The National Institute of Aerospace / Galois Inc. / NASA LaRC

An Introduction to Copilot

Macallan Cruff
mcruff@andrew.cmu.edu

Nis N. Wegmann
niswegmann@gmail.com

Lee Pike
leepike@galois.com

Chris Hathhorn
hathhorn@gmail.com

Sebastian Niller
sebastian.niller@gmail.com

Lauren Pick
lmpick@zoho.com

Georges-Axel Jaloyan
georges-axel.jaloyan@ens.fr

Hampton, Virginia, United States, December 3, 2018

Abstract

This document contains a tutorial on Copilot and its accompanying tools. We do not attempt to give a complete, formal description of Copilot (references are provided in the bibliography), rather we aim at demonstrating the fundamental concepts of the language by using idiomatic expositions and examples.

Contents

This research is supported by NASA Contract NNL08AD13T from the Aviation Safety Program Office.

Acknowledgement

The authors are grateful for NASA Contract NNL08AD13T to Galois Inc. and the National Institute of Aerospace, which partially supported this work. Thanks to Lars Kuhtz, Benedetto Di Vito, Robin Morisset, Kaveh Darafsheh, Alwyn Goodloe.

1 Introduction

Neither formal verification nor testing can ensure system reliability. Nearly 20 years ago, Butler and Finelli showed that testing alone cannot verify the reliability of ultra-critical software [BF93]. *Runtime verification* (RV) [GP10], where monitors detect and respond to property violations at runtime, has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. Technically speaking, a RV monitor takes a logical specification ϕ and execution trace τ of state information of the system under observation (SUO) and decides whether τ satisfies ϕ. The *Simplex Architecture* [Sha01] provides a model architectural pattern for RV, where a monitor checks that the executing SUO satisfies a specification and, if the property is violated, the RV system will switch control to a more conservative component that can be assured using conventional means that *steers* the system into a safe state. *High-assurance* RV provides an assured level of safety even when the SUO itself cannot be verified by conventional means.

Copilot is a RV framework for specifying and generating monitors for C programs that is embedded into the functional programming language Haskell [Jon02]. A working knowledge of Haskell is necessary to use Copilot effectively; a variety of books and free web resources introduce Haskell. Copilot uses Haskell language extensions specific to the Glasgow Haskell Compiler (GHC).

1.1 Target Application Domain

Copilot is a domain-specific language tailored to programming *runtime monitors* for *hard real-time*, *distributed*, *reactive systems*. Briefly, a runtime monitor is program that runs concurrently with a target program with the sole purpose of assuring that the target program behaves in accordance with a pre-established specification. Copilot is a language for writing such specifications.

A reactive system is a system that responds continuously to its environment. All data to and from a reactive system are communicated progressively during execution. Reactive systems differ from transformational systems which transform data in a single pass and then terminate, as for example compilers and numerical computation software.

A hard real-time system is a system that has a statically bounded execution time and memory usage. Typically, hard real-time systems are used in mission-critical software, such as avionics, medical equipment, and nuclear power plants; hence, occasional dropouts in the response time or crashes are not tolerated.

A distributed system is a system which is layered out on multiple pieces of hardware. The distributed systems we consider are all synchronized, i.e., all components agree on a shared global clock.

1.2 Installation

Before downloading the copilot source code, you must first install an up-to-date version of GHC. (The minimal required version is 7.10.1) The easiest way to do this is to download and install the Haskell Platform, which is freely distributed from here:

`http://hackage.haskell.org/platform`

Because Copilot compiles to C code, you must also install a C compiler such as gcc (`https://gcc.gnu.org/install/`). **XCODE?** After having installed the Haskell Platform and C compiler, Copilot can be downloaded and installed in the following two ways:

- **Using Cabal:** Copilot can be downloaded and installed by executing the following command:

  ```
  > cabal install copilot
  ```

  This should, if everything goes well, install Copilot on your system.

- **Using Git:**

  ```
  git clone https://github.com/Copilot-Language/Copilot.git
  git submodule update --init
  make test
  ```

To use the language, your Haskell module should contain the following import:

```
import Language.Copilot
```

To use the back-ends, import them, respectively:

```
import Copilot.Compile.C99
import Copilot.Compile.SBV
```

If you need to use functions defined in the Prelude that are redefined by Copilot (e.g., arithmetic operators), import the Prelude as qualified:

```
import qualified Prelude as P
```

1.3 Structure

Copilot is distributed through a series of packages at Hackage:

- copilot-language: Contains the language front-end.
- copilot-theorem: Contains some extensions to the language for proving properties about copilot programs using various SMT solvers and model checkers.
- copilot-core: Contains an intermediate representation for Copilot programs (shared by all back-ends).
- copilot-c99: A back-end for Copilot targeting C99 (based on Atom, `http://hackage.haskell.org/package/atom`). **Not updated anymore, might be deprecated soon.**
- copilot-sbv: A back-end for Copilot targeting C99 (based on SBV, `http://hackage.haskell.org/package/sbv`).
- copilot-libraries: A set of utility functions for Copilot, including a clock-library, a linear temporal logic framework, a voting library, and a regular expression framework.
- copilot-cbmc: A driver for proving the correspondence between code generated by the copilot-c99 and copilot-sbv back-ends.

Many of the examples in this paper can be found at `https://github.com/Copilot-Language/Copilot/tree/copilot2.0/Examples`.

1.4 Sampling

The idea of sampling representative data from a large set of data is well established in data science and engineering. For instance, in digital signal processing, a signal such as music is sampled at a high enough rate to obtain enough discrete points to represent the physical sound wave. The fidelity of the recording is dependant on the sampling rate. Sampling a state variable of an executing program is similar, but variables are rarely continuous signals so they lack the nice properties of continuity. Monitoring based on sampling state-variables has historically been disregarded as a runtime monitoring approach, for good reason: without the assumption of synchrony between the monitor and observed software, monitoring via sampling may lead to false positives and false negatives [DDE08]. For example, consider the property $(0; 1; 1)^*$, written as a regular expression, denoting the sequence of values a monitored variable may take. If the monitor samples the variable at the inappropriate time, then both false negatives (the monitor erroneously rejects the sequence of values) and false positives (the monitor erroneously accepts the sequence) are possible. For example, if the actual sequence of values is $0, 1, 1, 0, 1, 1$, then an observation of $0, 1, 1, 1, 1$ is a

false negative by skipping a value, and if the actual sequence is $0, 1, 0, 1, 1$, then an observation of $0, 1, 1, 0, 1, 1$ is a false positive by sampling a value twice.

However, in a hard real-time context, sampling is a suitable strategy. Often, the purpose of real-time programs is to deliver output signals at a predicable rate and properties of interest are generally data-flow oriented. In this context, and under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, while false positives are possible, false negatives are not. A false positive is possible, for example, if the program does not execute according to its schedule but just happens to have the expected values when sampled. If a monitor samples an unacceptable sequence of values, then either the program is in error, the monitor is in error, or they are not synchronized, all of which are faults to be reported.

Most of the popular runtime monitoring frameworks inline monitors in the observed program to avoid the aforementioned problems with sampling. However, inlining monitors changes the real-time behavior of the observed program, perhaps in unpredicable ways. Solutions that introduce such unpredictability are not a viable solution for ultra-critical hard real-time systems. In a sampling-based approach, the monitor can be integrated as a separate scheduled process during available time slices (this is made possible by generating efficient constant-time monitors). Indeed, sampling-based monitors may even be scheduled on a separate processor (albeit doing so requires additional synchronization mechanisms), ensuring time and space partitioning from the observed programs. Such an architecture may even be necessary if the monitored program is physically distributed.

When deciding whether to use Copilot to monitor systems that are not hard real-time, the user must determine if sampling is suitable to capture the errors and faults of interest in the SUO. In many cyber-physical systems, the trace being monitored is coming from sensors measuring physical data such as GPS coordinates, air speed, and actuation signals. These continuous signals do not change abruptly so as long as it is sampled at a suitable ratei, that usually must be determined experimentally, sampling is sufficient.

2 Interpreting and Compiling

The Copilot RV framework comes with both an interpreter and a compiler.

2.1 Interpreting Copilot

To use Copilot's interpreter, one must first invoke the GHC Interpreter via the Cabal sandbox. Invoking the interpreter with the `cabal repl` command starts a session that looks like:

```
cabal repl
Preprocessing library copilot-2.1.2...
GHCi, version 7.10.1: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Language.Copilot ( src/Language/Copilot.hs, interpreted )
Ok, modules loaded: Language.Copilot.
*Language.Copilot>
```

A copilot file `file` is loaded as follows:

```
Language.Copilot> :l file
```

Notice that `file` has to be located in the directory *Copilot*.

The copilot interpreter is invoked as follows:

```
Language.Copilot> interpret 10 propTempRiseShutOff
```

The first argument to the function *interpret* is the number of iterations that we want to evaluate. The third argument is the specification (of type `Spec`) that we wish to interpret.

The interpreter outputs the values of the arguments passed to the trigger, if its guard is true, and `--` otherwise. We will discuss triggers in more detail later, but for now, just know that they produce an output only when the guard function is true. For example, consider the following Copilot program:

```
spec = do
  trigger "trigger1" (even nats) [arg nats, arg $ odd nats]
  trigger "trigger2" (odd nats) [arg nats]
```

where `nats` is the stream of natural numbers, and `even` and `odd` are the guard functions that take a stream and return whether the point-wise values are even or odd, respectively. The lists at the end of the trigger represent the values the trigger will output when the guard is true. The output of

```
interpret 10 spec
```

is as follows:

```
trigger1:   trigger2:
(0,false)  --
--         (1)
```

```
(2,false)  --
--         (3)
(4,false)  --
--         (5)
(6,false)  --
--         (7)
(8,false)  --
--         (9)
```

Note that trigger1 outputs both the number and whether that number is odd, while trigger2 only outputs the number. This output reflects the arguments passed to them.

Sometimes it is convenient to observe the behavior of a stream without defining a trigger. We can do so declaring an *observer*. For example:

```
spec :: Spec
spec = observer "obs" nats
```

can be interpreted using

```
interpret 5 spec
```

as usual and yields

```
obs:
0
1
2
3
4
```

2.2 Compiling Copilot

Compiling in Copilot is straightforward. First, we pick a back-end to compile to. Currently, two back-ends are implemented, both of which generate constant-time and constant-space C code. One back-end is called *copilot-c99* and targets the Atom language,[1] originally developed by Tom Hawkins at Eaton Corp. for developing control systems (in the process of depreciation). The second back-end is called *copilot-sbv* and targets the SBV language,[2] originally developed by Levent Erkök. SBV is primarily used as an interface to SMT solvers [BSST09] and also contains a C-code generator. Both languages are open-source.

The two back-ends are installed with Copilot, and they can be imported, respectively, as

```
import Copilot.Compile.C99
```

and

[1] `http://hackage.haskell.org/package/atom`
[2] `http://hackage.haskell.org/package/sbv`

```
import Copilot.Compile.SBV
```

After importing a back-end, the interface for compiling is as follows: [3]

```
reify spec >>= compile defaultParams
```

(The compile function takes a parameter to rename the generated C files; `defaultParams` is the default, in which there is no renaming.)

The compiler then generates several files in a separate-subfolder, some of which are called:

- "driver.c" —
- "copilot.h" —

The file named "copilot.h" contains prototypes for all external variables, functions, and arrays, and contains a prototype for the "step"-function, which evaluates a single iteration.

