Fuzzing Channel-Based Concurrency Runtimes using Types and Effects

QUENTIN STIÉVENART, Vrije Universiteit Brussel, Belgium
MAGNUS MADSEN, Aarhus University, Denmark

Modern programming languages support concurrent programming based on channels and processes. Channels enable synchronous and asynchronous message-passing between independent light-weight processes making it easy to express common concurrency patterns.

The implementation of channels and processes in compilers and language runtimes is a difficult task that relies heavily on traditional and error-prone low-level concurrency primitives, raising concerns about correctness and reliability.

In this paper, we present an automatic program generation technique to test such programming language implementations. We define a type and effect system for programs that communicate over channels and where every execution is guaranteed to eventually terminate. We can generate and run such programs, and if a program fails to terminate, we have found a bug in the programming language implementation.

We implement such an automatic program generator and apply it to Go, Kotlin, Crystal, and Flix. We find two new bugs in Flix, and reproduce two bugs; one in Crystal and one in Kotlin.

CCS Concepts: • **Theory of computation** → **Program semantics**; • **Software and its engineering** → **Compilers**; **Concurrent programming structures**.

Additional Key Words and Phrases: automatic test generation, channels and processes, effect systems

ACM Reference Format:
Quentin Stiévenart and Magnus Madsen. 2020. Fuzzing Channel-Based Concurrency Runtimes using Types and Effects. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 186 (November 2020), 27 pages. https://doi.org/10.1145/3428254

1 INTRODUCTION

Modern programming languages, such as Go, Kotlin, Crystal, and Flix, support concurrent programming with channels and processes. In these languages, processes are independent units of execution that are light-weight, requiring significantly less memory than operating system threads. Channels enable processes to communicate synchronously and asynchronously by message-passing without the use of explicit locks. Processes can spawn sub-processes, create channels, receive and send messages on channels, and select on a set of channels. For un-buffered channels, when a process performs a send (put) operation, it must wait until another process is ready to receive (get) the message. Similarly, if a process performs a get, it must wait until another process performs a put. For buffered channels, the situation is similar, except that each channel has an internal capacity, and put operations do not block until the capacity has been reached. The select operation allows a process to receive from (or send to) a single channel in a set of channels, selecting the first channel that is ready to send or receive. The select operation is strictly more expressive than a get (or

Authors' addresses: Quentin Stiévenart, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, quentin.stievenart@vub.be; Magnus Madsen, Aarhus University, Åbogade 34, Aarhus, Denmark, magnusm@cs.au.dk.

2475-1421/2020/11-ART186
https://doi.org/10.1145/3428254

put) operation since it allows waiting on multiple channels simultaneously. These concurrency primitives make it easy to express common concurrency patterns, such as producer-consumer, fork-join, and load balancers.

While channels and processes simplify concurrent programming for users of languages that support them, their implementation in compilers and runtimes is a difficult task. Such implementations typically rely on shared memory concurrency protected by locks. Thus, race conditions and deadlocks lurk in the shadows. The select operation significantly complicates implementations since it must be able to operate on multiple channels simultaneously while still ensuring proper mutual exclusion. For example, locks taken on multiple channels must always be acquired in the same global order to prevent a deadlock between multiple select operations.

We want to automatically test implementations of channel and process-based concurrency primitives in compilers and runtimes. Our key idea is to automatically generate communicating programs where every execution must terminate. If such a program fails to terminate, we have found a bug in the implementation. We can now state the problem we address in this paper:

How to generate *interesting* and *terminating* programs that use channels and processes?

By interesting, we mean programs that spawn processes and communicate on channels, and are likely to expose bugs in the language implementation of the channel and process primitives.

Inspired by Pałka et al. [2011] and Midtgaard et al. [2017], we take the following approach: We define a type and effect system for a calculus with channels and processes. We then design a sub-language of effects whose programs are guaranteed to terminate. To generate interesting programs, we define a collection of heuristics that capture communication behavior which must terminate. The heuristics try to cause contention on channels, i.e. to overlap communication on the same channels from multiple processes, with the hope that such behavior will expose races or deadlocks in the programming language runtime.

The result of this work is an automatic and sound technique for the generation of interesting and terminating concurrent programs that communicate over channels. As we will see, such programs can be used to find bugs in programming language implementations. These bugs are potentially very serious since they affect all users of the language. We believe that our work is the first that attempts to find such semantically deep bugs in compilers and runtimes for programming languages.

In summary, the paper makes the following contributions:

- (**Terminating Effects**) We define a sub-language of effects for a calculus with channels and processes, and prove that programs with these effects always terminate.
- (**Implementation & Evaluation**) We implement an automatic program generator, based on the effect language, and use it to generate communicating Flix, Crystal, Kotlin, and Go programs that always terminate. We run the program generator and we find two previously unknown bugs in Flix, and we reproduce two bugs; one in Crystal and one in Kotlin.

