed again after the loop terminated with a `break`. Note that in Jacobs approach only functionals (i.e., functions on $St \Rightarrow \mathit{StatResult}$) can convert abnormalities to normal.

For the semantics of GotoWhile I generalise Huismann and Jacobs approach and use functions of the form $\mathit{SRes} \longrightarrow \mathit{SRes}$, where SRes is as before a disjoint union with many components (see Figure 2 for details). The generalisation makes it possible to insert a function that catches a certain abnormality in between two arbitrary statements. This possibility is precisely what is needed for labels: Their semantics converts goto abnormalities (produced by goto statements) to normal.

3.1 Program states and the semantics of expressions

Figures 2–7 contain the denotational semantics of GotoWhile. As usual I only consider type-correct programs. However, sometimes the semantics checks a type-correctness condition to simplify the definitions. Figure 2 describes the relevant sets and the semantics of expressions. I let $\mathbb{N}$ denote the set of natural numbers, $\mathbb{N}^+$ the positive natural numbers, $\mathbb{Z}$ the integers, $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ the booleans, $\mathbb{V}$ the set of all variables of GotoWhile and $\mathbb{L}$ the set of string labels.

For the purpose of this paper the structure of program states is rather irrelevant, as long as a state can store the value of variables. However, I prefer to give a concrete definition. The set of program states, denoted by *State*, is the set of functions mapping variables to values. A value is either an integer (injection *var*) or an array (injection *arr*). An array consists of a tuple,

$\mathbb{N}$: natural numbers $\mathbb{N}^+$: positive natural numbers $\mathbb{Z}$: integers
$\mathbb{B}$: booleans $\mathbf{1}$: unit $\mathbb{V}$: variables $\mathbb{L}$: string labels

$$
\begin{array}{rcl}
\mathit{Array} & = & \overset{\mathit{size}:}{\mathbb{N}} \times \overset{\mathit{fields}:}{(\mathbb{N} \Rightarrow \mathbb{Z})} \\
\mathit{Value} & = & \overset{\mathit{var}:}{\mathbb{Z}} \uplus \overset{\mathit{arr}:}{\mathit{Array}} \\
\mathit{State} & = & \mathbb{V} \Rightarrow \mathit{Value} \\
\mathit{ExRes}_T & = & \overset{\mathit{ok}:}{T} \uplus \overset{\mathit{fail}:}{\mathbf{1}} \\
\mathit{SRes} & = & \overset{\mathit{ok}:}{\mathit{State}} \uplus \overset{\mathit{break}:}{\mathit{State}} \uplus \overset{\mathit{case}:}{(\mathit{State} \times \mathbb{Z})} \uplus \\
 & & \qquad \overset{\mathit{default}:}{\mathit{State}} \uplus \overset{\mathit{goto}:}{(\mathit{State} \times \mathbb{L})} \uplus \overset{\mathit{fail}:}{\mathbf{1}} \uplus \overset{\mathit{hang}:}{\mathbf{1}}
\end{array}
$$

$$
\begin{array}{rcll}
[\![\mathit{expr}]\!] & : & \mathit{State} \longrightarrow \mathit{ExRes}_T & T \in \{\mathbb{Z}, \mathbb{B}\} \\
[\![\mathit{ivar}]\!](s) & = & \begin{cases} \mathit{ok}(i) & \text{if } s(\mathit{ivar}) = \mathit{var}(i) \\ \mathit{fail} & \text{otherwise} \end{cases} & s \in \mathit{State}, \mathit{ivar} \in \mathbb{V} \\
[\![\mathit{avar}[\mathit{expr}]]\!](s) & = & \begin{cases} \mathit{ok}(i) & \begin{array}{l} \text{if } [\![\mathit{expr}]\!](s) = \mathit{ok}(n) \;\wedge \\ \quad s(\mathit{avar}) = \mathit{arr}(\mathit{size}, \mathit{fields}) \;\wedge \\ \qquad 0 \le n < \mathit{size} \;\wedge\; \mathit{fields}(n) = i \end{array} \\ \mathit{fail} & \text{otherwise} \end{cases} & \\
[\![\mathit{ex}_1 \; \mathit{op} \; \mathit{ex}_1]\!](s) & = & \begin{cases} \mathit{ok}(i_3) & \begin{array}{l} \text{if } [\![\mathit{ex}_1]\!](s) = \mathit{ok}(i_1) \;\wedge\; [\![\mathit{ex}_2]\!](s) = \mathit{ok}(i_2) \;\wedge \\ \hfill [\![\mathit{op}]\!](i_1, i_2) = i_3 \end{array} \\ \mathit{fail} & \text{otherwise} \end{cases} & \\
[\![!\mathit{expr}]\!](s) & = & \begin{cases} \mathit{ok}(\neg b) & \text{if } [\![\mathit{expr}]\!](s) = \mathit{ok}(b) \\ \mathit{fail} & \text{otherwise} \end{cases} &
\end{array}
$$

Fig. 2. Relevant sets and the semantics of expressions

where the first component is its size (projection *size*) and the second component (projection *fields*) is a function mapping indices to integers.