Another example can be found in Copilot/Examples/Examples2.hs. At the top of the file we alias `Copilot.Compile.SBV` as `S` for simplicity. Observe line 75:

```
reify spec >>= S.compile (S.Params { S.prefix = Just "secondexamplespec"} )
```

Which, when executed, prefixes "secondexamplespec" to all of the "standard" file names and the step function.

3 Language

Copilot is embedded into the functional programming language Haskell [Jon02], and a working knowledge of Haskell is necessary to use Copilot effectively. Copilot is a pure declarative language; i.e., expressions are free of side-effects and are referentially transparent. A program written in Copilot, which from now on will be referred to as a *specification*, has a cyclic behavior, where each cycle consists of a fixed series of steps:

- Sample external variables, arrays, and functions.
- Update internal variables.
- Fire external triggers. (In case the specification is violated.)
- Update observers (for debugging purpose).

[3] Two explanations are in order: (1) `reify` allows sharing in the expressions to be compiled [Gil09a], and `>>=` is a higher-order operator that takes the result of reification and "feeds" it to the compile function.

We refer to a single cycle as an *iteration* or a *step*.

All transformation of data in Copilot is propagated through streams. A stream is an infinite, ordered sequence of values which must conform to the same type. E.g., we have the stream of Fibonacci numbers:

$$s_{fib} = \{0, 1, 1, 2, 3, 5, 8, 13, 21, \dots\}$$

We denote the nth value of the stream s as $s(n)$, and the first value in a sequence s as $s(0)$. For example, for s_{fib} we have that $s_{fib}(0) = 0$, $s_{fib}(1) = 1$, $s_{fib}(2) = 1$, and so forth.

Constants as well as arithmetic, boolean, and relational operators are lifted to work pointwise on streams:

```
x :: Stream Int32
x = 5 + 5

y :: Stream Int32
y = x * x

z :: Stream Bool
z = x == 10 && y < 200
```

Here the streams `x`, `y`, and `z` are simply *constant streams*:

$$\texttt{x} \rightsquigarrow \{10, 10, 10, \dots\},\ \texttt{y} \rightsquigarrow \{100, 100, 100, \dots\},\ \texttt{z} \rightsquigarrow \{\mathrm{T},\ \mathrm{T},\ \mathrm{T},\ \dots\}$$

Two types of *temporal* operators are provided, one for delaying streams and one for looking into the future of streams:

```
(++) :: [a] -> Stream a -> Stream a
drop :: Int -> Stream a -> Stream a
```

Here `xs ++ s` prepends the list `xs` at the front of the stream `s`. For example the stream `w` defined as follows, given our previous definition of `x`:

```
w = [5,6,7] ++ x
```

evaluates to the sequence $\texttt{w} \rightsquigarrow \{5, 6, 7, 10, 10, 10, \dots\}$. The expression `drop k s` skips the first `k` values of the stream `s`, returning the remainder of the stream. For example we can skip the first two values of `w`:

```
u = drop 2 w
```

which yields the sequence $\texttt{u} \rightsquigarrow \{7, 10, 10, 10, \dots\}$.

3.1 Streams as Lazy-Lists

A key design choice in Copilot is that streams should mimic *lazy lists*. In Haskell, the lazy-list of natural numbers can be programmed like this:

```
nats_ll :: [Int32]
nats_ll = [0] ++ zipWith (+) (repeat 1) nats_ll
```

As both constants and arithmetic operators are lifted to work pointwise on streams in Copilot, there is no need for `zipWith` and `repeat` when specifying the stream of natural numbers:

```
nats :: Stream Int32
nats = [0] ++ (1 + nats)
```

In the same manner, the lazy-list of Fibonacci numbers can be specified in Haskell as follows:

```
fib_ll :: [Int32]
fib_ll = [1, 1] ++ zipWith (+) fib_ll (drop 1 fib_ll)
```

In Copilot we simply throw away `zipWith`:

```
fib :: Stream Int32
fib = [1, 1] ++ (fib + drop 1 fib)
```

Copilot specifications must be *causal*, informally meaning that stream values cannot depend on future values. For example, the following stream definition is allowed:

```
f :: Stream Word64
f = [0,1,2] ++ f

g :: Stream Word64
g = drop 2 f
```

But if instead `g` is defined as `g = drop 4 f`, then the definition is disallowed. While an analogous stream is definable in a lazy language, we bar it in Copilot, since it requires future values of `f` to be generated before producing values for `g`. This is not possible since Copilot programs may take inputs in real-time from the environment (see Section **??**).

3.2 Functions on Streams

Given that constants and operators work pointwise on streams, we can use Haskell as a macro-language for defining functions on streams. The idea of using Haskell as a macro language is powerful since Haskell is a general-purpose higher-order functional language.

Example 1:
We define the function `even`*, which given a stream of integers returns a boolean stream which is true whenever the input stream contains an even number, as follows:*

x_i:	y_{i-1}:	y_i:
F	F	F
F	T	T
T	F	T
T	T	F

```
latch :: Stream Bool -> Stream
    Bool
latch x = y
  where
  y = if x then not z else z
  z = [False] ++ y
```

	0	1	2	3	4
x	F	T	T	F	F
y	F	T	F	F	F

Figure 1: A latch [Example 3]. The specification function is provided at the left and the implementation in copilot is provided in the middle. The right shows an example of the latch, where x is $\{F, T, T, F, F, \dots\}$ and the initial value of y (used with x_0 to find y_0 since there is no y_{-1}) is False.

```
even :: Stream Int32 -> Stream Bool
even x = x `mod` 2 == 0
```

Applying `even` *on* `nats` *(defined above) yields the sequence* $\{T, F, T, F, T, F, \dots\}$.

If a function is required to return multiple results, we simply use plain Haskell tuples:

Example 2:

We define complex multiplication as follows:

```
mul_comp
  :: (Stream Double, Stream Double)
  -> (Stream Double, Stream Double)
  -> (Stream Double, Stream Double)
(a, b) `mul_comp` (c, d) = (a * c - b * d, a * d + b * c)
```

Here `a` *and* `b` *represent the real and imaginary part of the left operand, and* `c` *and* `d` *represent the real and imaginary part of the right operand.*

3.3 Stateful Functions

In addition to pure functions, such as `even` and `mul_comp`, Copilot also facilitates *stateful* functions. A *stateful* function is a function which has an internal state, e.g. as a latch (as in electronic circuits) or a low/high-pass filter (as in a DSP).

Example 3:

We consider a simple latch, as described in [Far04], with a single input and a boolean state. A latch is a way of simulating memory in circuits by feeding back output gates as inputs. Whenever the input is true the internal state is reversed. The operational behavior and the implementation of the latch is shown in Figure 1.[4]

[4]In order to use conditionals (i.e., if-then-else) in Copilot specifications, as in Figures 1 and 2, the GHC language extension `RebindableSyntax` must be set on.

$\texttt{inc}_i$:	$\texttt{reset}_i$:	$\texttt{cnt}_i$:
F	F	$\texttt{cnt}_{i-1}$
$*$	T	0
T	F	$\texttt{cnt}_{i-1} + 1$

```
counter :: Stream Bool -> Stream Bool
        -> Stream Int32
counter inc reset = cnt
  where
  cnt = if reset then 0
          else if inc then z + 1
                 else z
  z = [0] ++ cnt
```

Figure 2: A resettable counter. The specification is provided at the left and the implementation is provided at the right.

Example 4:
We consider a resettable counter with two inputs, `inc` *and* `reset`*. The input* `inc` *increments the counter and the input* `reset` *resets the counter. The internal state of the counter,* `cnt`*, represents the value of the counter and is initially set to zero. At each cycle, i, the value of* $\texttt{cnt}_i$ *is determined as shown in the left table in Figure 2.*

3.4 Types

Copilot is a typed language, where types are enforced by the Haskell type system to ensure generated C programs are well-typed. Copilot is *strongly typed* (i.e., type-incorrect function application is not possible) and *statically typed* (i.e., type-checking is done at compile-time). The base types are Booleans, unsigned and signed words of width 8, 16, 32, and 64, floats, and doubles. All elements of a stream must belong to the same base type. These types have instances for the class `Typed a`, used to constrain Copilot programs.

We provide a `cast` operator

```
cast :: (Typed a, Typed b) => Stream a -> Stream b
```

that casts from one type to another. The cast operator is only defined for casts that do not lose information, so an unsigned word type `a` can only be cast to another unsigned type at least as large as `a` or to a signed word type strictly larger than `a`. Signed types cannot be cast to unsigned types but can be cast to signed types at least as large.

There also exists an `unsafeCast` operator which allows casting from any type to any other (except from floating point numbers to integer types):

```
unsafeCast :: (Typed a, Typed b) => Stream a -> Stream b
```

3.5 Interacting With the Target Program

All interaction with the outside world is done by sampling *external symbols* and by evoking *triggers*. External symbols are symbols that are defined outside Copilot and which reflect the visible state of the target program that we are monitoring. They include variables, arrays, and functions (with

a non-void return type). Analogously, triggers are functions that are defined outside Copilot and which are evoked when Copilot needs to report that the target program has violated a specification constraint.

External Variables. As discussed in Section 1.4, *sampling* is an approach for monitoring the state of an executing system based on sampling state-variables, while assuming synchrony between the monitor and the observed software. Copilot targets hard real-time embedded C programs so the state variables that are observed by the monitors are variables of C programs. Copilot monitors run either in the same thread or a separate thread as the system under observation and the only variables that can be observed are those that are made available through shared memory. This means local variables cannot be observed. Currently, Copilot can only observe basic C data structures and cannot handle C structs, dynamic data structures, or two dimensional arrays.

Copilot has both an interpreter and a compiler.The compiler must be used to monitor an executing program. The Copilot reification process generates a C monitor from a Copilot specification. The variables that are observed in the C code must be declared as *external* variables in the monitor. The external variables have the same name as the variables being monitored in the C code are treated as shared memory. The interpreter is intended for exploring ideas and algorithms and is not intended to monitor executing C programs. It may seem external variables would have no meaning if the monitor was run in the interpreter, but Copilot gives the user the ability to specify default stream values for an external variable that get used when the monitor interpreted.

A Copilot specification is *open* if defined with external symbols in the sense that values must be provided externally at runtime. To simplify writing Copilot specifications that can be interpreted and tested, constructs for external symbols take an optional environment for interpretation.

External variables are similar to global variables in other languages. They are defined by using the `extern` construct:

```
extern :: Typed a => String -> Maybe [a] -> Stream a
```

It takes the name of an external variable, a possible Haskell list to serve as the environment for the interpreter, and generates a stream by sampling the variable at each clock cycle. For example,

```
sumExterns :: Stream Word64
sumExterns = let ex1 = extern "e1" (Just [0..])
                 ex2 = extern "e2" Nothing
             in  ex1 + ex2
```

is a stream that takes two external variables `e1` and `e2` and adds them. The first external variable contains the infinite list `[0,1,2,...]` of values for use when interpreting a Copilot specification containing the stream. The other variable contains no environment (`sumExterns` must have an environment for both of its variables to be interpreted).

Sometimes, type inference cannot infer the type of an external variable. For example, in the stream definition

```
extEven :: Stream Bool
extEven = e0 `mod` 2 == 0
  where e0 = externW8 "x" Nothing
```

the type of `extern "x"` is ambiguous, since it cannot be inferred from a Boolean stream and we have not given an explicit type signature. For convenience, typed `extern` functions are provided, e.g., `externW8` or `externI64` denoting an external unsigned 8-bit word or signed 64-bit word, respectively. Please see the grammar in Appendix B for the list of all sampling functions.