2 MOTIVATION

We begin with a brief introduction to channels and processes before we present our automatic program generation technique. We will use the Flix programming language for the examples, but the ideas are equally applicable to other languages. In fact, a key observation is that Go, Kotlin, Crystal, and Flix all share the same semantics for channels and processes. Consequently, our technique is applicable to all of these languages.

2.1 Channel and Process-Based Concurrency

We can create a new process with the spawn expression:

```
def main(): Unit = spawn (1 + 2)
```

The new process computes $1 + 2$ and discards the result. Meanwhile, the main process continues execution. The `spawn` expression returns `Unit` and does not provide any means for communication between the original process and the spawned process.

We use channels to communicate between processes. A channel comes in one of two variants: buffered or un-buffered. A buffered channel has a capacity set at creation time. The capacity specifies the number of messages that the channel can hold. If a process attempts to put a message into a buffered channel that is full, then the process is blocked until space becomes available. If, on the other hand, a process attempts to get a message from an empty channel, the process is blocked until a message is put into the channel. An un-buffered channel works like a buffered channel of size zero; for a get and a put to happen both processes must rendezvous (synchronize) such that the message can be passed directly from the sender to the receiver. Consider the program:

```
def plus(c: Channel[Int]): Unit = c <- (21 + 21); ()
def main(): Int =
    let c = chan Int 0;
    spawn plus(c);
    <- c
```

Here `main` constructs a new channel c, spawns a process to run `plus`, and waits for a message on c with the get expression `<- c`. In the spawned process, `plus` computes $21 + 21$ and sends the result on the channel c with the put expression `c <- (21 + 21)`. Hence `main` returns 42.

We can use the `select` expression to receive a message from *one* ready channel out of a set of channels. Consider the program:

```
def meow(c: Channel[String]): Unit = c <- "Meow!"; ()
def woof(c: Channel[String]): Unit = c <- "Woof!"; ()
def main(): String =
    let c1 = chan 1;
    let c2 = chan 1;
    spawn meow(c1);
    spawn woof(c2);
    select {
        case m <- c1 => m
        case m <- c2 => m
    }
```

Here `main` constructs two channels, spawns two processes to run the `meow` and `woof` functions, and then selects on both channels. The result is that `main` returns "Meow!" or "Woof!" depending on which of the two processes is first able to send its message. The `select` expression is strictly more powerful than the get and put primitives since it is able to wait on multiple channels simultaneously.

2.2 Technique in Action

Figure 1 shows a legal Flix program generated by our technique. The program begins by creating three un-buffered channels c_1, c_2, and c_3. Next, it spawns three new processes that each perform a single put operation on one of the three channels. Meanwhile, the main process performs a select. In the first case, the select receives an element from c_2 and then the body waits for an element from c_1 and then from c_3. Thus, if this case is taken, an element is received from each of the three channels and all processes terminate. The second case is conceptually similar, except that to receive an element from c_3 another select is used with a duplicate case. Finally, the third case is a combination of the two previous cases, but it also receives exactly one value from each of the channels c_1, c_2, and c_3. Thus, all processes must always terminate.

Flix
```
def main(): Unit = {
  let c1 = chan Unit 0;
  let c2 = chan Unit 0;
  let c3 = chan Unit 0;
  spawn { c1 <- () };
  spawn { c3 <- () };
  spawn { c2 <- () };
  select {
    case _ <- c2 => <- c1 ; <- c3
    case _ <- c2 => <- c1;
      select {
        case _ <- c3 => ()
        case _ <- c3 => ()
      }
    case _ <- c1 => <- c2;
      select {
        case _ <- c3 => ()
        case _ <- c3 => ()
      }
  }
}
```

Fig. 1. A Generated Program that Deadlocks in Flix.

However, if we compile and run this program with Flix (version 0.5.0) it sometimes fails to terminate! The JVM reports:

```
Found one Java-level deadlock:
=============================
"main":
  waiting for ownable synchronizer ..., (...),
  which is held by "Thread-59"
"Thread-59":
  waiting for ownable synchronizer ..., (...),
  which is held by "main"
```

The program has deadlocked! We have discovered a bug in the runtime implementation of channels and processes in Flix.

2.3 Automatic Program Generation

How can we automatically generate programs such as the one in Figure 1? We could use the grammar of the language to generate random, but syntactically correct programs, as done with grammar-based fuzzing [Godefroid et al. 2008] or parser-directed fuzzing [Mathis et al. 2019]. But such programs might fail to type check, and even if they can be compiled, how do we know whether they are executed correctly?

Instead, relying on the established techniques of Pałka et al. [2011] and Midtgaard et al. [2017], we take the following approach: We define a type and effect system for a language with channels and processes. A backwards reading of the type rules allows us to generate expressions that are typeable. The effect system captures the communication behavior of an expression. Our idea is to generate programs that always terminate. Hence, if a generated program fails to terminate

(or crashes), we have found a bug. However, not all well-typed and well-effected expressions are guaranteed to terminate. Thus, we define a sub-language of effects whose corresponding programs are guaranteed to terminate. The definition of the sub-language is inspired by common concurrency patterns and is conservative, but not exhaustive: there are many terminating programs which are not captured by this sub-language. Even so, we shall argue that the sub-language contains many interesting programs. The sub-language tries to capture programs whose execution cause contention on channels, i.e. causes multiple simultaneous operations on the same channels, with the hope of discovering races or deadlocks.