The program states are kind of dynamically typed. Thus it is possible that the access to an integer variable fails because the current state stores an array for this variable. Similarly, an array access can fail because the index is too large or negative. The former kind of error cannot occur in well-typed programs. However, the latter kind cannot be sensibly excluded. Therefore the semantics I am defining has to deal with undefined expressions.

In the semantics I let $[\![-]\!]$ denote the semantic function. Thus $[\![-]\!]$ is overloaded for every syntactic category. The semantics of expressions is completely standard. Because expressions of GotoWhile cannot have side effects, it suffices to use functions $\mathit{State} \longrightarrow \mathit{ExRes}_T$, where T is either $\mathbb{Z}$ or $\mathbb{B}$, as denotations. The set ExRes_T is a disjoint union of T (denoting the value of an expression) and $\mathbf{1} = \{*\}$ (denoting one of the errors described above).

The semantics of an integer variable *ivar* is as expected: If the current state maps *ivar* to an integer then it returns this integer, otherwise it fails. In the semantics of an array expression

$$
\begin{array}{rcl}
\llbracket \mathit{statement} \rrbracket & : & \mathit{SRes} \longrightarrow \mathit{SRes} \\
\llbracket \texttt{nop} \rrbracket & = & \left| \, x \mapsto x \right. \\
\llbracket v = \mathit{expr} \rrbracket & = & \left| \begin{array}{lll} \mathit{ok}(s) & \mapsto & \begin{cases} \mathit{ok}(s[v := \mathit{var}(i)]) & \text{if } \llbracket \mathit{expr} \rrbracket(s) = \mathit{ok}(i) \\ \mathit{fail} & \text{otherwise} \end{cases} \\ x & \mapsto & x \end{array} \right. \\
\mathit{upd\_arr} & : & \mathit{State} \times \mathbb{V} \times \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathit{SRes} \\
\mathit{upd\_arr}(s, a, i, v) & = & \begin{cases} \mathit{ok}\big(s\big[a := \mathit{arr}(si, \mathit{fi}[i := v])\big]\big) \\ \qquad\qquad\qquad \text{if } s(a) = \mathit{arr}(si, \mathit{fi}) \ \wedge \ 0 \le i < si \\ \mathit{fail} \qquad\qquad \text{otherwise} \end{cases} \\
\llbracket a[ex_1] = ex_2 \rrbracket & = & \left| \begin{array}{lll} \mathit{ok}(s) & \mapsto & \begin{cases} \mathit{upd\_arr}(s, a, i, v) & \text{if } \llbracket ex_1 \rrbracket(s) = \mathit{ok}(i) \ \wedge \\ & \qquad \llbracket ex_2 \rrbracket(s) = \mathit{ok}(v) \\ \mathit{fail} & \text{otherwise} \end{cases} \\ x & \mapsto & x \end{array} \right. \\
\llbracket \texttt{break} \rrbracket & = & \left| \begin{array}{lll} \mathit{ok}(s) & \mapsto & \mathit{break}(s) \\ x & \mapsto & x \end{array} \right. \\
\llbracket \texttt{goto } l \rrbracket & = & \left| \begin{array}{lll} \mathit{ok}(s) & \mapsto & \mathit{goto}(s, l) \\ x & \mapsto & x \end{array} \right. \\
\llbracket \texttt{new\_array } av \ sx \ vx \rrbracket & = & \left| \begin{array}{lll} \mathit{ok}(s) & \mapsto & \begin{cases} \mathit{ok}(s[av := \mathit{arr}(si, \lambda n \,.\, v)]) \\ \qquad \text{if } \llbracket sx \rrbracket(s) = \mathit{ok}(si) \wedge \llbracket vx \rrbracket(s) = \mathit{ok}(v) \\ \mathit{fail} \qquad \text{otherwise} \end{cases} \\ x & \mapsto & x \end{array} \right. \\
\llbracket st_1; st_2 \rrbracket & = & \llbracket st_2 \rrbracket \circ \llbracket st_1 \rrbracket \\
\llbracket \mathit{expr} \rrbracket^{\#} & : & \mathit{SRes} \longrightarrow \mathit{SRes} \\
\llbracket \mathit{expr} \rrbracket^{\#}(x) & = & \begin{cases} \mathit{fail} & \text{if } x = \mathit{ok}(s) \ \wedge \ \llbracket \mathit{expr} \rrbracket(s) = \mathit{fail} \\ x & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 3. Semantics of statements. Part I: simple statements and composition

$avar[expr]$ we have to check if the expression $expr$ does not fail,[4] if the current state s holds an array for $avar$ and finally if the value of the expression obeys the array bounds.

The semantics of the binary operations and of negation contains no surprise. Here I assume the standard definitions of addition $\llbracket + \rrbracket$, multiplication $\llbracket * \rrbracket$, integer division $\llbracket / \rrbracket$ (rounding towards zero), modulus $\llbracket \% \rrbracket$, and equality $\llbracket == \rrbracket$.