In general it is best practice to define external symbols with top-level definitions, e.g.,

```
e0 :: Stream Word8
e0 = extern  "e0" (Just [2,4..])
```

so that the symbol name and its environment can be shared between streams.

Besides variables, external arrays and arbitrary functions can be sampled. The external array construct has the type

```
externArray :: (Typed a, Typed b, Integral a)
            => String -> Stream a -> Int
            -> Maybe [[a]] -> Stream b
```

The construct takes (1) the name of an array, (2) a stream that generates indexes for the array (of integral type), (3) the fixed size of the array, and (4) possibly a list of lists that is the environment for the external array, representing the sequence of array values. For example,

```
extArr :: Stream Word32
extArr = externArray "arr1" arrIdx size
           (Just $ repeat (permutations [0,1,2]))
  where
  arrIdx :: Stream Word8
  arrIdx = [0] ++ (arrIdx + 1) `mod` size

  size = 3
```

`extArr` is a stream of values drawn from an external array containing 32-bit unsigned words. The array is indexed by an 8-bit variable. The index is ensured to be less than three by using modulo arithmetic. The environment provided produces an infinite list of all the permutations of the list `[0,1,2]`.[5]

[5]The function `permutations` comes from the Haskell standard library `Data.list`.

Example 5:
Say we have defined a lookup-table (in C99) of a discretized continuous function that we want to use within Copilot:

```
double someTable[42] = { 3.5, 3.7, 4.5, ... };
```

We can use the table in a Copilot specification as follows:

```
lookupSomeTable :: Stream Word16 -> Stream Double
lookupSomeTable idx =
  externArray "someTable" idx 42 Nothing
```

Given the following values for `idx`*,* $\{1, 0, 2, 2, 1, \dots\}$*, the output of* `lookupSomeTable idx` *would be*

$$\{3.7, 3.5, 4.5, 4.5, 3.7, \dots\}$$

Finally, the constructor `externFun` takes (1) a function name, (2) a list of arguments, and (3) a possible list of values to provide its environment.

```
externFun :: Typed a => String -> [FunArg]
          -> Maybe [a] -> Stream a
```

Each argument to an external function is given by a Copilot stream. For example,

```
func :: Stream Word16
func = externFun "f" [arg e0, arg nats] Nothing
  where
  e0 = externW8 "x" Nothing
  nats :: Stream Word8
  nats = [0] ++ nats + 1
```

samples a function in C that has the prototype

```
uint16_t f(uint8_t x, uint8_t nats);
```

Both external arrays and functions must, like external variables, be defined in the target program that is monitored. Additionally, external functions must be without side effects, so that the monitor does not cause undesired side-effects when sampling functions. Finally, to ensure Copilot sampling is not order-dependent, external functions cannot contain streams containing other external functions or external arrays in their arguments, and external arrays cannot contain streams containing external functions or external arrays in their indexes. They can both take external variables, however.

Triggers. Triggers, the only mechanism for Copilot streams to effect the outside world, are defined by using the `trigger construct`:

```
trigger :: String -> Stream Bool -> [TriggerArg] -> Spec
```

The first parameter is the name of the external function, the second parameter is the guard which determines when the trigger should be evoked, and the third parameter is a list of arguments which is passed to the trigger when evoked. Triggers can be combined into a specification by using the *do*-notation:

```
spec :: Spec
spec = do
  trigger "f" (even nats) [arg fib, arg (nats * nats)]
  trigger "g" (fib > 10) []
  let x = externW64 "x" (Just [1..])
  trigger "h" (x < 10) [arg x]
```

The order in which the triggers are defined is irrelevant. To interpret this spec we run:

```
interpret 10 spec
```

which will yield the following output:

```
f:        g:         h:
(1,0)     ()        (1)
--        ()        (2)
(2,4)     ()        (3)
--        ()        (4)
(5,16)    ()        (5)
--        ()        (6)
(13,36)   --        (7)
--        --        (8)
(34,64)   --        (9)
--        --         --
```

Example 6:
We consider an engine controller with the following property: If the temperature rises more than 2.3 degrees within 0.2 seconds, then the fuel injector should not be running. Assuming that the global sample rate is 0.1 seconds, we can define a monitor that surveys the above property:

```
propTempRiseShutOff :: Spec
propTempRiseShutOff =
  trigger "over_temp_rise"
    (overTempRise && running) []

  where
  max = 500 -- maximum engine temperature
```

```
temps :: Stream Float
temps = [max, max, max] ++ temp

temp = extern "temp" Nothing

overTempRise :: Stream Bool
overTempRise = drop 2 temps > (2.3 + temps)

running :: Stream Bool
running = extern "running" Nothing
```

Here, we assume that the external variable `temp` *denotes the temperature of the engine and the external variable* `running` *indicates whether the fuel injector is running. The external function* `over_temp_rise` *is called without any arguments if the temperature rises more than 2.3 degrees within 0.2 seconds and the engine is not shut off. Notice there is a latency of one tick between when the property is violated and when the guard becomes true.*

3.6 Explicit Sharing

One of the basic tasks of a compiler is to perform optimization by eliminating common subexpressions. Consider the following Haskell code:

```
y = sin 30
myexample   = y * y
```

The expression `y` appears twice in `myexample` and `y` will be computed each time it is called. If the common subexpression is not too computationally expensive and does not occur too many times, then the multiple invocations probably is not too big a problem. On the other hand, it would be far preferable if the compiler would recognize the common expression and compute it once saving the value and sharing it whenever the value is needed. Computing an expression and temporarily storing its value in order to eliminate duplicate function calls is called *sharing*. In Haskell, the `let` binding operator facilitates sharing so

```
let  y = sin 30
in  y *y
```

results in the compiler computing `sin 30` saving its value and reusing this value in both occurrrances of `y`.

```
s1 = let x = nats + nats                    s2 = local (nats + nats) $
     in x * x                                    λ x -> x * x
```

Figure 3: Implicit sharing (s1) versus explicit sharing (s2).

EDSLs such as Copilot generally keep an internal representation of the program as an AST and

without sharing these syntax trees can grow really large and result a lot of duplicate code being created during code generation. Copilot facilitates sharing in expressions by the *local*-construct:

```
local
  :: (Typed a, Typed b)
  => Stream a
  -> (Stream a -> Stream b)
  -> Stream b
```

The local construct works similar to `let`-bindings in ordinary Haskell that we discussed above. From a semantic point of view, the streams `s1` and `s2` from Figure 3 are identical. As we will see in Section 4.1, however, certain advanced Copilot programs may force the compiler to build syntax trees that blow up exponentially. In such cases, using explicit sharing helps to avoid this problem.

Please also note that at this time sharing is not possible if you wish to use the C code verification features built into Copilot that use Frama-C. This is because let bindings are not implemented in ACSL, hence it is impossible to run Frama-C on a Copilot program that uses local bindings.

3.7 Magic labels

To ensure that the requirements and safety analyses performed early in systems development are reflected throughout the lifecycle, many guidelines for safety-critical software, such as DO-178C, require documentation of traceability from requirements to object code. In the case of Copilot, the C code generation often generates dozens of files often obfuscating the code. Consequently, to promote the acceptance of high-assurance RV, Copilot gives the programmer the capability to insert labels into expressions that are inserted into the generated C code making it easy to trace the C monitor code to an expression in the monitor specification.

The label operation has the following type signature:

```
label
  :: (Typed a)
  => String
  -> Stream a
  -> Stream a
```

The term *magic label* refers to the possibility to add special characters in labels, namely "?", which would modify the code generation by splitting an expression so that it generates two simple rather than one complex C function. To invoke the splitting character, simply do:

```
label "?blabla" expr
```

where `expr` is any stream. This fundamentally does not change the big-step semantics of the generated program, however it splits a big expression into smaller expressions which can then be more easily read or verified with Frama-C.

Here is an example of the splitting process in Figure 4 for the following simple Copilot code:

```
import qualified Copilot.Compile.SBV as S

alt :: Stream Bool
alt = (label "?splitting" $ not $ externB "externvar" Nothing)

spec :: Spec
spec = do
trigger "trigger" (alt) []

main = do
reify spec >>= S.proofACSL S.defaultParams
```

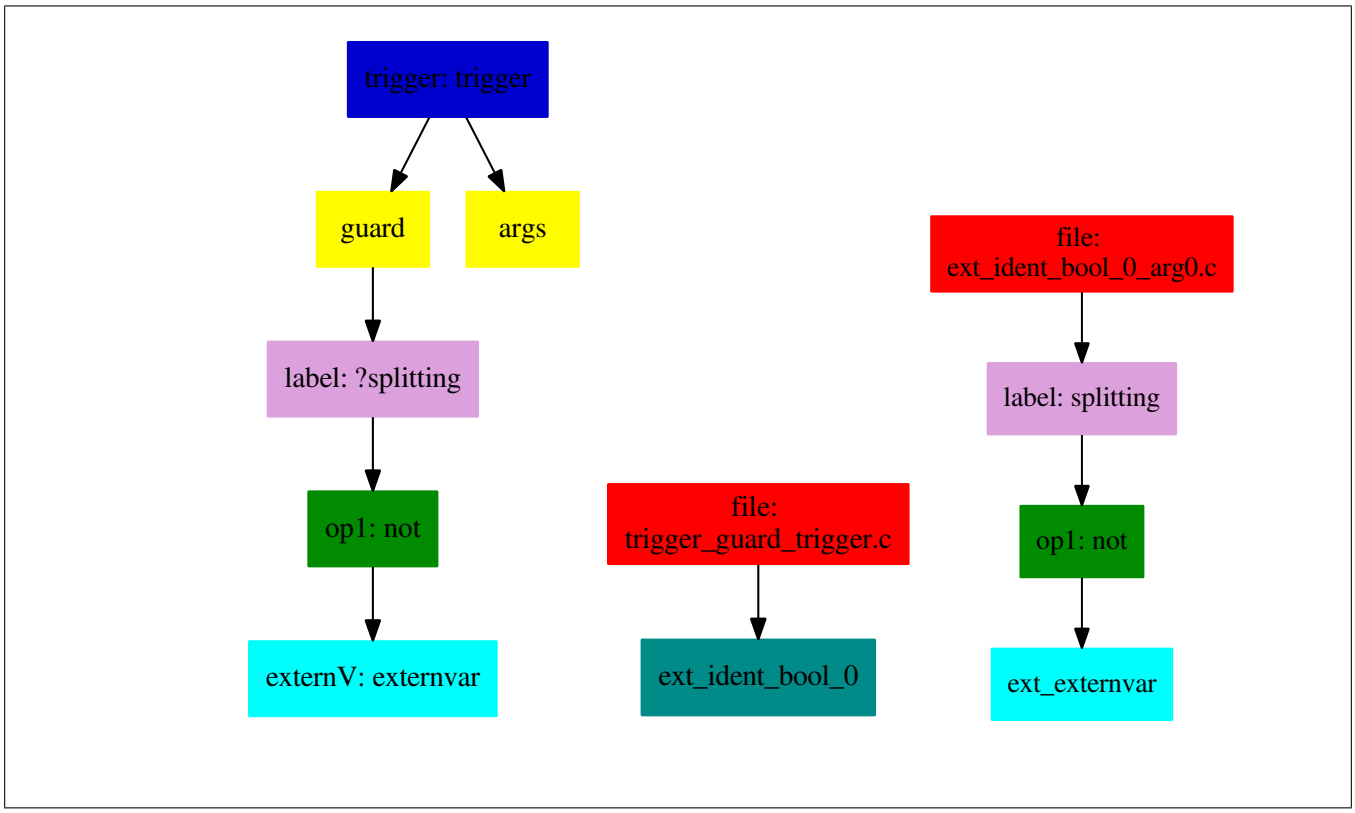

Figure 4: The splitting process. The AST on the left is split into two smaller ASTs, that then generate two separate functions in two separate files, making it easier to prove the contracts for them.