In order to reason about program termination, we need an operational semantics. Given this semantics, we can prove that a class of program terminates. Once we have defined termination of programs, we can reason about termination of effects: given a type and an effect, we can prove that *all* programs with this type and effect terminate. The problem of generating terminating programs is therefore reduced to the problem of generating terminating effects.

By working at the effect-level, following the approach of Pałka et al. [2011] and Midtgaard et al. [2017], we can reason at a higher-level than the language syntax which has two benefits:

- We can generate terminating programs for multiple target languages.
- We can generate a wider range of programs, which are semantically different, but are guaranteed to terminate.

To understand why the type and effect system is useful, let us consider some examples.

The following is a syntactically valid Flix program:

```
let c = chan 0; spawn { c <- "Hello World" }; if (<- c) 1 else 2
```

While such a program is straightforward to generate from the grammar of the Flix programming language, it is ill-typed and hence it cannot be compiled with the Flix compiler. An approach that generates programs solely based on the grammar of a programming language is likely to produce many ill-typed programs which cannot be compiled nor executed, and hence are not useful for uncovering bugs. By using not only the grammar of the language, but also a backwards reading of its type system, we can ensure that we only generate well-typed programs.

For example, the following is a syntactically and well-typed Flix program:

```
let c = chan 0; <- c;
```

While this program compiles and can be executed, it does not terminate. The problem is that the program tries to receive a value from the channel c, but a value is never sent on c. We can generate such programs, compile them, and execute them, but we cannot know whether they are supposed to terminate or not. If we do not know what the result of an execution should be, we cannot determine if the execution was correct. To resolve this, we can use an effect system to describe the communication behavior of a program.

For example, the effect $\textsc{Chan}(c); \textsc{Spawn}(\textsc{Put}(c))$; $\textsc{Get}(c)$ describes the program:

```
let c = chan 0; spawn { c <- 1 }; <- c
```

But it also describes many other programs, for example:

```
let c = chan 0; spawn { c <- 2 }; <- c
let c = chan 0; spawn { c <- 1 + 2 }; <- c
let c = chan 0; spawn { c <- "Hello World" }; <- c; 1 + 2
let c = chan 0; let f = () -> { c <- "Hello World" }; spawn f(); <- c
...
```

Working at the type and effect level, we can abstract over all such programs, and prove once and for all that they terminate.

$$
\begin{array}{rcll}
v \in \mathit{Val} & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid c & \\
\\
e \in \mathit{Exp} & ::= & x & [\textsc{Var}] \\
& \mid & v & [\textsc{Val}] \\
& \mid & \mathsf{let}\, x := e_1\, \mathsf{in}\, e_2 & [\textsc{Let}] \\
& \mid & \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 & [\textsc{IfThenElse}] \\
& \mid & e_1; e_2 & [\textsc{Sequence}] \\
& \mid & \mathsf{channel}^c\, t & [\textsc{NewChannel}] \\
& \mid & \leftarrow e & [\textsc{GetChannel}] \\
& \mid & e_1 \leftarrow e_2 & [\textsc{PutChannel}] \\
& \mid & \mathsf{select}\, \{\, \overrightarrow{sr} \,\} & [\textsc{Select}] \\
& \mid & \mathsf{spawn}^p\, \{\, e \,\} & [\textsc{Spawn}] \\
\\
sr \in \mathit{SelectRule} & ::= & \mathsf{case}\, x := \leftarrow y \Rightarrow e & [\textsc{SelectGet}] \\
& \mid & \mathsf{case}\, x \leftarrow y \Rightarrow e & [\textsc{SelectPut}] \\
\\
x, y \in \mathit{Variables} & & \text{is a set of variable names.} & \\
c \in \mathit{Channels} & & \text{is a set of channel names.} & \\
p \in \mathit{Processes} & & \text{is a set of process names.} &
\end{array}
$$

Fig. 2. Syntax of λ_{chan}.

3 A CHANNEL AND PROCESS CALCULUS

To present our approach, we introduce λ_{chan}, a minimal calculus with support for channels and processes. The calculus is based on Nielson and Nielson [1999] extended with the select construct. We have deliberately kept the calculus simple; it has no recursion and no functions. We are interested in the communication behavior of programs that terminate, and hence the lack of recursion is not a concern. The design of λ_{chan} has been chosen such that all programs in λ_{chan} are straightforward to translate into equivalent Go, Kotlin, Crystal, and Flix programs. A reader who is already familiar with these types of calculi and their type and effect systems may safely skip to Section 4.