3.2 Simple statements and composition

Figure 3 describes the semantics of unlabeled simple statements and of composition. As mentioned before I take functions of the form $\mathit{SRes} \longrightarrow \mathit{SRes}$ as denotations of statements and blocks of statements. *SRes* abbreviates *statement result* and is defined in Figure 2. It is a

[4] Index expression returning a boolean are excluded by only considering type-correct programs.

disjoint union consisting of seven ingredients, one for each of the possibilities a statement of GotoWhile can terminate. The components of *SRes* are used as follows:

ok The statement terminates normally. The state contained is the program state after executing the statement.

break The statement terminates with a break abnormality because it or one of the preceding statements was a `break`. The state contained is the state immediately before the `break` with which the execution should be continued at the end of the loop or the switch.

case The case abnormality is used in switch statements to skip statements between the `switch` and the matching case. The case abnormality contains a state (for continuation) and an integer (to find the right case label).

default The default abnormality is used to jump to the default case in switch statements.

goto The goto abnormality is used for goto jumps. Besides a state it carries a string identifying the target label.

fail A statement terminates with *fail* if it is not type correct or if an array-bound error occurred. It is not possible to recover from such errors, thus *fail* does not carry a state.

hang The semantics of a statement yields *hang* if the statement does not terminate because of a while loop or because of backward jumps. One cannot recover from *hang*.

I call elements of *SRes complex states* in contrast to the states in *State*. The complex states of the form $ok(-)$ are called *normal*. The other ones are *abnormal* complex states or *abnormalities*. Abnormalities different from *hang* and *fail* are converted to normal at certain places in the semantics. The phrase *to catch an abnormality* refers to this conversion.

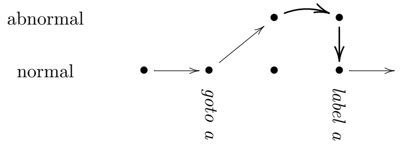
In the defining equations in Figure 3 I use *pattern matching*, indicated by bars | and maps $\mapsto$, as it is customary in functional programming: The patterns are checked from top to bottom and the one with the first match takes effect. Further, if a pattern like $ok(s)$ matches then the variable s is bound to the successor state carried by *ok*. Consider for instance the semantics of variable assignment ($v = expr$). If the argument is of the form $ok(s)$ then the result is determined by the case distinction. That is, the successor state s contained in the argument is used to determine the value of *expr*. If this value is of the form $ok(i)$, that is, if *expr* terminates normally with result i, then also the variable assignment terminates normally. In this case the successor state is obtained from s by modifying the mapping of the variable v.

If the expression *expr* does not terminate normally, then the whole assignment terminates with fail. Finally, if the *SRes* argument of the assignment is not of the form $ok(s)$, then the catch-all phrase $x \mapsto x$ takes effect and the argument is passed on as the result of the assignment. So the assignment takes only effect if started in a normal complex state.

The semantics of array assignment is a bit more complex but otherwise very similar to variable assignment. It first evaluates the two expressions and if both yield a value then the utility function *upd_array* is used. This function yields a normal result if in the state s the variable a is mapped to an array and if the index i is within the bounds.

The semantics of `break`, `goto` and `new_array` is obvious: `break` and `goto` yield their respective abnormalities if entered normally. The `new_array` statement evaluates the expressions and assigns a freshly initialised array to the variable av if everything goes well.

The semantics of sequential composition is simply function composition.[5] The interplay of statement composition and the abnormalities can be depicted as follows:



Execution starts in a normal state and therefore every statement has an effect on the program state. Then a goto-statement yields a goto-abnormality that captures the current state and the target label a. The abnormality is passed around via composition, however no assignment takes effect, so the state captured in the abnormality (if any) remains unchanged. The semantics of the label (see Subsection 3.4 below) finally turns the abnormality to normal again and the execution commences. Compound statements (like `if`, see the next subsection) take special care to pass the abnormalities around so that even a deeply nested label can catch it.

In defining the semantics of complex statements it sometimes helps to view an expression as a statement (discarding the result). This conversion is denoted with $[\![-]\!]^{\#}$ and it corresponds to the Kleisli extension if *SRes* is viewed as a monad. As expressions do not have side effects such expression statements have only an effect if the argument state is normal and the expression fails for that state.

3.3 `If` and `switch`

The semantics of `if` and `switch` in Figure 4 is slightly more complicated than naively expected. The complications arise because of the possibility to jump into the middle of an else-clause or into the body of a switch-statement or even to jump from the then-clause into the else-clause in one if-statement.

If we enter an if-statement normally we evaluate the condition and if it yields *true* the then-clause st_1 is executed. Now there is a case distinction: If the then-clause terminates with an abnormality (caused by a goto-statement, for instance) we have to execute the else-clause as well, because the jump target might be in there. The remaining cases in the semantics of the if statement are obvious now. Note that if the if-statement is entered with an abnormality we have to pass it through both the then- and the else-clause.

The semantics of switch follows the same theme. First, if the switch is entered with either a break, default, or case abnormality then this abnormality must belong to a switch statement in the environment (or a loop in the environment for the break abnormality). Therefore, in this case, the switch statement is skipped altogether.