4 Examples

Here we give a few examples of Copilot code. Additional examples can be found in the Examples directory of the Copilot language.

4.1 The Boyer-Moore Majority-Vote Algorithm

In this section we demonstrate how to use Haskell as an advanced macro language on top of Copilot by implementing an algorithm for solving the voting problem in Copilot.

Reliability in mission critical software is often improved by replicating the same computations on separate hardware and by doing a vote in the end based on the output of each system. The majority vote problem consists of determining if in a given list of votes there is a candidate that has more than half of the votes, and if so, of finding this candidate.

The Boyer-Moore Majority Vote Algorithm [MB81, Hes05] solves the problem in linear time and constant memory. It does so in two passes: The first pass chooses a candidate; and the second pass asserts that the found candidate indeed holds a majority.

The algorithm for the first pass involves the sequence of elements we are interpreting, a single element that represents the current majority, and a counter, which is initially set to zero. The algorithm is as follows:

- Initialize an element m and a counter i where $i = 0$
- For each element x of the input sequence:
  - If $i = 0$ then let $m = x$ and $i = 1$
  - else if $m = x$ then increment i
  - else let $i = i - 1$
- Return m

This algorithm will produce an output even if there is no majority, which is why the second pass is needed to verify that the output of the first pass is valid.

```
majorityPure :: Eq a => [a] -> a
majorityPure []     = error "majorityPure: empty list!"
majorityPure (x:xs) = majorityPure' xs x 1

majorityPure' []     can _   = can
majorityPure' (x:xs) can cnt =
  let
    can' = if cnt == 0 then x else can
    cnt' = if cnt == 0 || x == can then succ cnt else pred cnt
  in
    majorityPure' xs can' cnt'
```

Figure 5: The first pass of the majority vote algorithm in Haskell.

The first pass can be implemented in Haskell as shown in Figure 5. The second pass, which simply checks that a candidate has more than half of the votes, is straightforward to implement

```
aMajorityPure :: Eq a => [a] -> a -> Bool
aMajorityPure xs can = aMajorityPure' 0 xs can > length xs `div` 2

aMajorityPure' cnt []     _   = cnt
aMajorityPure' cnt (x:xs) can =
  let
    cnt' = if x == can then cnt+1 else cnt
  in
    aMajorityPure' cnt' xs can
```

Figure 6: The second pass of the majority vote algorithm in Haskell.

```
majority :: (Eq a, Typed a) => [Stream a] -> Stream a
majority []     = error "majority: empty list!"
majority (x:xs) = majority' xs x 1

majority' []     can _   = can
majority' (x:xs) can cnt =
  local
    (if cnt == 0 then x else can) $
      λ can' ->
        local (if cnt == 0 || x == can then cnt+1 else cnt-1) $
          λ cnt' ->
            majority' xs can' cnt'
```

Figure 7: The first pass of the majority vote algorithm in Copilot.

```
aMajority :: (Eq a, Typed a) => [Stream a] -> Stream a -> Stream Bool
aMajority xs can = aMajority' 0 xs can > (fromIntegral (length xs) `div` 2)

aMajority' cnt []     _   = cnt
aMajority' cnt (x:xs) can =
  local
    (if x == can then cnt+1 else cnt) $
      λ cnt' ->
        aMajority' cnt' xs can
```

Figure 8: The second pass of the majority vote algorithm in Copilot.

and is shown in Figure 6. E.g. applying `majorityPure` on the string `AAACCBBCCCBCC` yields `C`, which `aMajorityPure` can confirm is in fact a majority.

When implementing the majority vote algorithm for Copilot, we can use reuse almost all of the code from the Haskell implementation. However, as functions in Copilot are macros that are expanded at compile time, care must be taken in order to avoid an explosion in the code size. Hence, instead of using Haskell's built-in *let*-blocks, we use explicit sharing, as described in Section 3.6. The Copilot implementations of the first and the second pass are given in Figure 7 and Figure 8 respectively. Comparing the Haskell implementation with the Copilot implementation, we see that the code is almost identical, except for the type signatures and the explicit sharing annotations.

The full Well Clear specification can be found in Appendix A.

One application that suits well to the Copilot language is the monitoring of violations of separation criteria between two aircraft. In civil aviation, the air traffic management system is designed to keep aircraft at a safe distance from each other avoid even the possibility of a mid-air collision. In particular, aircraft are generally considered safety separated if they maintain 5 mile horizontal distance from another aircraft and 1000 ft vertical separation. Technically, we define a safe-separation volume around an aircraft that if violated by an intruder aircraft, the two aircraft are said to be in conflict. The mathematical definition of this volume of space around the moving aircraft is referred to as Well Clear. The well-clear boundary defines the *well-clear violation volume*, such that "aircraft pairs jointly occupying this volume are considered to be in a well-clear violation" [MNH+15].

Observe the *Well clear violation* section of the code below. The function `wcv` takes in parameters and checks the horizontal and vertical boundries for violation via the `horizontalWCV` and `verticalWCV` functions respectively.

```
-------------------------
-- Well clear violation --
-------------------------

wcv :: (Vect2 -> Vect2 -> Stream Double) ->
       Vect2 -> Stream Double ->
       Vect2 -> Stream Double ->
       Stream Bool
wcv tvar s sz v vz = (horizontalWCV tvar s v) && (verticalWCV sz vz)

verticalWCV :: Stream Double -> Stream Double -> Stream Bool
verticalWCV sz vz =
  ((abs $ sz) <= zthr) ||
  (0 <= (tcoa sz vz) && (tcoa sz vz) <= tcoathr)

horizontalWCV :: (Vect2 -> Vect2 -> Stream Double) -> Vect2 -> Vect2 -> Stream Bool
horizontalWCV tvar s v =
  (norm s <= dthr) ||
  (((dcpa s v) <= dthr) && (0 <= (tvar s v)) && ((tvar s v) <= tthr))
```

The main parameters for these functions are `s, v, sz,` and `vz`. `s,v` represents the horizontal position and velocity of the aircraft being monitored, while `sz,vz` represents the vertical position and velocity. We assume that the "other" aircraft is at the origin, so their horizontal and vertical positions are zero.

To determine if there is a horizontal well clear violation we check whether the position of our own craft (`s`) is less than the distance threshold (`dthr`), which is the closest two aircraft can be within well-clear regulations. If it is, then there is a violation. If not, we then check that the distance at closest point of approach (`dcpa`) corresponding to our position and speed is also less than the distance threshold *and* that the time of that closest point of approach (`tvar`) corresponding to our position and speed is greater than zero and less than the time threshold. If both of these are true, then we have a violation.

Essentially, we have a violation if either our aircraft is too close to the "other" aircraft (less than `dthr`) or if our aircraft will be too close to the "other aircraft" within a certain amount of time. For example, assume that the distance threshold we choose is 400m. If `s` is less than 400m, we will have a violation. Otherwise, assume our time threshold is 30 seconds. If at the point in the paths of our aircrafts where the two paths are closest is less than 400m apart *and* if both aircrafts hit that point in their paths within 30 seconds of each other, there is a well-clear violation.

The vertical well clear violation function works much the same way. It first checks whether the vertical position (`sz`) is less than the vertical distance threshold. It then checks that the time at which the two aircraft will have the same vertical given the distance and velocity is less than the time to co-altitude threshold, which is the minimum time at which the two aircraft are allowed to have the same altitude.

The rest of the Well Clear Code can be found in Appendix A. The horizontal and vertical distance and time thresholds are given as external variables, as described in Section 3.5. Similarly, the relative velocity and position for our aircraft with respect to the "other" aircraft are given as external variables in the *Relative velocity/position* section of the code. The *Vector stuff, Time variables,* and *Some tools for times* sections of the code give us various streams and functions that we reference in the *Well clear violation* section and allow us to do the calculations.

5 Logic Libraries

There are three logic libraries for Copilot that are useful for monitoring temporal logic properties: the LTL, PTLTL, and MTL libraries.

5.1 Bounded Linear Temporal Logic

The LTL library provides an implementation of a bounded version of Linear Temporal Logic. Particularly, it provides implementations for the temporal operators ◯ (`next`), □ (`always`), ◊ (`eventually`), **U** (`until`), and **R** (`release`). Descriptions of the functions in the bounded LTL library are given in Figure 9.

All LTL properties except those involving only `next` require a nonnegative bound `n` that gives the number of transitions after the current period that the property must hold for in order for the bounded LTL property to be true. A stream must have sufficient history when being used with an

LTL operator, i.e., any stream to which `next` is applied must have one value that may be dropped from it and if the nonnegative bound `n` of an LTL operator is greater than 0, then there must be `n` values in the stream that can be dropped (over `n` values in the stream).

`next s` $\rightsquigarrow s_{\bigcirc}$	where	$\forall t \in \mathbb{N}\ (s_{\bigcirc}(t) = s(t+1))$
`always n s` $\rightsquigarrow s_{\Box}$	where	$\forall t \in \mathbb{N}\ (s_{\Box}(t) = \texttt{True} \Leftrightarrow \forall t' \in [0, n] \oplus t\ (s(t') = \texttt{True}))$
`eventually n s` $\rightsquigarrow s_{\Diamond}$	where	$\forall t \in \mathbb{N}\ (s_{\Diamond}(t) = \texttt{True} \Leftrightarrow \forall t' \in [0, n] \oplus t\ (s(t') = \texttt{True}))$
`until n s0 s1` $\rightsquigarrow s_{\mathbf{U}}$	where	$\forall t \in \mathbb{N}\ (s_{\mathbf{U}}(t) = \texttt{True} \Leftrightarrow$
		$\exists t' \in [0, n] \oplus t\ (s_1(t') = \texttt{True}\ \wedge\ \forall t'' \in [t, t')\ (s_0(t'') = \texttt{True})))$
`release n s0 s1` $\rightsquigarrow s_{\mathbf{R}}$	where	$\forall t \in \mathbb{N}(s_{\mathbf{R}}(t) = \texttt{True} \Leftrightarrow \forall t' \in [0, n] \oplus t\ (s_1(t') = \texttt{True}) \vee$
		$\exists t' \in [0, n] \oplus t\ (s_0(t') = \texttt{True}\ \wedge\ \forall t'' \in [t, t']\ (s_1(t'') = \texttt{True})))$

where `n` $\rightsquigarrow n$, `s` $\rightsquigarrow s$, `s0` $\rightsquigarrow s_0$, and `s1` $\rightsquigarrow s_1$

Figure 9: A description of the LTL library functions.

Figure 10 provides an example of the use of LTL library functions to generate streams for monitors:

- The first stream `lockedUntil` is used to monitor the property that when a lock is signalled to be unlocked (`canUnlock` becomes true), the lock is allowed to be unlocked (`canUnlock` becomes true) within four sampled values, and the the lock remains locked until it is allowed to be unlocked.
- The second stream `switchOnInTime`, is used to monitor that when an event `isEvent` occurs, a switch turns on (`isSwitchOn` becomes true) within three sampled values.

5.2 Past-Time Linear Temporal Logic

The PTLTL library provides an implementation of Past-Time Linear Temporal Logic, including implementations for temporal operators $\overleftarrow{\bigcirc}$ (`previous`), $\overleftarrow{\Box}$ (`alwaysBeen`), $\overleftarrow{\Diamond}$ (`eventuallyPrev`), and $\mathbf{S}$ (`since`), which can be used to monitor properties that must hold over all the current and past values of the stream.