3.1 Syntax of λ_{chan}

Figure 2 shows the syntax of λ_{chan}. The values of the language are unit, booleans, and channels. The language consists of sequential and parallel expressions. The sequential expressions include the usual constructs: variables, values, let-bindings, if-then-elses, and sequence expressions. The parallel expressions allow channel and process creation, and include communication primitives: The new channel expression $\mathsf{channel}^c\, t$ creates a channel named c with elements of type t. In λ_{chan}, all channels can be given a static name since there is no recursion, looping, or functions. We restrict ourselves to un-buffered channels for λ_{chan}. The get channel expression $\leftarrow e$ receives a value from the channel e. Similarly, the put channel expression $e_1 \leftarrow e_2$ sends the value of e_2 to the channel e_1. The select expression performs a non-deterministic choice between multiple get and put operations on multiple channels: a single branch of the select expression will be executed, depending on which channel is ready for sending or receiving values. The spawn^p expression creates a process named p. Similarly to channels, processes can also be given a static name.

We use $\overrightarrow{a}$ as a short hand for $a_1, \cdots, a_n$ and we use a_i to refer to the i'th element of $\overrightarrow{a}$.

$$
\begin{array}{rcl}
E \in \mathit{Contexts} & ::= & \bullet \\
& | & \mathsf{let}\, x := E \,\mathsf{in}\, e \\
& | & \mathsf{if}\, \mathsf{E}\, \mathsf{then}\, e\, \mathsf{else}\, e \mid E; e \\
& | & \leftarrow E \mid E \leftarrow e \mid v \leftarrow E
\end{array}
$$

Fig. 3. Evaluation Contexts of λ_{chan}.

$$
\mathsf{let}\, x := v \,\mathsf{in}\, e \longrightarrow e[x \mapsto v] \qquad v; e_2 \longrightarrow e_2 \qquad \mathsf{if}\, \mathtt{true}\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \longrightarrow e_2
$$

$$
\mathsf{if}\, \mathtt{false}\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \longrightarrow e_3
$$

Fig. 4. Sequential Evaluation Rules of λ_{chan}.

$$
P \in \mathit{Configuration} \quad ::= \quad \mathit{Processes} \longrightarrow \mathit{Exp}
$$

Fig. 5. Configuration of λ_{chan}.

Absence of Functions, Looping, and Recursion. The calculus does not include any constructs for defining or calling functions, nor for recursion, or any other looping construct. This is a deliberate design choice motivated by the following observation: If there exists a program that can crash or deadlock due to a bug in the concurrency runtime then it implies that there exists a finite execution of that program leading to the crash or deadlock. This execution can be written as a finite sequence of channel operations (creating, sending, receiving, and selecting).

While it is not impossible that a concurrency bug could arise through a unique combination of channel operations and e.g., recursion, we think that such a situation is unlikely. We leave it as interesting future work to explore richer calculi to determine if such bugs occur in practice.

3.2 Semantics

The semantics of λ_{chan} is defined in two parts: sequential evaluation is concerned with the sequential constructs of the languages (conditionals, bindings, and sequences) and parallel evaluation is concerned with the constructs that act over multiple processes (communications and creation of channels and processes).

3.2.1 Sequential Evaluation. The evaluation rules for the sequential part of the language are defined in terms of the evaluation contexts shown in Figure 3 together with the reduction rules shown in Figure 4. The evaluation contexts are *only* used for *sequential* evaluation. The parallel evaluation rules are defined using the sequential evaluation rules, as we shall see shortly. A sequential evaluation step from an expression e to an expression e' is denoted $e \longrightarrow e'$. The rules are standard.

3.2.2 Parallel Evaluation. For parallel evaluation, we define a *configuration*, shown in Figure 5, as a (partial) map from process names to expressions. Intuitively, the map tracks the current processes and their expressions. We write $P[p : e]$ to denote the process map P extended with a process p that evaluates expression e. The parallel evaluation rules act on such configurations and are defined in Figure 6. A parallel evaluation step from configuration P to configuration P' is denoted $P \Longrightarrow P'$. The transitive closure of $\Longrightarrow$ is denoted $\Longrightarrow^{\star}$. The (E-Process) rule states that if a sequential evaluation rule can be applied to an existing process then the whole configuration can take the corresponding

$$\text{(E-Process)}\ \frac{e_1 \rightarrow e_2}{P[p : E[e_1]] \Rightarrow P[p : E[e_2]]} \qquad \text{(E-Channel)}\ \frac{}{P[p : E[\mathsf{channel}^c\, t]] \Rightarrow P[p : E[c]]}$$

$$\text{(E-Sync)}\ \frac{}{P[p_1 : E_1[\leftarrow c]][p_2 : E_2[c \leftarrow v]] \Rightarrow P[p_1 : E_1[v]][p_2 : E_2[()]]}$$

$$\text{(E-Spawn)}\ \frac{}{P[p : E[\mathsf{spawn}^{p_0}\ \{e_0\}]] \Rightarrow P[p : E[()]][p_0 : e_0]}$$