In case the switch is entered normally we check if the value of the expression ix is matched by some case-label in the body st of the switch. If it is, we enter the body with a case abnormality, otherwise with a default abnormality. The distinction between case- and default-abnormalities is necessary because the default label might stand before some case labels and the execution might fall through from the default case into the other cases. If the body st terminates with a break or default abnormality it is converted to normal. Note that st cannot terminate with a

[5] I only deal with abstract syntax here. The semantics of a block of n statements would be the composition (of the semantics) of the n statements.

$$[\![\texttt{if}(bx)\ st_1\ \texttt{else}\ st_2]\!] = \left| \begin{array}{l} ok(s) \mapsto \begin{cases} ok(s') & \text{if } [\![bx]\!](s) = ok(true) \ \wedge \\ & \quad [\![st_1]\!](ok\ s) = ok(s') \\ [\![st_2]\!] \circ [\![st_1]\!](ok\ s) & \text{if } [\![bx]\!](s) = ok(true) \ \wedge \\ & \quad [\![st_1]\!](ok\ s) \neq ok(-) \\ [\![st_2]\!](ok\ s) & \text{if } [\![bx]\!](s) = ok(false) \\ fail & \text{otherwise} \end{cases} \\ x \quad \mapsto [\![st_2]\!] \circ [\![st_1]\!](x) \end{array} \right.$$

$$labels \ : \ labeled\text{-}statement \longrightarrow \mathcal{P}(\mathbb{Z})$$

$$labels(st) = \text{collect all case labels in } st \text{ not covered by an inner switch}$$

$$sw(bo, x) \ : \ (SRes \Rightarrow SRes) \times SRes \longrightarrow SRes$$

$$sw(bo, x) = \begin{cases} ok(s) & \text{if } bo(x) = break(s) \\ ok(s) & \text{if } bo(x) = default(s) \\ bo(x) & \text{otherwise} \end{cases}$$

$$[\![\texttt{switch}(ix)\ st]\!] = \left| \begin{array}{l} break(s) \mapsto break(s) \\ default(s) \mapsto default(s) \\ case(s, i) \mapsto case(s, i) \\ ok(s) \mapsto \begin{cases} sw([\![st]\!], case(s, i)) & \text{if } [\![ix]\!] = ok(i) \ \wedge \\ & \quad i \in labels(st) \\ sw([\![st]\!], default(s)) & \text{if } [\![ix]\!] = ok(i) \ \wedge \\ & \quad i \notin labels(st) \\ fail & \text{otherwise} \end{cases} \\ x \mapsto \begin{cases} ok(s) & \text{if } [\![st]\!](x) = break(s) \\ [\![st]\!](x) & \text{otherwise} \end{cases} \end{array} \right.$$

Fig. 4. Semantics of statements, part II: *if* and *switch*

break abnormality, because we check the presence of a matching case. It can terminate with a default abnormality if no default label is present in st.

The remaining last pattern in the semantics of `switch` deals with the abnormalities goto, fail and hang. We do not have to care about *fail* and *hang*, because they are never caught. However, a goto-abnormality must be passed into st and if it is caught and a break statement is executed, then the resulting break abnormality must be converted to normal.

3.4 The semantics of labeled statements

All labels get their own semantics as functions $SRes \longrightarrow SRes$, see Figure 5. These functions are simply inserted into the sequential composition (of the denotations) of the statements that form a block. The semantics of a single label is the obvious: It turns the corresponding abnormalities to normal and leaves everything else untouched.

$$\llbracket \textit{labeled-statement} \rrbracket \quad : \quad \mathit{SRes} \longrightarrow \mathit{SRes}$$

$$\llbracket \mathit{label} : \mathit{statement} \rrbracket \quad = \quad \llbracket \mathit{label} : \rrbracket \circ \llbracket \mathit{statement} \rrbracket$$

$$\llbracket \texttt{case}\ i : \rrbracket \quad = \quad \left| \begin{array}{lcl} \mathit{case}(s,i) & \mapsto & \mathit{ok}(s) \\ x & \mapsto & x \end{array} \right.$$

$$\llbracket \texttt{default} : \rrbracket \quad = \quad \left| \begin{array}{lcl} \mathit{default}(s) & \mapsto & \mathit{ok}(s) \\ x & \mapsto & x \end{array} \right.$$

$$\llbracket l : \rrbracket \quad = \quad \left| \begin{array}{lcl} \mathit{goto}(s,l) & \mapsto & \mathit{ok}(s) \\ x & \mapsto & x \end{array} \right.$$

Fig. 5. Semantics of labeled statements

Note that the semantics discussed so far correctly handles switch statements and all forward jumps. Backward jumps, which might create nonterminating loops, are treated in Subsection 3.6.

3.5 While loops

Traditionally, while loops get their denotational semantics via a least-fixed-point construction. This approach requires a complete partial ordering and appropriate monotone and continuous functions. Here I extend Huismann and Jacobs semantics for while loops such that it correctly handles jumps into and out of the loop. The approach of Huismann and Jacobs is considerably simpler, because it does not depend on partial orders. The semantics of Huismann and Jacobs coincides with the traditional approach, see [11, Proposition 7]. A similar proposition does also hold for the semantics of GotoWhile.