The code in Figure 12 provides an example of the use of the PTLTL function `alwaysBeen` to produce a stream that could be used in a monitor: the stream `countMonotonicallyIncreasing` is true when the values of the nonnegatively-valued stream `count` have increased monotonically so far.

```
lockedUntil :: Stream Bool
lockedUntil = unlockSignal ==> until 4 isLocked canUnlock
  where
  unlockSignal :: Stream Bool
  unlockSignal =
    (replicate 5 False) ++ (extern "unlock_signal" Nothing)
  isLocked :: Stream Bool
  isLocked = (replicate 5 False) ++ (extern "locked" Nothing)
  canUnlock :: Stream Bool
  canUnlock = (replicate 5 False) ++ (extern "can_unlock" Nothing)

switchOnInTime :: Stream Bool
switchOnInTime = isEvent ==> eventually 3 isSwitchOn
  where
  isSwitchOn :: Stream Bool
  isSwitchOn = (replicate 4 False) ++ (extern "on" Nothing)
  isEvent :: Stream Bool
  isEvent = (replicate 4 False) ++ (extern "event" Nothing)

spec :: Spec
spec = do
  trigger "unlock_problem" (not lockedUntil) []
  trigger "switch_on_time_problem" (not switchOnInTime) []
```

Figure 10: An example use of some LTL library functions.

`previous s` $\rightsquigarrow s_{\overleftarrow{\bigcirc}}$	where	$s_{\overleftarrow{\bigcirc}}(0) = \texttt{False} \wedge \forall t \in \mathbb{N}^+ (s_{\overleftarrow{\bigcirc}}(t) = s(t-1))$
`alwaysBeen s` $\rightsquigarrow s_{\overleftarrow{\Box}}$	where	$\forall t \in \mathbb{N}\ (s_{\overleftarrow{\Box}}(t) = \texttt{True} \Leftrightarrow \forall t' \in [0,t]\ (s(t') = \texttt{True})$
`eventuallyPrev s` $\rightsquigarrow s_{\overleftarrow{\Diamond}}$	where	$\forall t \in \mathbb{N}\ (s_{\overleftarrow{\Diamond}}(t) = \texttt{True} \Leftrightarrow \exists t' \in [0,t]\ (s(t') = \texttt{True})$
`since s0 s1` $\rightsquigarrow s_{\mathbf{S}}$	where	$s_{\mathbf{S}}(0) = \texttt{True} \wedge \forall t \in \mathbb{N}^+\ (s_{\mathbf{S}}(t) = \texttt{True} \Leftrightarrow \exists t' \in [0,t-1]$
		$(s_1(t') = \texttt{True} \Rightarrow \forall t'' \in [t'+1,t]\ (s_0(t'') = \texttt{True})))$

where `s` $\rightsquigarrow s$, `s0` $\rightsquigarrow s_0$, and `s1` $\rightsquigarrow s_1$

Figure 11: A description of the PTLTL library functions.

```
countMonotonicallyIncreasing :: Stream Bool
countMonotonicallyIncreasing = alwaysBeen (count >= ([0] ++ count))
  where
  count :: Stream Word32
  count = extern "count" Nothing
```

Figure 12: An example use of PLTL library function `alwaysBeen`.

5.3 Bounded Metric Temporal Logic

The MTL library is useful for monitoring properties involving bounded real time. While the LTL library can be used to monitor properties associated with real time values if the sampling rate is known, if samples are not taken precisely at the assumed rate and the amount of time between samples may vary, then depending on a constant sampling rate may result in undesired values.

The MTL library provides an implementation of bounded Metric Temporal Logic over a discrete time domain with implementations of metric temporal operators $\mathbf{U}_I$ (`until`), $\mathbf{S}_I$ (`since`), $\mathbf{R}_I$ (`release`), $\mathbf{T}_I$ (`trigger`), $\Diamond_I$ (`eventually`), $\Box_I$ (`always`), $\overleftarrow{\Diamond}_I$ (`eventuallyPrev`), and $\overleftarrow{\Box}_I$ (`alwaysBeen`) along with matching variants $\mathbf{U}^{\downarrow}_I$ (`matchingUntil`), $\mathbf{S}^{\downarrow}_I$ (`matchingSince`), $\mathbf{R}^{\downarrow}_I$ (`matchingRelease`), and $\mathbf{T}^{\downarrow}_I$ (`matchingTrigger`). A description of the MTL library functions (omitting the matching variants) is given in Figure 13.

`until l u clk dist s0 s1` $\rightsquigarrow s_{\mathbf{U}}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\mathbf{U}}(t) = \texttt{True} \Leftrightarrow$
		$\exists d \in [l,u]\ s.t.\ t+d \in \mathbb{T}\ (s^{clk}_1(t+d) = \texttt{True}$
		$\wedge\ \forall t' \in [0,d) \oplus t \cap \mathbb{T}\ (s^{clk}_0(t') = \texttt{True})))$
`since l u clk dist s0 s1` $\rightsquigarrow s_{\mathbf{S}}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\mathbf{S}}(t) = \texttt{True} \Leftrightarrow$
		$\exists d \in [l,u]\ s.t.\ t-d \in \mathbb{T}\ (s^{clk}_1(t-d) = \texttt{True}$
		$\wedge\ \forall t' \in -[0,d) \oplus t \cap \mathbb{T}\ (s^{clk}_0(t') = \texttt{True})))$
`release l u clk dist s0 s1` $\rightsquigarrow s_{\mathbf{R}}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\mathbf{R}}(t) = \texttt{True} \Leftrightarrow$
		$\forall d \in [l,u]\ s.t.\ t+d \in \mathbb{T}\ (s^{clk}_1(t+d) = \texttt{True}$
		$\vee\ \exists t' \in [0,d) \oplus t \cap \mathbb{T}\ (s^{clk}_0(t') = \texttt{True})))$
`trigger l u clk dist s0 s1` $\rightsquigarrow s_{\mathbf{T}}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\mathbf{R}}(t) = \texttt{True} \Leftrightarrow$
		$\forall d \in [l,u]\ s.t.\ t-d \in \mathbb{T}\ (s^{clk}_1(t-d) = \texttt{True}$
		$\vee\ \exists t' \in -[0,d) \oplus t \cap \mathbb{T}\ (s^{clk}_0(t') = \texttt{True})))$
`eventually l u clk dist s` $\rightsquigarrow s_{\Diamond}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\Diamond}(t) = \texttt{True} \Leftrightarrow$
		$\exists d \in [l,u]\ s.t.\ t+d \in \mathbb{T}\ (s^{clk}(t+d) = \texttt{True}))$
`always l u clk dist s` $\rightsquigarrow s_{\Box}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\Box}(t) = \texttt{True} \Leftrightarrow$
		$\forall d \in [l,u]\ s.t.\ t+d \in \mathbb{T}\ (s^{clk}(t+d) = \texttt{True}))$
`eventuallyPrev l u clk dist s` $\rightsquigarrow s_{\overleftarrow{\Diamond}}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\overleftarrow{\Diamond}}(t) = \texttt{True} \Leftrightarrow$
		$\exists d \in [l,u]\ s.t.\ t-d \in \mathbb{T}\ (s^{clk}(t-d) = \texttt{True}))$
`alwaysBeen l u clk dist s` $\rightsquigarrow s_{\overleftarrow{\Box}}$	where	$\forall t \in \mathbb{T}\ (s^{clk}_{\overleftarrow{\Box}}(t) = \texttt{True} \Leftrightarrow$
		$\forall d \in [l,u]\ s.t.\ t-d \in \mathbb{T}\ (s^{clk}(t-d) = \texttt{True}))$

where $\mathbb{T}$ is the set of sampled times, `l` $\rightsquigarrow l$, `u` $\rightsquigarrow u$, `s` $\rightsquigarrow s$, `s0` $\rightsquigarrow s_0$, `s1` $\rightsquigarrow s_1$, `clk` $\rightsquigarrow clk$, and for any stream s and time $t \in \mathbb{T}$, $s^{clk}(t) = s(clk^{-1}(t))$

Figure 13: A description of the MTL library functions.

Each MTL operator must, in addition to its operand(s), be provided with a lower and upper bound for the interval, a stream `clk` of integral time values sampled at the same rate as the stream(s) being operated on, and a positive integer number of time units `dist` giving the smallest

possible difference between adjacent clock samples. Interval bounds must be nonnegative integers, and intervals include all sampled times that fall between or equal the provided bounds.

For example, the property $\Box_{[10,30]}P$ corresponds to `always 10 30 clk 2 p` where `clk` is the stream of clock samples that have a period of at least 2 and `p` is a `Stream Bool` that is true when P holds.

As with the LTL library, any streams used with future MTL operators must have sufficient history. Since it is less straightforward to guarantee that streams have sufficient history for use with MTL operators, the past MTL operators are more convenient and practical for use in monitors.

Figure 14 contains a specification that monitors a plane's airspeed for a violation of the property that a plane's speed does not exceed a maximum airspeed `maxAS` for 60 seconds or longer, where it is known that streams' values are sampled at a period that is greater than or equal to 1 second.

If this property holds, then at some time within the past 60 seconds, the airspeed should be less than or equal to the maximum airspeed, i.e., the following MTL property should hold:

$$\overleftarrow{\Diamond}_{[0,60]}\ \texttt{airspeed} \leq \texttt{maxAS}$$

Whenever this property does not hold, the trigger is evoked in response to the detected issue.

```
maxAirspeedViolation :: Spec
maxAirspeedViolation =
  Copilot.Language.trigger
    "airspeed_violation"
    (not (eventuallyPrev 0 60 clk 1 (airspeed <= maxAS)))
    [arg airspeed, arg clk]
  where
  clk :: Stream Word64
  clk = extern "clk_sec" Nothing
  airspeed :: Stream Double
  airspeed = extern "airspeed" Nothing
  maxAS :: Stream Double
  maxAS = extern "max_airspeed" Nothing
```

Figure 14: An example use of the MTL library function `eventuallyPrev`.

6 Advanced Topics

Here we discuss more in depth topics within the Copilot Language. These topics are not necessary to gain a basic understanding of the language, but they are helpful as a reference and for more in depth understanding.

6.1 Proofs of Monitors

In addition to proofs on the soundness of the Copilot compilation process, we also support (in the `Copilot.Theorem` module) some automated proving of safety properties about Copilot specifications themselves. A proposition is a Copilot value of type `Prop Existential` or `Prop Universal`, which can be introduced using `exists` and `forall`, respectively. These are functions taking as an argument a normal Copilot stream of type `Stream Bool`. Propositions can be added to a specification using the `prop` and `theorem` functions, where `theorem` must also be passed a tactic for autmatically proving the proposition. Consider this Copilot monitor specification for a version of the fibonacci sequence:

```
module Fib where

import Prelude ()
import Copilot.Language
import Copilot.Language.Reify
import Copilot.Theorem
import Copilot.Theorem.Prover.SMT

fib = do
  theorem "fibn_gre_n" (forall $ fibn >= n) $ kInduction def cvc4

  observer "fibn" fibn
  observer "n"    n
  where
    fibn :: Stream Word32
    fibn = [1, 1] ++ (fibn + drop 1 fibn)
    n = [0] ++ (n + 1)
```

In the specification above, in addition to observing the value of streams in the specification, `fibn_gre_n` names a theorem provable automatically with induction using the Z3 SMT solver. This theorem can be checked during reification:

```
*Fib> reify fib
fibn_gre_n: valid (proved with k-induction (k = 3))
Finished: fibn_gre_n: proof checked successfully
```

The `Copilot.Theorem` module provides two main mechanisms for interacting with SMT solver backends: a generic backend at `Copilot.Theorem.Prover.SMT` that nominally supports a wide range of SMT solvers and `Copilot.Theorem.Prover.Z3`, a backend specialized to Z3. The example above uses the generic `SMT` backend with the CVC4 SMT solver.