$$\text{(E-Select-Select)}\ \frac{i, j \in \mathbb{N} \qquad sr_i = \mathsf{case}\ x := \leftarrow c \Rightarrow e_i \qquad sr'_j = \mathsf{case}\ c \leftarrow v \Rightarrow e_j}{P[p_1 : E_1[\mathsf{select}\ \{\overrightarrow{sr}\}]][p_2 : E_2[\mathsf{select}\ \{\overrightarrow{sr'}\}]] \Rightarrow P[p_1 : E_1[e_i[x \mapsto v]]][p_2 : E_2[e_j]]}$$

$$\text{(E-Select-Get)}\ \frac{i \in \mathbb{N} \qquad sr_i = \mathsf{case}\ x := \leftarrow c \Rightarrow e_i}{P[p_1 : E_1[\mathsf{select}\ \{\overrightarrow{sr}\}]][p_2 : E_2[c \leftarrow v]] \Rightarrow P[p_1 : E_1[e_i[x \mapsto v]]][p_2 : E_2[()]]}$$

$$\text{(E-Select-Put)}\ \frac{i \in \mathbb{N} \qquad sr_i = \mathsf{case}\ c \leftarrow v \Rightarrow e_i}{P[p_1 : E_1[\mathsf{select}\ \{\overrightarrow{sr}\}]][p_2 : E_2[\leftarrow c]] \Rightarrow P[p_1 : E_1[e_i]][p_2 : E_2[v]]}$$

Fig. 6. Parallel Evaluation Rules of λ_{chan}.

step where the sequential rule has been applied to that process. The (E-Channel) rule creates a new channel using the static channel name. The (E-Spawn) rule creates a new process with the static process name p and adds it to the process map. The (E-Sync) rule synchronizes two processes, where process p_1 is performing a get operation on a channel c, and process p_2 is performing a put operation on the same channel. Note that this rule is non-deterministic: more than two processes may be trying to synchronize on the same channel. The (E-Select-Get) (resp. (E-Select-Put)) rules synchronize a process p_1 that is selecting over multiple channels with a process p_2 getting (resp. putting) a value on one of the channels on which p_1 is selecting. The process p_1 then proceeds to evaluate the body of the select branch that has been selected. The (E-Select-Select) proceeds similarly when two processes are selecting on the same channel, where a process p_1 is trying to receive from the channel c and process p_2 is trying to send over the channel.

3.3 Types

The types of λ_{chan} are shown in Figure 7. The types are very simple. The language has the unit type for the unit value, the bool type for the booleans true and false, and the channel type channel(t, c) for the channel with elements of type t and channel name c.

3.4 Effects

The effects of λ_{chan} are shown in Figure 8. An effect describes the communication behavior of an expression. The empty effect ϵ denotes the absence of communication over any channel and the absence of channel or process creation. The $\textsc{Chan}(c)$ effect denotes the creation of a channel with name c. The $\textsc{Get}(c)$ effect denotes receiving a value from the channel c, and $\textsc{Put}(c)$ denotes

$$t \in \mathit{Types} \quad ::= \quad \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{channel}(t, c)$$

Fig. 7. Types of λ_{chan}.

$$\begin{array}{rcl} \varphi \in \mathit{Effects} & ::= & \epsilon \mid \textsc{Chan}(c) \\ & \mid & \textsc{Get}(c) \mid \textsc{Put}(c) \\ & \mid & \textsc{Spawn}^{p}(\varphi) \mid \varphi_1 \,;\, \varphi_2 \\ & \mid & \varphi_1 \,+\, \varphi_2 \mid \varphi_1^{sr} \oplus \varphi_2^{sr} \\ \\ \varphi^{sr} \in \mathit{SelectEffect} & ::= & \textsc{SelGet}(c, \varphi) \mid \textsc{SelPut}(c, \varphi) \end{array}$$

Fig. 8. Effects of λ_{chan}.

sending a value on the channel c. The $\textsc{Spawn}^{p}(\varphi)$ effects denotes the creation of a process with name p that itself has the effect φ. The sequence effect $\varphi_1 \,;\, \varphi_2$ denotes that the φ_1 effect happens first, followed by the φ_2 effect. The choice effect $\varphi_1 \,+\, \varphi_2$ denotes that one of the φ_1 and φ_2 effects must happen. The choice does *not* have to be "fair". The effect may always be φ_1 or φ_2. For example, the expression $\leftarrow \mathsf{c1}$ may be given the effect $\textsc{Get}(c_1) + \textsc{Put}(c_2)$ despite the fact that the expression will *never* perform a put on channel c_2. The select effect $\varphi_1^{sr} \oplus \varphi_2^{sr}$ denotes a compound effect where each φ_i^{sr} is of the form $\textsc{SelGet}(c_i, \varphi_i)$ or $\textsc{SelPut}(c_i, \varphi_i)$ for some effects φ_i. The select effect describes a non-deterministic choice between expressions with effect φ_1^{sr} and φ_2^{sr}. The difference between the choice effect and the select effect is the following: An expression with the choice effect $\textsc{Get}(c_1) + \textsc{Get}(c_2)$ may be stuck *even if there is another process ready to put on the channel* c_1. On the other hand, an expression with the select effect $\textsc{SelGet}(c_1, \epsilon) \oplus \textsc{SelGet}(c_2, \epsilon)$ cannot be stuck if there is a process ready to put on channel c_1.