The semantics of the while loop needs a number of utility functions, see Figure 6. The first one, *iter*, iterates the semantics of the conditions *co* and the body *bo* n times starting in state x. The iteration stops early if the condition fails or evaluates to false. The result of *iter* consists of the final complex state and a boolean. The boolean is *false* if the condition failed or evaluated to false at some stage. An argument of n to *iter* does n iterations of the body. For 0 it returns the argument state x together with *true*, indicating that the condition always evaluated to *true* so far.

The function *term*?, to be applied to the result of *iter*, tells if the iteration reached a termination point of the while loop. A termination point has been reached if the body of the while loop terminates with an abnormality or if the condition fails or yields *false*. Note that a while loop might have several (different) termination points because an abnormality at iteration n might be caught in the next iteration.

The (noncomputable) function *hangs*? decides if the while loop terminates or hangs (i.e., loops forever). The loop hangs, if for every $n > 0$ the result of *iter* is not a termination point. Note how the whole construction ensures that the semantics of the loop body is evaluated at least once, if one enters the loop in an abnormal state.

The semantics of while repels break abnormalities the same way switch does. The semantics yields *hang* if the loop does not terminate. Otherwise the body is iterated until the first

$$\mathit{iter} \quad : \quad (\mathit{State} \Rightarrow \mathit{ExRes}_{\mathbb{B}}) \times (\mathit{SRes} \Rightarrow \mathit{SRes}) \times \mathit{SRes} \longrightarrow \mathbb{N} \longrightarrow \mathit{SRes} \times \mathbb{B}$$

$$\mathit{iter}(\mathit{co},\mathit{bo},x) \quad = \quad \left| \begin{array}{l} 0 \ \mapsto \ (x, \mathit{true}) \\ n \ \mapsto \begin{cases} (x, \mathit{false}) & \text{if } x = \mathit{ok}(s) \ \wedge \ \mathit{co}(s) = \mathit{ok}(\mathit{false}) \\ (\mathit{fail}, \mathit{false}) & \text{if } x = \mathit{ok}(s) \ \wedge \ \mathit{co}(s) = \mathit{fail} \\ \mathit{iter}(\mathit{co},\mathit{bo},\mathit{bo}(x))\,(n-1) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\mathit{term?} \quad : \quad \mathit{SRes} \times \mathbb{B} \longrightarrow \mathbb{B}$$

$$\mathit{term?} \quad = \quad \left| \begin{array}{lcl} (\mathit{ok}(s), \mathit{false}) & \mapsto & \mathit{true} \\ (\mathit{ok}(s), \mathit{true}) & \mapsto & \mathit{false} \\ (-,-) & \mapsto & \mathit{true} \end{array} \right.$$

$$\mathit{hangs?} \quad : \quad (\mathit{State} \Rightarrow \mathit{ExRes}_{\mathbb{B}}) \times (\mathit{SRes} \Rightarrow \mathit{SRes}) \times \mathit{SRes} \longrightarrow \mathbb{B}$$

$$\mathit{hangs?}(\mathit{co},\mathit{bo},x) \quad = \quad \begin{cases} \mathit{true} & \text{if } \forall i \in \mathbb{N}^+ \,.\, \mathit{term?}(\mathit{iter}(\mathit{co},\mathit{bo},i)) = \mathit{false} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{while} \quad : \quad (\mathit{State} \Rightarrow \mathit{ExRes}_{\mathbb{B}}) \times (\mathit{SRes} \Rightarrow \mathit{SRes}) \times \mathit{SRes} \longrightarrow \mathit{SRes}$$

$$\mathit{while}(\mathit{co},\mathit{bo},x) \quad = \quad \begin{cases} \mathit{hang} & \text{if } \mathit{hangs?}(\mathit{co},\mathit{bo},x) = \mathit{true} \\ \pi_1(\mathit{iter}(\mathit{co},\mathit{bo},x)\ (n)) & \text{otherwise} \\ \quad \text{where } n = \min\{i \in \mathbb{N}^+ \mid \mathit{term?}(\mathit{iter}(\mathit{co},\mathit{bo},x)\ (i))\} & \end{cases}$$

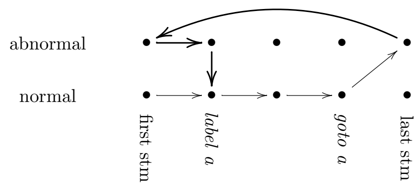
$$\llbracket \texttt{while}(\mathit{bx})\ \mathit{st} \rrbracket \quad = \quad \left| \begin{array}{lcl} \mathit{break}(s) & \mapsto & \mathit{break}(s) \\ x & \mapsto & \begin{cases} \mathit{ok}(s) & \text{if } \mathit{while}(\llbracket \mathit{bx} \rrbracket, \llbracket \mathit{st} \rrbracket, x) = \mathit{break}(s) \\ \mathit{while}(\llbracket \mathit{bx} \rrbracket, \llbracket \mathit{st} \rrbracket, x) & \text{otherwise} \end{cases} \end{array} \right.$$

Fig. 6. Semantics of the `while` statements

termination point (discarding 0 iterations). If the iteration yields a break abnormality it is converted to normal.