See Figure 15 for an overview of SMT solvers that have at least been tested to work with our tool. The second column contains the value of type `Backend` from the `Copilot.Theorem.Prover.SMT` module which must be passed to the tactics in this module for executing the tactic using that SMT solver (e.g., as is done in the second argument to `kInduction` in the example above). To use a

Tool	`Backend`	Version	Interface	NL Real Arith.	Trig. funs.	Quantif.	Bitvec.
Alt-Ergo	`altErgo`	0.99.1	SMTLib2	✓	X	✓	X
CVC4	`cvc4`	1.4	SMTLib2	Some	X	✓	✓
DReal	`dReal`	2.15.01	SMTLib2	✓	✓	X	X
MathSAT	`mathSat`	5.3.7	SMTLib2	Some	X	X	✓
MetiTarski	`metit`	2.5	TPTP	✓	✓	✓	X
Yices	`yices`	2.4.0	SMTLib2	Some	X	X	✓
Z3	`z3`	4.4.0	SMTLib2	✓	X	✓	✓

Figure 15: An overview of some supported SMT solvers.

certain SMT solver, the solver must be installed and the executable should reside in one of the directories listed in the `$PATH` environment variable.

As seen in the figure mentioned above, different SMT solvers have different feature sets and the methods for using the various features are generally not well standardized. As a result, we also have backend specialized to Z3, one of the more feature-rich SMT solvers we've tested. The example above using the Z3 backend looks very similar to the version above:

```
module Fib where

import Prelude ()
import Copilot.Language
import Copilot.Language.Reify
import Copilot.Theorem
import Copilot.Theorem.Prover.Z3

fib = do
  theorem "fibn_gre_n" (forall $ fibn >= n) $ kInduction def

  observer "fibn" fibn
  observer "n"    n
  where
    fibn :: Stream Word32
    fibn = [1, 1] ++ (fibn + drop 1 fibn)
    n = [0] ++ (n + 1)
```

Instead of importing `Copilot.Theorem.Prover.SMT`, we import `Copilot.Theorem.Prover.Z3`, and we no longer need to pass the `cvc4` argument to `kInduction` (because the SMT solver backend will be Z3). But now when we go to reify `fib`:

```
*Fib> reify fib
fibn_gre_n: unknown (proof by k-induction failed)
Finished: fibn_gre_n: proof failed
Warning: failed to check some proofs.
```

This demonstrates an important limitation of the `Copilot.Theorem.Prover.SMT` backend: it encodes fixed-width integers using the SMTLib `Int` type (with some extra propositions about the bounds of `Int` variables)—it *does not model overflow*. The `Copilot.Theorem.Prover.Z3` backend, however, encodes Copilot's fixed-width integer types using Z3's bitvectors. Now if we allow `fibn` to overflow, the theorem from the above example is clearly not true.

The above example contains a single `theorem`. A `theorem` takes three arguments: the name of the theorem (which is only used to identify the theorem in output), a proposition, and a proof "tactic" to use when trying to automatically prove the proposition using SMT solvers. The `Copilot.Theorem.Prover.SMT` and `Copilot.Theorem.Prover.Z3` modules both export the same set of tactics[6]:

- `onlySat`: check that the proposition is satisfiable.
- `onlyValidity`: check that the negation of the proposition is unsatisfiable.
- `induction`: special case of k-induction, with $k = 0$.
- `kInduction`: version of induction where the induction hypothesis is strengthened by including k previous states.

Additionally, a few other tactics live in `Copilot.Theorem.Tactics`:

- `instantiate`: turn a proof for a universally quantified proposition into a proof for an existentially quantified one.
- `assume`: prove a proposition true under an assumption.
- `admit`: prove anything.

Also, we support some older versions of the Kind2 model checker.

6.2 Correctness of the generated code

6.2.1 Frama-c

For this, make sure the following is installed:

- GNU parallel
- frama-c (version at least sodium)
- why3

[6]In fact, these modules share a lot a duplicated code and desparately need to be refactored.

- cvc4 prover

It is possible to generate C code optimized for frama-c by using the `proofACSL` function from SBV.Compile (instead of `compile` as shown in 2.2). It will then generate the C source code (which should be in the `copilot-sbv-codegen` folder). Go into that folder and just run `make fwp` to use frama-c with the WP plugin. This should generate two log files: a first exhaustive one called `logfwp`, and another which summaries the former which is called `logfwpcompact`. The command make fwp is detailed in Figure 16.

```
parallel frama-c -wp -wp-out . -wp-timeout 20 -wp-prover CVC4 -wp-split {} ::: *.c | tee >
    logfwp >(grep 'Proved\|Unknown\|Timeout\|Failed\|Qed:\s\|CVC4:\s\|Parsing .*\.c' >
    logfwpcompact) >(grep 'Proved\|Qed:\s\|CVC4:\s\|Unknown\|Timeout\|Failed\|Parsing .*\.c')
```

Figure 16: The bash command.

It is also possible to run frama-c with the value analysis plugin by running `make fval`. Be careful, the value analysis plugin requires to preload all the C sources files together, which requires a lot of RAM memory for big projects.

It is recommended to refactor your code before trying to use frama-c on it, by for example splitting non-trivial expressions using magic labels, deleting all local variables (use functions) and bitwise operators. If this step is neglected, it may happen that frama-c would start swapping all your memory resulting in a system crash.

6.2.2 Splint

Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. If you have splint installed on your computer, you can run splint on the C source code generated by the Copilot compiler.

6.2.3 Dot

It is also possible to generate dot graphs for each C source file generated. The diagrams are intended as documentation to make the files more understandable. For this, you should check that you have dot installed. Extract the dot code from a C source file, and then just run `dot -Tps code.gv -o code.ps` [7].

[7] A script doing that automatically is provided here `https://raw.githubusercontent.com/Copilot-Language/examplesForACSL/master/WCV/scriptdot.cpp`

Here we provide a brief description of Copilot compilation process using the SBV backend. After parsing, we obtain a first AST that is then reified to a core AST (some beta reduction are done), using standard reification techniques in [Gil09b]. Then AST transforms are applied that preserve the semantics. This transforms some operators that do not exist in SBV into expressions with only operators with SBV (recip, abs, ...). The AST is then transformed into and AST format used by SBV, and in the process we generate both ACSL source code and DOT code. This means that the SBV AST contains "decorative" nodes that contain a ACSL or DOT code describing the child AST (TODO modify expression).

The ACSCL and DOT nodes are then converted into comments into the C sources files that are then checked against the source code generated for the child AST with frama-c wp plugin. This helps us increase the confidence in the compilation process, by checking two different techniques (pretty printing and compilation) each against the other, hence reducing the probability that a similar bug is present both in the pretty printer (which generates the ACSL contracts) and the SBV compiler.

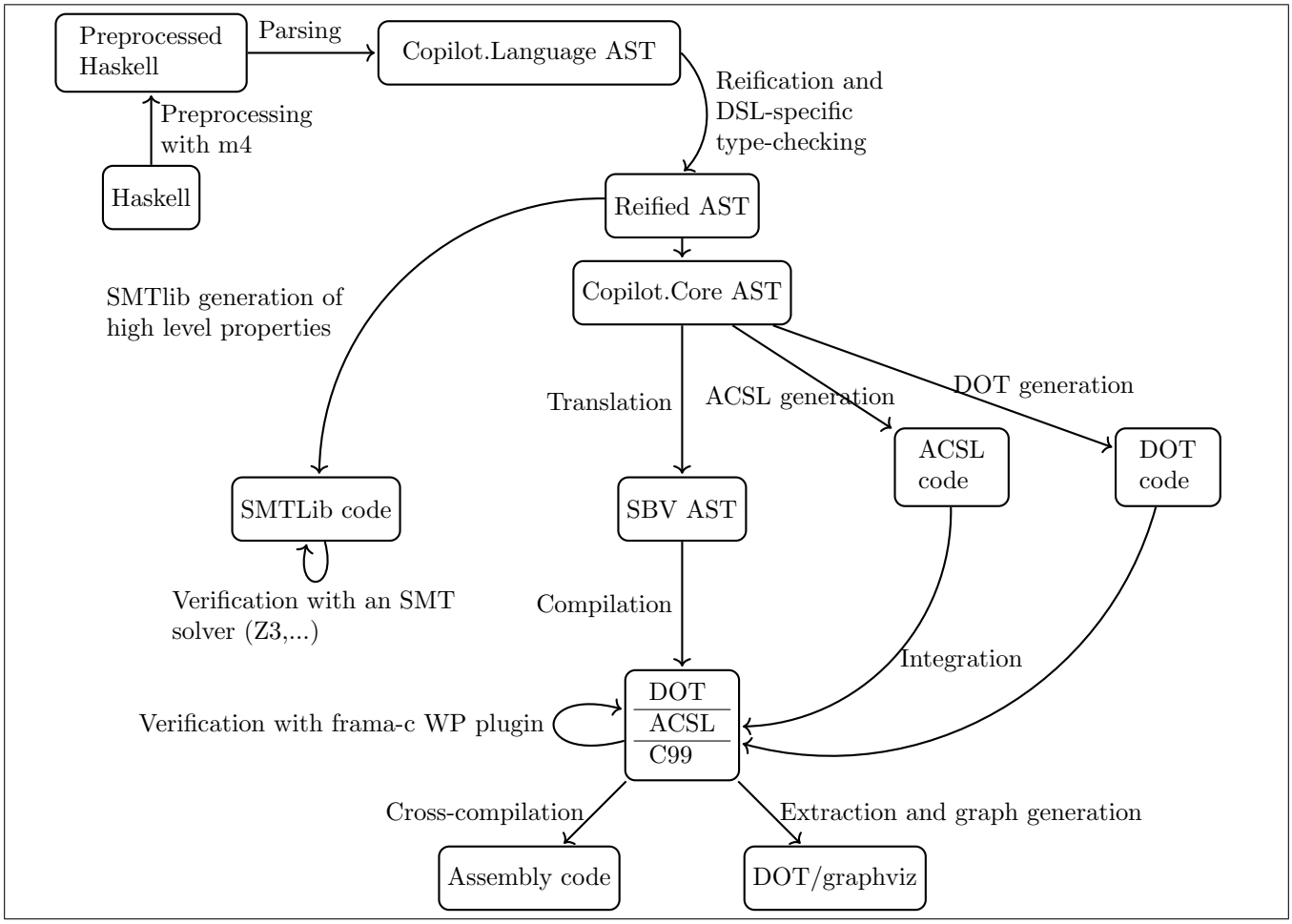

Figure 17: The whole process of WCV compilation.

If you go in the `copilot-sbv-codegen` folder, and run `make all` inside it, it should generate an archive file named `internal.a`, using the compiler defined in the file `copilot.mk` (default is compcert, you have to install it with its standard library). This can create some problems during the linking process if you change the compiler for the final compilation (typically `gcc -m32 -lm main.c internal.a` where `main.c` is where your controller code is). For this purpose, we recommend installing first the cross compiler (example `arm-none-eabi-gcc`), and compile the *whole* project using that very same compiler using the command `arm-none-eabi-gcc -m32 -lm [options] *.c`.