We define the *dual effect* of a get or put effect as:

$$\overline{\textsc{Get}(c)} = \textsc{Put}(c) \quad \overline{\textsc{Put}(c)} = \textsc{Get}(c)$$

3.5 Type and Effect System

The type and effect rules for λ_{chan} are shown in Figure 9. A typing judgment is of the form $\Gamma \vdash e : t \,\&\, \varphi$, meaning that under environment Γ expression e has type t and produces effect φ.

The (T-Var), (T-Unit), (T-True), and (T-False) rules are straightforward. The (T-Let) rule types variable bindings, sequencing the effects of its two sub-expressions. The (T-Seq) rule is similar. The (T-If) rule types if-then-else expressions, producing first the effect generated by the condition (φ_1), followed by a choice between the effects of both branches ($\varphi_2 \,+\, \varphi_3$). The (T-Chan) rule types the creation of a channel with name c, producing a $\textsc{Chan}(c)$ effect. The (T-Get) (resp. (T-Put)) rule types the receiving of a value from a channel (resp. the sending of a value over a channel) with name c, generating a $\textsc{Get}(c)$ (resp. $\textsc{Put}(c)$) effect. The (T-Select) rule types a select statement, using rules (TSelect-Get) and (T-Select-Put) to type each select rule. The type of each select case has to match, and their effects are combined with the $\oplus$ operator. The (T-Spawn) rule types a spawn statement, wrapping the effect of the spawned expression inside a spawn effect.

Unlike the calculus of Nielson and Nielson [1999], our calculus does not include type and effect sub-typing. The type and effect sub-typing of Nielson and Nielson [1999] introduces sub-types and sub-effects for functions and for conditionals, while other sub-typing and sub-effecting rules are classical reflexive, transitive, and distributive rules. In the case of functions, this is not relevant

$$\Gamma \vdash x : \Gamma(x) \,\&\, \epsilon \text{ (T-Var)} \qquad \Gamma \vdash () : \text{unit} \,\&\, \epsilon \text{ (T-Unit)} \qquad \Gamma \vdash \text{true} : \text{bool} \,\&\, \epsilon \text{ (T-True)}$$

$$\Gamma \vdash \text{false} : \text{bool} \,\&\, \epsilon \text{ (T-False)} \qquad \frac{\Gamma \vdash e_1 : t_1 \,\&\, \varphi_1 \qquad \Gamma[x \mapsto t_1] \vdash e_2 : t_2 \,\&\, \varphi_2}{\Gamma \vdash \text{let } x := e_1 \text{ in } e_2 : t_2 \,\&\, \varphi_1 \,;\, \varphi_2} \text{ (T-Let)}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \,\&\, \varphi_1 \qquad \Gamma \vdash e_2 : t \,\&\, \varphi_2 \qquad \Gamma \vdash e_3 : t \,\&\, \varphi_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \,\&\, \varphi_1 \,;\, (\varphi_2 \,+\, \varphi_3)} \text{ (T-If)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \,\&\, \varphi_1 \qquad \Gamma \vdash e_2 : t_2 \,\&\, \varphi_2}{\Gamma \vdash e_1; e_2 : t_2 \,\&\, \varphi_1 \,;\, \varphi_2} \text{ (T-Seq)} \qquad \frac{\Gamma \vdash e : t \,\&\, \varphi}{\Gamma \vdash \text{spawn } \{\, e \,\} : \text{unit} \,\&\, \textsc{Spawn}(\varphi)} \text{ (T-Spawn)}$$

$$\Gamma \vdash \text{channel}^c\ t : \text{channel}(t, c) \,\&\, \text{chan}(c) \text{ (T-Chan)} \qquad \frac{\Gamma \vdash e : \text{channel}(t, c) \,\&\, \varphi}{\Gamma \vdash \leftarrow e : t \,\&\, \varphi \,;\, \textsc{Get}(c)} \text{ (T-Get)}$$

$$\frac{\Gamma \vdash e_1 : \text{channel}(t, c) \,\&\, \varphi_1 \qquad \Gamma \vdash e_2 : t \,\&\, \varphi_2}{\Gamma \vdash e_1 \leftarrow e_2 : \text{unit} \,\&\, \varphi_1 \,;\, \varphi_2 \,;\, \textsc{Put}(c)} \text{ (T-Put)} \qquad \frac{\forall i.\, \Gamma \vdash sr_i : t \,\&\, \varphi_i^{sr}}{\Gamma \vdash \text{select}\{\, \overrightarrow{\text{sr}} \,\} : t \,\&\, \varphi_1^{sr} \oplus \cdots \oplus \varphi_n^{sr}} \text{ (T-Select)}$$