3.6 Whole programs and backward jumps

To finish the semantics of GotoWhile I only have to discuss backward jumps. The following picture explains the solution:



One starts a program with a normal state (so start to read the picture following the thin arrows in the bottom line). Eventually towards the end a goto-statement is executed and therefore the whole sequence of statements terminates with an goto-abnormality. The solution is to simply

$$
\begin{array}{rcl}
g\_iter & : & (SRes \Rightarrow SRes) \longrightarrow \mathbb{N} \longrightarrow SRes \longrightarrow SRes \\
g\_iter\ bo & = & \left| \begin{array}{l} 0 \mapsto bo \\ n \mapsto bo \circ (g\_iter\ bo\ (n-1)) \end{array} \right. \\
g\_term? & : & SRes \longrightarrow \mathbb{B} \\
g\_term? & = & \left| \begin{array}{ll} goto(s) & \mapsto false \\ x & \mapsto true \end{array} \right. \\
g\_hangs? & : & (SRes \Rightarrow SRes) \times SRes \longrightarrow \mathbb{B} \\
g\_hangs?(bo, x) & = & \begin{cases} true & \text{if } \forall i \in \mathbb{N} \,.\, g\_term?(g\_iter\ bo\ i\ x) = false \\ false & \text{otherwise} \end{cases} \\
[\![prog]\!] & : & State \longrightarrow StRes \\
[\![prog]\!](s) & = & \begin{cases} hang & \text{if } g\_hangs?([\![prog]\!], ok(s)) = true \\ g\_iter\ [\![prog]\!]\ n\ (ok\ s) & \text{otherwise} \\ \quad \text{where } n = \min\{i \in \mathbb{N} \mid g\_term?(g\_iter\ [\![prog]\!]\ i\ (ok\ s))\} & \end{cases}
\end{array}
$$

Fig. 7. Semantics for complete programs and `goto` loops

repeat all statements. This time we enter the first statement with an goto-abnormality, so everything is skipped until the label. Then execution commences normally.

Backward jumps can be used to construct loops so we need a semantics similar to that of while loops for complete programs. However, these goto-loops are much simpler because there is no condition and also the termination points can be recognised more easily.

Figure 7 contains the semantic definitions. The iteration function is called *g_iter* this time and it simply composes *bo*, the semantics of all statements of the program, $n+1$ times with itself (for an argument n). A termination point (as described by *g_term?*) has been reached if *bo does not* terminates with an goto-abnormality. The semantics of a complete program is a function $State \longrightarrow Res$. It maps a state s to *hang* if the program viewed as one labeled statement contains a goto-loop that does not terminate. Otherwise the labeled statement is iterated an appropriate number of times.[6]

The ideas developed here can be directly applied to C or C++. This is currently done in the VFiasco project [12]. In C++ (and also in C) one cannot jump across function boundaries. Therefore the construction of goto-loops in Figure 7 must be applied to every function body of the C++ source (and not to the whole program). There are further restrictions on jumps in C++. For instance one cannot not jump over a declaration of data with nontrivial constructors. The semantics of C++ can reflect this restriction with functions that turn goto-abnormalities into *fail* inserted at appropriate places. For a complete semantics of C++ one also need a semantics for data types that can deal with pointers to data and type casts of data and of pointers [8].

[6] The overloading of $[\![-]\!]$ leads to an ambiguity in Figure 7. The $[\![prog]\!]$ one the right hand side refers to the semantics of *prog* as a labeled statement.

4 A Hoare rule for while loops in GotoWhile

The semantics presented in the preceding section is mostly operational in the following sense: For all statements except `while` and backward jumps the semantics is defined as state transformation that can be executed symbolically. This might get complicated on paper but works remarkably well in a theorem prover, which keeps track of all the variable updates in the program state. However, the semantics of loops, especially the minimum construction, is difficult to handle in a theorem prover. The difficulty only applies to terminating while loops. With the given semantics nontermination can be proved with a simple induction.

This section presents a total correctness theorem for while loops in the spirit of Hoare-logic. Similar theorems can be found in [13, 1]. The version in this section differs in the following points: It correctly handles jumps into and out of the loop. It treats the break abnormality, which is not contained in Java^{light} and thus not treated in [13]. Further I present only one theorem, treating every possible abnormality, instead of one theorem for each abnormality as in [1].

For the definition of the variant and the invariant I need to check whether the loop is terminated by the condition in a normal way (i.e., the condition is evaluated and returns *false*). This check is captured in the following definition:

$$
n\text{-}term(cx, x) \stackrel{\text{def}}{=} \begin{cases} true & \text{if } x = ok(s) \ \wedge\ [\![cx]\!](s) = ok(false) \\ false & \text{otherwise} \end{cases}
$$

A while loop stops the iteration precisely if either $n\text{-}term(cx, x) = true$ or if $[\![st]\!] \circ [\![bx]\!]^{\#}(x)$ is an abnormal state.