References

[BF93] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19:3–12, January 1993.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[DDE08] M.B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 228–237, 2008.

[Far04] H.A. Farhat. *Digital design and computer organization*. Number v. 1 in Digital Design and Computer Organization. CRC Press, 2004.

[Gil09a] Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[Gil09b] Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM.

[GP10] Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.

[Hes05] Wim H. Hesselink. The boyer-moore majority vote algorithm, 2005.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.

[MB81] Strother J. Moore and Robert S. Boyer. MJRTY - A Fast Majority Vote Algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, February 1981.

[MNH+15] César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, and María Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015)*, Prague, Czech Republic, September 2015.

[Sha01] Lui Sha. Using simplicity to control complexity. *IEEE Software*, pages 20–28, July/August 2001.

A Well Clear Code

Here we give the specification for Well Clear as described in Section **??**

```
module WCV where

import Prelude ()

import Copilot.Language
import Copilot.Language.Reify
import Copilot.Theorem

import qualified Copilot.Language.Operators.Propositional as P

dthr, tthr, zthr, tcoathr :: Stream Double
dthr    = extern "dthr" Nothing
tthr    = extern "tthr" Nothing
zthr    = extern "zthr" Nothing
tcoathr = extern "tcoathr" Nothing

type Vect2 = (Stream Double, Stream Double)

-------------------------------
-- Relative velocity/position --
-------------------------------

vx, vy, vz :: Stream Double
```

```
vx = extern "relative_velocity_x" Nothing
vy = extern "relative_velocity_y" Nothing
vz = extern "relative_velocity_z" Nothing

v :: (Stream Double, Stream Double)
v = (vx, vy)

sx, sy, sz :: Stream Double
sx = extern "relative_position_x" Nothing
sy = extern "relative_position_y" Nothing
sz = extern "relative_position_z" Nothing

s :: (Stream Double, Stream Double)
s = (sx, sy)

-------------------
-- Vector stuff --
-------------------

(|*|) :: Vect2 -> Vect2 -> Stream Double
(x1, y1) |*| (x2, y2) = (x1 * x2) + (y1 * y2)

sq :: Vect2 -> Stream Double
sq x = x |*| x

norm :: Vect2 -> Stream Double
norm = sqrt ∘ sq

det :: Vect2 -> Vect2 -> Stream Double
det (x1, y1) (x2, y2) = (x1 * y2) - (x2 * y1)

(~=) :: Stream Double -> Stream Double -> Stream Bool
a ~= b = (abs (a - b)) < 0.001

neg :: Vect2 -> Vect2
neg (x, y) = (negate x, negate y)

---------------------
-- Time variables --
---------------------

tau :: Vect2 -> Vect2 -> Stream Double
tau s v = mux (s |*| v < 0) ((-(sq s)) / (s |*| v)) (-1)

tcpa :: Vect2 -> Vect2 -> Stream Double
```

```
tcpa s v@(vx, vy) = mux (vx ~= 0 && vy ~= 0) 0 (-(s |*| v)/(sq v))

taumod :: Vect2 -> Vect2 -> Stream Double
taumod s v = mux (s |*| v < 0) ((dthr * dthr - (sq s))/(s |*| v)) (-1)

tep :: Vect2 -> Vect2 -> Stream Double
tep s v = mux ((s |*| v < 0) && ((delta s v dthr) >= 0))
  (theta s v dthr (-1))
  (-1)

delta :: Vect2 -> Vect2 -> Stream Double -> Stream Double
delta s v d = (d*d) * (sq v) - ((det s v)*(det s v))

theta :: Vect2 -> Vect2 -> Stream Double -> Stream Double -> Stream Double
theta s v d e = (-(s |*| v) + e * (sqrt $ delta s v d)) / (sq v)

-------------------------
-- Some tools for times --
-------------------------

tcoa :: Stream Double -> Stream Double -> Stream Double
tcoa sz vz = mux ((sz * vz) < 0) ((-sz) / vz) (-1)

dcpa :: Vect2 -> Vect2 -> Stream Double
dcpa s@(sx, sy) v@(vx, vy) = norm (sx + (tcpa s v) * vx, sy + (tcpa s v) * vy)

-------------------------
-- Well clear violation --
-------------------------

wcv :: (Vect2 -> Vect2 -> Stream Double) ->
       Vect2 -> Stream Double ->
       Vect2 -> Stream Double ->
       Stream Bool
wcv tvar s sz v vz = (horizontalWCV tvar s v) && (verticalWCV sz vz)

verticalWCV :: Stream Double -> Stream Double -> Stream Bool
verticalWCV sz vz =
  ((abs $ sz) <= zthr) ||
  (0 <= (tcoa sz vz) && (tcoa sz vz) <= tcoathr)

horizontalWCV :: (Vect2 -> Vect2 -> Stream Double) -> Vect2 -> Vect2 -> Stream Bool
horizontalWCV tvar s v =
  (norm s <= dthr) ||
  (((dcpa s v) <= dthr) && (0 <= (tvar s v)) && ((tvar s v) <= tthr))
```

```
-------------------
-- Local convexity --
-------------------

t1, t2, t3 :: Stream Double
t1 = extern "t1" Nothing
t2 = extern "t2" Nothing
t3 = extern "t3" Nothing

locallyConvex :: (Vect2 -> Vect2 -> Stream Double) -> Stream Bool
locallyConvex tvar = (0 <= t1 && t1 <= t2 && t2 <= t3)
  ==> not ( (wcv tvar (sx + t1*vx, sy + t1*vy) (sz + t1*vz) v vz)
      &&  (not $ wcv tvar (sx + t2*vx, sy + t2*vy) (sz + t2*vz) v vz)
      &&  (wcv tvar (sx + t3*vx, sy + t3*vy) (sz + t3*vz) v vz))

-------------
-- Spec --
-------------

spec :: Spec
spec = do

------------------------
-- Horizontal symmetry --
------------------------
  theorem "1a" (forall $ (tau s v)    ~= (tau (neg s) (neg v)))     arith
  theorem "1b" (forall $ (tcpa s v)   ~= (tcpa (neg s) (neg v)))    arith
  theorem "1c" (forall $ (taumod s v) ~= (taumod (neg s) (neg v)))  arith
  theorem "1d" (forall $ (tep s v)    ~= (tep (neg s) (neg v)))     arith

------------------------
-- Horizontal ordering --
------------------------
  theorem "2a" (forall $ ((s |*| v) < 0 && (norm s) > dthr && (dcpa s v) <= dthr)
    ==> ((tep s v) <= (taumod s v)))
     arith
  theorem "2b" (forall $ ((s |*| v) < 0 && (norm s) > dthr && (dcpa s v) <= dthr)
    ==> ((taumod s v) <= (tcpa s v)))
     arith
  theorem "2c" (forall $ ((s |*| v) < 0 && (norm s) > dthr && (dcpa s v) <= dthr)
    ==> ((tcpa s v) <= (tau s v)))
     arith

-------------
```

```
-- Symmetry --
---------------
  theorem "3a" (forall $
    (wcv tau s sz v vz)    == (wcv tau (neg s) (-sz) (neg v) (-vz)))
    arith
  theorem "3b" (forall $
    (wcv tcpa s sz v vz)   == (wcv tcpa (neg s) (-sz) (neg v) (-vz)))
    arith
  theorem "3c" (forall $
    (wcv taumod s sz v vz) == (wcv taumod (neg s) (-sz) (neg v) (-vz)))
    arith
  theorem "3d" (forall $
    (wcv tep s sz v vz)    == (wcv tep (neg s) (-sz) (neg v) (-vz)))
    arith

---------------
-- Inclusion --
---------------
  theorem "4i"   (forall $ (wcv tau s sz v vz)    ==>(wcv tcpa s sz v vz))
    arith
  theorem "4ii"  (forall $ (wcv tcpa s sz v vz)   ==>(wcv taumod s sz v vz ))
    arith
  theorem "4iii" (forall $ (wcv taumod s sz v vz)==>(wcv tep s sz v vz))
    arith

-------------------
-- Local convexity --
-------------------
  theorem "5a" (forall $ locallyConvex tcpa)        arith
  theorem "5b" (forall $ locallyConvex taumod)      arith
  theorem "5c" (forall $ locallyConvex tep)         arith
  theorem "6"  (P.not (forall $ locallyConvex tau)) arithSat

-------------------
-- Triggers --
-------------------
  trigger "alert_WCVtau"    (wcv tau s sz v vz)    []
  trigger "alert_WCVtcpa"   (wcv tcpa s sz v vz)   []
  trigger "alert_WCVtaumod" (wcv taumod s sz v vz) []
  trigger "alert_WCVtep"    (wcv tep s sz v vz)    []

-------------------------------------------------------------------------

arith :: Proof Universal
arith    = onlyValidity def { nraNLSat = True, debug = False }
```

```
arithSat :: Proof Existential
arithSat = onlySat      def { nraNLSat = True, debug = False }
```

B BNF Grammar

$$
\begin{array}{rcl}
\langle defs \rangle & ::= & (\langle def \rangle)^* \\
\\
\langle ctype \rangle & ::= & \texttt{Bool} \\
& | & \texttt{Int8} \\
& | & \texttt{Int16} \\
& | & \texttt{Int32} \\
& | & \texttt{Int64} \\
& | & \texttt{Word8} \\
& | & \texttt{Word16} \\
& | & \texttt{Word32} \\
& | & \texttt{Word64} \\
& | & \texttt{Float} \\
& | & \texttt{Double} \\
\\
\langle def \rangle & ::= & (\langle id \rangle \texttt{::Stream } \langle ctype \rangle)?\langle id \rangle \texttt{=} \langle spec \rangle \\
\\
\langle id \rangle & ::= & (\texttt{a}-\texttt{z})(\texttt{a}-\texttt{z}|\texttt{A}-\texttt{Z}|\texttt{0}-\texttt{9}|\_|\texttt{-}|\texttt{'})^* \\
\\
\langle string \rangle & ::= & \texttt{"}(\texttt{a}-\texttt{z}|\texttt{A}-\texttt{Z})(\texttt{a}-\texttt{z}|\texttt{A}-\texttt{Z}|\_|\texttt{0}-\texttt{9})^*_{\leq 30}\texttt{"} \\
\\
\langle stream \rangle & ::= & \langle valueList \rangle \texttt{++} \langle stream \rangle \\
& | & \langle funStream \rangle \\
\\
\langle valueList \rangle & ::= & [(\langle vbool \rangle)^*_,] \\
& | & [(\langle vint \rangle)^*_,] \\
& | & [(\langle vfloat \rangle)^*_,]
\end{array}
$$

$$
\begin{array}{lcl}
\langle vbool \rangle & ::= & \texttt{true} \\
& | & \texttt{false} \\
\\
\langle vint \rangle & ::= & [\texttt{+}|\texttt{-}](\texttt{0}-\texttt{9})^{+} \\
\\
\langle vfloat \rangle & ::= & \langle vint \rangle.(\texttt{0}-\texttt{9})^{+} \\
\\
\langle funStream \rangle & ::= & \langle externV \rangle\ \langle string \rangle\ \langle contextV \rangle \\
& | & \texttt{label}\ \langle string \rangle\ \langle funStream \rangle \\
& | & \texttt{externFun}\ \langle string \rangle\ \langle argList \rangle\ \langle contextF \rangle \\
& | & \langle externA \rangle\ \langle string \rangle\ \langle stream \rangle\ \langle int \rangle\ \langle contextA \rangle \\
& | & \langle op1 \rangle\ \langle funStream \rangle \\
& | & \langle funStream \rangle\ \langle op2Infix \rangle\ \langle funStream \rangle \\
& | & \langle op3 \rangle\ \langle funStream \rangle\ \langle funStream \rangle\ \langle funStream \rangle \\
& | & \langle dropStream \rangle
\end{array}
$$