$$\frac{\Gamma(y) = \text{channel}(t', c) \qquad \Gamma[x \mapsto t'] \vdash e : t \,\&\, \varphi}{\Gamma \vdash \text{case } x := \leftarrow y \Rightarrow e : t \,\&\, \textsc{SelGet}(c, \varphi)} \text{ (T-Select-Get)}$$

$$\frac{\Gamma(x) = \text{channel}(t', c) \qquad \Gamma(y) = t' \qquad \Gamma \vdash e : t \,\&\, \varphi}{\Gamma \vdash \text{case } x \leftarrow y \Rightarrow e : t \,\&\, \textsc{SelPut}(c, \varphi)} \text{ (T-Select-Put)}$$

Fig. 9. Type and effect system for λ_{chan}.

to our setting as we do not model functions in the calculus. In the case of conditionals, one could extend the calculus presented here to have e.g., effect b_1 be a sub-effect of $b_1 + b_2$. Instead of doing so, we will present a more general approach in our effect generation rules of Section 4 that care about preserving the termination of the effects. If one wishes to support sub-typing and sub-effecting in the generation of programs, the rules of Nielson and Nielson [1999] could be added in the rules defined in Section 4.

4 TERMINATING EFFECTS

The type and effect system captures the communication behavior of λ_{chan} programs and can be used to generate valid programs given a type and an effect. However, how can we predict what the result of executing such programs should be? It is *insufficient* to know the effect of a program: we need a stronger property that can be automatically checked. In this paper, we focus on termination. That is, we want to generate effects (and ultimately programs) that terminate for any execution. If such a program fails to terminate (or crashes) we will have found a bug.

We are now left with the question of: *how do we generate an interesting terminating effect*? Not all effects correspond to terminating programs. Take, for example, the effect $\textsc{Chan}(c); \textsc{Get}(c)$ which fails to terminate since there is no put operation on the channel c, and the get operation hangs forever. Conversely, not all terminating programs are interesting. The effect $\epsilon; \textsc{Spawn}(\epsilon)$ terminates, but such effects are probably uninteresting for testing an implementation of channels and processes. To overcome these issues, we generate a terminating effect φ in three steps:

(1) (**Generation**) We define a context-free grammar of terminating effects that cause a lot of channel contention, i.e. communication over the same channels by multiple processes. We prove that the programs that are in the language of effects must terminate.
(2) (**Expansion**) We duplicate fragments of the effect, to make it more interesting, while preserving termination.
(3) (**Reordering**) We reorder fragments of the effect while preserving termination.

We focus on termination as our correctness oracle and hence we shall not concerned with the particular values that flow over channels, only the communication itself. Thus, and for simplicity, we only consider channels of unit type. We leave it as interesting future work to determine if sending or receiving more elaborate values can reveal additional buggy behavior.

We shall use these important definitions in the proofs:

Definition 4.1 (Final). A configuration P is *final* if there is no other configuration P' such that $P \Rightarrow P'$, i.e. the configuration cannot take a step according to the evaluation relation $\Rightarrow$.

Definition 4.2 (Terminated). A process p in a configuration P where $p \in \mathsf{dom}(P)$ is *terminated* if the expression of p is a value, i.e., $P(p) \in \mathsf{Value}$. A configuration is *terminated* if all its processes are terminated.

Definition 4.3 (Terminating). A configuration P is said to be *terminating* if all its executions reach a terminated configuration, i.e. for all P' that are final and such that $P \Rightarrow^\star P'$, then P' is terminated. An expression e is said to be *terminating* if the configuration $[p : e]$ is terminating. An effect φ is said to be *terminating* if all expressions e that have this effect (i.e., $e : t \;\&\; \varphi$ for some t) are terminating.

4.1 Effect Generation

The first step is to generate a terminating effect. Figure 10 shows an inductive definition of a sub-language of effects. This sub-language is inspired by common concurrency patterns and aims to capture programs whose execution cause contention on channels, i.e. causes multiple simultaneous operations on the same channels, with the hope of discovering races or deadlocks. We shall discuss each rule in detail shortly. An important property is that each rule is compositional in the sense that the channels are always fresh. That is, each rule may introduce a set of channels, but no sub-effect can refer to the same channels.

The (G-Final), (G-Seq), (G-Choice) and (G-Spawn) rules are straightforward. For (G-Final), the empty effect is always terminating. For (G-Seq), if we have two terminating effects and they do not reference the same channels then their sequence must be terminating. For (G-Choice), if we have two terminating effects then choosing one of them is terminating. For (G-Spawn) if an effect is terminating then running it in a fresh process is terminating.

We now discuss the rest of the rules in turn.