Definition 1 (Goto-While Invariant). *A pair of predicates $P, Q \subseteq SRes$ is a* goto-while invariant *with respect to the condition expression bx and the loop body st if for all $x \in SRes$*

$$
P(x) \quad implies \quad \begin{cases} Q(x) & if\ \ n\text{-}term(cx, x) \\ P(ok(s)) & else\ if\ [\![st]\!] \circ [\![co]\!]^{\#}(x) = ok(s) \\ Q([\![st]\!] \circ [\![co]\!]^{\#}(x)) & otherwise \end{cases}
$$

where the "else if" case is effective only if $n\text{-}term(cx, x) = false$.

The preceding invariant definition may seem odd because it applies to two predicates. The problem here is to distinguish the termination points from those states in which the iteration continues. The predicate P must contain at least all those states in which the iterations continues. Therefore we assume P in the above implication and require P only in the case in which the iteration continues (namely the second). The predicate Q must contain at least those states in which the iteration stops. Therefore we *do not* assume Q but require it in the first and the third case. The theorem below is of the form $P(x) \Longrightarrow Q([\![\texttt{while}...]\!](x))$. That is, starting in a state x in which the iteration continues the loop reaches a state in which iteration has stopped. Note that P and Q are usually *not* disjoint. State in which you enter the while loop and leave it immediately have to be in P *and* Q.

In classical Hoare logic the condition (which is false on termination points and true otherwise) distinguishes the states in which the iteration stops from those states in which it continues. However, in GotoWhile one can leave a loop via `break` in a state in which the condition is true. Note also that the distinction of normal and abnormal states is not sufficient: If one jumps into a loop the state before the first iteration is an abnormal one.

The following variant definition is completely standard except that the variant is only required to decrease for states contained in an invariant. The possibility to exclude states is convenient if the program state is dynamically typed.

Definition 2 (Goto-While Variant). *Let* $(O, <)$ *be a well-founded order and let* f *be a function* $SRes \longrightarrow O$ *and consider a condition* cx *and a body* st *of a while loop with an invariant* (P, Q). *The function* f *is a* goto-while variant *with respect to* cx, st *and* (P, Q) *if the following holds. Let* $x \in SRes$ *with* $P(x)$ *and* $n\text{-}term(cx, x) = false$ *and such that* $[\![st]\!] \circ [\![bx]\!]^{\#}(x)$ *is a normal state. In this case it is required that* $f([\![st]\!] \circ [\![bx]\!]^{\#}(x)) < f(x)$.

To express the main theorem I need the *catch-break transformation* of a predicate, denoted with $Q^{\dagger}$ for a predicate $Q \subseteq SRes$. The problem is that the invariant definition allows only to conclude that the predicate Q holds on the result of the iteration function *iter*. However, in this result the semantics of while loops transforms a potential break abnormality to normal. Formally the catch-break transformation $Q^{\dagger}$ is the direct image of Q under *catching breaks*. In basic terms:

$$Q^{\dagger} \stackrel{\text{def}}{=} \{\, x \in SRes \mid (x \neq break(-) \wedge Q(x)) \;\vee\; \exists s \in State \,.\, x = ok(s) \wedge Q(break\ s) \,\}$$

Theorem 3 (Total Goto-While Correctness). *Consider a while loop with condition* cx *and body* st. *Let* (P, Q) *be an invariant and* f *be a variant for* cx *and* st. *Then for all* $x \in SRes$ *with* $x \neq break(-)$ *the following holds:*

$$P(x) \quad implies \quad Q^{\dagger}\big([\![\mathtt{while}(co)st]\!](x)\big)$$

The theorem has been proved in PVS, see Lemma `hoare_gwhile` in theory `HoareGWhile`.

Proof (Sketch). First we exploit the variant to show that the loop terminates and that thus the set of termination points is not empty. Then one uses induction on the minimal number of iterations necessary to reach a termination point to show that Q holds on the termination point. □

5 Duff's device verified

Duff's device [14] is a trick to express loop unrolling directly in C or C++ without extra code to treat the leftover partial loop. Figure 8 shows it in the syntax of GotoWhile.[7] The trick is to use a switch statement where all but one of the case labels are in the middle of a while loop. Further all cases fall through to the end of the while loop. Despite the impression it makes Duff's device is legal C and C++ code (however, it is not valid in Java).

Theorem 4 (Total Correctness of Duff's device). *Let* $s \in State$ *such that dest and src are mapped to arrays in* s *and that count is mapped to a nonnegative integer. Assume further that both arrays contain at least* `count` *elements. In this case the program Duff's device in Figure 8 started in* s *terminates normally in a state* s' *where* s *and* s' *differ* only *in the following: In* s'

- `rounds` *has the value* 0
- `i` *has the value of* `count` *in* s
- *any element in* `dest` *with an index less than* `count` *has been changed to be equal to the element in* `src` *with the same index.*

[7] Originally Tom Duff needed to copy an array to an memory mapped device, see [14]. Therefore, in his version, the destination address was constant (and, of course, he used the usual C idioms: pointers and post increment of pointers).


```
rounds = count / 8;
i = 0;
switch(count % 8){
  case 0: while( !(rounds == 0) ) {
            rounds = rounds - 1;
            dest[i] = src[i]; i = i + 1;
  case 7:   dest[i] = src[i]; i = i + 1;
  case 6:   dest[i] = src[i]; i = i + 1;
  case 5:   dest[i] = src[i]; i = i + 1;
  case 4:   dest[i] = src[i]; i = i + 1;
  case 3:   dest[i] = src[i]; i = i + 1;
  case 2:   dest[i] = src[i]; i = i + 1;
  case 1:   dest[i] = src[i]; i = i + 1;
          };
}
```

The above code snippet assumes that initially `count` is an integer variable greater or equal to zero and that `dest` and `src` are array variables containing at least `count` elements.