$\langle externV \rangle$ ::= `extern`
| `externB`
| `externI8`
| `externI16`
| `externI32`
| `externI64`
| `externW8`
| `externW16`
| `externW32`
| `externW64`
| `externF`
| `externD`

$\langle externA \rangle$ ::= `externArray`
| `externArrayB`
| `externArrayI8`
| `externArrayI16`
| `externArrayI32`
| `externArrayI64`
| `externArrayW8`
| `externArrayW16`
| `externArrayW32`
| `externArrayW64`
| `externArrayF`
| `externArrayD`

$\langle contextV \rangle$ ::= `Nothing`
| `Just` $\langle stream \rangle$

$\langle contextF \rangle$ ::= `Nothing`
| `Just` $\langle stream \rangle$

$\langle contextA \rangle$ ::= `Nothing`
| `Just` $[(\langle valueList \rangle)^*_{,}]$

$\langle argList \rangle$ ::= $[$(`arg` $\langle stream \rangle$)$^*_{,}]$
| `[]`
| (`arg` $\langle stream \rangle$):$\langle argList \rangle$

$$
\begin{array}{lcl}
\langle dropStream \rangle & ::= & \langle id \rangle \\
& | & \texttt{constant}\ \langle value \rangle \\
& | & \texttt{drop}\ \langle int \rangle\ \langle stream \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle op1 \rangle & ::= & \texttt{not} \mid \texttt{abs} \mid \texttt{signum} \mid \texttt{complement} \\
& | & \texttt{recip} \mid \texttt{exp} \mid \texttt{sqrt} \\
& | & \texttt{log} \mid \texttt{sin} \mid \texttt{cos} \mid \texttt{tan} \\
& | & \texttt{asin} \mid \texttt{acos} \mid \texttt{atan} \\
& | & \texttt{sinh} \mid \texttt{cosh} \mid \texttt{tanh} \\
& | & \texttt{asinh} \mid \texttt{acosh} \mid \texttt{atanh} \\
& | & \texttt{cast} \mid \texttt{unsafeCast}
\end{array}
$$

$$
\begin{array}{lcl}
\langle op2Infix \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{`mod`} \mid \texttt{`div`} \\
& | & \texttt{/} \mid \texttt{**} \mid \texttt{`logBase`} \\
& | & \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{/=} \mid \texttt{>=} \mid \texttt{>} \\
& | & \texttt{||} \mid \texttt{\&\&} \mid \texttt{`xor`} \mid \texttt{==>} \\
& | & \texttt{.\&.} \mid \texttt{.|.} \mid \texttt{.\^{}.} \mid \texttt{.>>.} \mid \texttt{.<<.}
\end{array}
$$

$$
\begin{array}{lcl}
\langle op3 \rangle & ::= & \texttt{mux}
\end{array}
$$

C Typing Rules

$$\frac{s \text{ is a } \langle string \rangle \text{ token}}{\Gamma \vdash s : \texttt{String}} \quad (\textsc{stringConst})$$

$$\frac{\tau \in \text{inst}(\Gamma(s))}{\Gamma \vdash s : \tau} \quad (\textsc{inst})$$

$$\frac{\Gamma \vdash s : \texttt{Stream}\ \tau}{\Gamma \vdash \texttt{arg}\ s : \texttt{Arg}} \quad (\textsc{arg})$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{constant}\ x : \texttt{Stream}\ \tau} \quad (\textsc{valConst})$$

$$\frac{\Gamma \vdash i : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{Stream}\ \tau}{\Gamma \vdash \texttt{drop}\ i\ x : \texttt{Stream}\ \tau} \quad (\textsc{drop})$$

$$\frac{\Gamma \vdash ls : [a] \qquad \Gamma \vdash s : \texttt{Spec}\ a}{\Gamma \vdash ls +\!+ s : \texttt{Spec}\ a} \quad (\textsc{Append})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream}\ \tau}{\Gamma \vdash \texttt{label}\ s\ x : \texttt{Stream}\ \tau} \quad (\textsc{label})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream}\ \tau}{\Gamma \vdash \texttt{extern}\ s\ x : \texttt{Stream}\ \tau} \quad (\textsc{ext})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Bool}}{\Gamma \vdash \texttt{externB}\ s\ x : \texttt{Stream Bool}} \quad (\textsc{ext}\mathrm{B})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Int8}}{\Gamma \vdash \texttt{externI8}\ s\ x : \texttt{Stream Int8}} \quad (\textsc{ext}\mathrm{I8})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Int16}}{\Gamma \vdash \texttt{externI16}\ s\ x : \texttt{Stream Int16}} \quad (\textsc{ext}\mathrm{I16})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Int32}}{\Gamma \vdash \texttt{externI32}\ s\ x : \texttt{Stream Int32}} \quad (\textsc{ext}\mathrm{I32})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Int64}}{\Gamma \vdash \texttt{externI64}\ s\ x : \texttt{Stream Int64}} \quad (\textsc{ext}\mathrm{I64})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Word8}}{\Gamma \vdash \texttt{externW8}\ s\ x : \texttt{Stream Word8}} \quad (\textsc{ext}\mathrm{W8})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Word16}}{\Gamma \vdash \texttt{externW16}\ s\ x : \texttt{Stream Word16}} \quad (\textsc{ext}\mathrm{W16})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Word32}}{\Gamma \vdash \texttt{externW32}\ s\ x : \texttt{Stream Word32}} \quad (\textsc{ext}\mathrm{W32})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Word64}}{\Gamma \vdash \texttt{externW64}\ s\ x : \texttt{Stream Word64}} \quad (\textsc{ext}\mathrm{W64})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Float}}{\Gamma \vdash \texttt{externF}\ s\ x : \texttt{Stream Float}} \quad (\textsc{ext}\textsc{Float})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash x : \texttt{Stream Double}}{\Gamma \vdash \texttt{externD}\ s\ x : \texttt{Stream Double}} \quad (\textsc{ext}\textsc{Double})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash a : \texttt{[Arg]} \qquad \Gamma \vdash x : \texttt{Stream}\ \tau}{\Gamma \vdash \texttt{externFun}\ s\ a\ x : \texttt{Stream}\ \tau} \quad (\textsc{extFun})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[}\tau\texttt{]]}}{\Gamma \vdash \texttt{externArray}\ s\ i\ m\ x : \texttt{Stream}\ \tau} \quad (\textsc{extA})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Bool]]}}{\Gamma \vdash \texttt{externArrayB}\ s\ i\ m\ x : \texttt{Stream Bool}} \quad (\textsc{extAB})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Int8]]}}{\Gamma \vdash \texttt{externArrayI8}\ s\ i\ m\ x : \texttt{Stream Int8}} \quad (\textsc{extAI8})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Int16]]}}{\Gamma \vdash \texttt{externArrayI16}\ s\ i\ m\ x : \texttt{Stream Int16}} \quad (\textsc{extAI16})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Int32]]}}{\Gamma \vdash \texttt{externArrayI32}\ s\ i\ m\ x : \texttt{Stream Int32}} \quad (\textsc{extAI32})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Int64]]}}{\Gamma \vdash \texttt{externArrayI64}\ s\ i\ m\ x : \texttt{Stream Int64}} \quad (\textsc{extAI64})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Word8]]}}{\Gamma \vdash \texttt{externArrayW8}\ s\ i\ m\ x : \texttt{Stream Word8}} \quad (\textsc{extAW8})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Word16]]}}{\Gamma \vdash \texttt{externArrayW16}\ s\ i\ m\ x : \texttt{Stream Word16}} \quad (\textsc{extAW16})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Word32]]}}{\Gamma \vdash \texttt{externArrayW32}\ s\ i\ m\ x : \texttt{Stream Word32}} \quad (\textsc{extAW32})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Word64]]}}{\Gamma \vdash \texttt{externArrayW64}\ s\ i\ m\ x : \texttt{Stream Word64}} \quad (\textsc{extAW64})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Float]]}}{\Gamma \vdash \texttt{externArrayF}\ s\ i\ m\ x : \texttt{Stream Float}}\ (\textsc{extAFloat})$$

$$\frac{\Gamma \vdash s : \texttt{String} \qquad \Gamma \vdash i : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad \Gamma \vdash m : \texttt{Integer} \qquad \Gamma \vdash x : \texttt{[[Double]]}}{\Gamma \vdash \texttt{externArrayD}\ s\ i\ m\ x : \texttt{Stream Double}}\ (\textsc{extADouble})$$

$$\frac{\Gamma \vdash x : \texttt{Stream Bool} \qquad op \in \{\texttt{not}\}}{\Gamma \vdash op\ x : \texttt{Stream Bool}}\ (\textsc{op1Bool})$$

$$\frac{\Gamma \vdash x : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{complement}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \tau}\ (\textsc{op1Bitwise})$$

$$\frac{\Gamma \vdash x : \texttt{Integral}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{abs}, \texttt{signum}\}}{\Gamma \vdash op\ x : \texttt{Stream}\ \tau}\ (\textsc{op1Num})$$

$$\frac{\Gamma \vdash x : \texttt{Stream Bool} \qquad \Gamma \vdash y : \texttt{Stream Bool} \qquad op \in \{\texttt{||}, \texttt{\&\&}, \texttt{`xor`}, \texttt{==>}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream Bool}}\ (\textsc{op2Bool})$$

$$\frac{\Gamma \vdash x : \texttt{Integral}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Integral}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{`mod`}, \texttt{`div`}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau}\ (\textsc{op2Integral})$$

$$\frac{\Gamma \vdash x : \texttt{Fractional}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Fractional}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{/}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau}\ (\textsc{op2Fractional})$$

$$\frac{\Gamma \vdash x : \texttt{Floating}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Floating}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{**}, \texttt{`logBase`}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau}\ (\textsc{op2Floating})$$

$$\frac{\Gamma \vdash x : \texttt{Num}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Num}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau}\ (\textsc{op2Num})$$

$$\frac{\Gamma \vdash x : \texttt{Eq}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Eq}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{==}, \texttt{/=}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau}\ (\textsc{op2Eq})$$

$$\frac{\Gamma \vdash x : \texttt{Ord}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Ord}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{<}, \texttt{<=}, \texttt{>=}, \texttt{>}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau}\ (\textsc{op2Ord})$$

$$\frac{\Gamma \vdash x : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad op \in \{\texttt{.\&.}, \texttt{.|.}, \texttt{.\^{}.}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau} \quad (\textsc{op2Bitwise})$$

$$\frac{\Gamma \vdash x : \texttt{Bits}\ \tau \Rightarrow \texttt{Stream}\ \tau \qquad \Gamma \vdash y : \texttt{Integral}\ \tau_1 \Rightarrow \texttt{Stream}\ \tau_1 \qquad op \in \{\texttt{.>>.}, \texttt{.<<.}\}}{\Gamma \vdash op\ x\ y : \texttt{Stream}\ \tau} \quad (\textsc{op2BwShift})$$

$$\frac{\Gamma \vdash x : \texttt{Stream Bool} \qquad \Gamma \vdash y : \texttt{Stream}\ \tau \qquad \Gamma \vdash z : \texttt{Stream}\ \tau \qquad op \in \{\texttt{mux}\}}{\Gamma \vdash op\ x\ y\ z : \texttt{Stream}\ \tau} \quad (\textsc{op3})$$