Fig. 8. Duff's device in GotoWhile

Again, this theorem has been proved in PVS, see lemma `duff_total` in theory `TestPrograms3`.

Proof (sketch). Starting from state s we use the semantics for the first two assignments and arrive at a state s' in which `i` is set to 0 and `rounds` is set to the integer division of `count` and 8. Now we do a case distinction. Either `count` is a multiple of 8 or not. In the first case we enter the while loop with a case abnormality, in the second we enter it in a normal state. In both cases we use the following instantiation of Theorem 3 to finish the proof:

- the invariant predicates P and Q contain states that are identical to s' apart from
  - for Q the state must be a normal one, for P it can as well be a case abnormality
  - for P `rounds` holds a nonnegative value less than `count`/8, for Q it is 0
  - elements with index below the value of `i` have been copied from `src` to `dest`
  - the value of `i` is equal to $8 * (\mathtt{count}/8 - ro) + rem$, where ro is 0 for Q and equal to `rounds` for P and where $\mathtt{rem} = \mathtt{count}\ \%\ 8$ in normal states and $\mathtt{rem} = 0$ for case abnormalities
- the natural numbers are used as well-founded ordering
- the variant maps a state to $\mathtt{rounds} + x$ where $x = 1$ for case abnormalities and $x = 0$ for normal states

□

6 Conclusion

This paper presents a compositional denotational semantics for a fragment of C++ containing assignments, goto jumps and switch and while statements. The underlying mathematics is remarkably simple. Neither partial orders nor domains are required. The main idea is to use a big disjoint union as program state, where the different injections describe the different execution modes of the program: normal execution, jumping, permanent failure, and so on. The semantics presented here is a generalisation of Huismann and Jacobs semantics for abrupt termination [1].

The semantics has been used to verify Duff's device, a program which contains jumps into the middle of a while loop. The verification uses a total correctness theorem inspired by Hoare logic. The theorem correctly handles jumps into and out of a loop.

The main idea of this paper, to use a complex state encapsulating different execution modes, can also be applied to operational semantics, especially to natural semantics, to obtain a simple and elegant treatment of jumps.

References

1. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In Maibaum, T., ed.: Fundamental Approaches to Software Engineering. Volume 1783 of Lecture Notes in Computer Science., Springer, Berlin (2000) 284–303
2. Hohmuth, M., Härtig, H.: Pragmatic nonblocking synchronization for real-time systems. In: USENIX Annual Technical Conference, Boston, MA (2001)
3. Gellerich, W., Kosiol, M., Ploedereder, E.: Where does GOTO go to? In: Reliable Software Technologies – Ada-Europe 1996. Volume 1088 of Lecture Notes in Computer Science., Springer (1996) 385–395
4. Tennent, R.D.: Denotational semantics. In S. Abramsky, D. M. Gabby, T.S.E.M., ed.: Handbook of Logic in Computer Science. Volume 3. Oxford Science Publications (1994) 169–322
5. Mosses, P.D.: Denotational semantics. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, volume B. Elsevier Science Publishers, Amsterdam (1990) 575–631
6. Tews, H.: Programmverifikation und –spezifikation mit Coalgebren. German lecture script. (2004) Available at wwwtcs.inf.tu-dresden.de/∼tews/VeriLecture/.
7. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T., eds.: Computer Aided Verification. Volume 1102 of Lecture Notes in Computer Science., Springer, Berlin (1996) 411–414
8. Hohmuth, M., Tews, H.: The semantics of C++ data types: Towards verifying low-level system components. In Basin, D., Wolff, B., eds.: TPHOLs 2003, Emerging Trends Proceedings. (2003) 127–144 Technical Report No. 187 Institut für Informatik Universität Freiburg.
9. International Organization for Standardization: ISO/IEC 14882:1998: Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (1998)
10. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley Series in Data Communications and Networking for Computer Progammers. John Wiley and Sons, Chichester (1992)
11. Jacobs, B., Poll, E.: Coalgebras and monads in the semantics of java. Theoretical Computer Science **291** (2003) 329–349
12. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD–FI02–03–März 2002, Dresden University of Technology (2002) Available from URL: `os.inf.tu-dresden.de/vfiasco/`.
13. von Oheimb, D.: Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müller, P., Poetzsch-Heffter, A., eds.: Formal Techniques for Java Programs, Technical Report 269, 5/2000, Fernuniversität Hagen. (2000) Available at www.informatik.fernuni-hagen.de/pi5/publications.html.
14. Duff, T.: Tom Duff on Duff's Device (2004) available at www.lysator.liu.se/c/duffs-device.html.