(G-PingPong). It is easy to see that the effect:

$$\textsc{Spawn}(\textsc{Put}(c)); \textsc{Get}(c)$$

where c is a fresh channel is a terminating effect. The effect has exactly one get and one put operation on c and each operation happens in different processes; one in a fresh process and the other in the original process. We can generalize the effect to a sequence of get and put operations:

$$\textsc{Spawn}([\textsc{Put}(c)]_0^n); [\textsc{Get}(c)]_0^n$$

where $[\varphi]_0^n$ means that φ is repeated $n+1$ times. This effect is also terminating, since we assume (and we will continue to assume) that the channel c is fresh and because each get operation is

$$\epsilon \in \text{Generation (G-Final)} \qquad \frac{\varphi_1 \in \text{Generation} \quad \varphi_2 \in \text{Generation}}{\varphi_1;\varphi_2 \in \text{Generation}}\ \text{(G-Seq)}$$

$$\frac{\varphi_1 \in \text{Generation} \quad \varphi_2 \in \text{Generation}}{\varphi_1 + \varphi_2 \in \text{Generation}}\ \text{(G-Choice)} \qquad \frac{\varphi \in \text{Generation}}{\textsc{Spawn}(\varphi) \in \text{Generation}}\ \text{(G-Spawn)}$$

$$\frac{c \text{ is fresh} \quad \varphi_i \in \{\textsc{Get}(c), \textsc{Put}(c)\} \quad n > 0}{\textsc{Spawn}(\square; [\varphi_i; \square]_0^n); [\square; \overline{\varphi_i}]_0^n \in \text{Generation}}\ \text{(G-PingPong)}$$

$$\frac{\forall i. c_i \text{ is fresh} \quad \varphi_i \in \{\textsc{Get}(c_i), \textsc{Put}(c_i)\} \quad n > 0}{[\textsc{Spawn}(\square; \varphi_i; \square)]_0^n; [\square; \overline{\varphi_i}]_0^n \in \text{Generation}}\ \text{(G-FanOut)}$$

$$\frac{\forall i.\, c_i \text{ is fresh} \quad n > 0}{[\textsc{Spawn}(\square; \textsc{Get}(c_{i-1}); \square; \textsc{Put}(c_i); \square); \square]_1^n; \textsc{Put}(c_0); \square; \textsc{Get}(c_n) \in \text{Generation}}\ \text{(G-Pipeline)}$$

$$\frac{\forall i. c_i \text{ is fresh} \quad \varphi_i \in \{\textsc{Get}(c_i), \textsc{Put}(c_i)\} \quad m, n > 0}{[[\textsc{Spawn}(\square; \varphi_i; \square)]_0^n; \square]_0^m; [\square; \underbrace{\text{branch}(\text{shuffle}([\overline{\varphi_i}]_0^n)) \oplus \ldots \oplus \text{branch}(\text{shuffle}([\overline{\varphi_i}]_0^n))}_{\text{arbitrary number of branches } (>0)}]_0^m \in \text{Generation}}\ \text{(G-Select)}$$

$$\text{branch}(\textsc{Get}(c); \varphi_1; \ldots; \varphi_n) = \textsc{SelGet}(c, \square; \varphi_1; \square; \ldots; \square; \varphi_n)\ \text{(Branch-Put)}$$

$$\text{branch}(\textsc{Put}(c); \varphi_1; \ldots; \varphi_n) = \textsc{SelPut}(c, \square; \varphi_1; \square; \ldots; \square; \varphi_n)\ \text{(Branch-Get)}$$

Fig. 10. Deadlock-Free Effects. $[x_i]_a^b$ is defined as $x_a; x_{a+1}; \cdots; x_{b-1}; x_b$. $\text{shuffle}(\varphi_0; \ldots; \varphi_n)$ is a random permutation of the sequence of effects $\varphi_0; \ldots; \varphi_n$.

matched by a corresponding put operation. We can further generalize the effect by observing that there is no reason for all the put operations to happen in the spawned process and all the get operations to happen in the original process. Instead, we simply need to make sure that the operations are pairwise matched:

$$\textsc{Spawn}([\varphi_i]_0^n); [\overline{\varphi_i}]_0^n$$

where $\varphi_i = \{\textsc{Get}(c), \textsc{Put}(c)\}$ and $\overline{\varphi_i}$ denotes the dual effect of φ_i. We can make one final generalization. If $\square$, $\square_i'$, and $\square_i''$ are terminating effects then we can interleave these effects to get:

$$\textsc{Spawn}(\square; [\varphi_i; \square_i']_0^n); [\square_i''; \overline{\varphi_i}]_0^n$$

where, as before, $\varphi_i = \{\textsc{Get}(c), \textsc{Put}(c)\}$. For simplicity, we will abuse notation and write simply $\square$ without subscripts with the understanding that each occurrence of $\square$ corresponds to a terminating effect not necessarily equal to any other $\square$ occurrence. Hence the above rule is equivalent to the shorter and more readable rule:

$$\textsc{Spawn}(\square; [\varphi_i; \square]_0^n); [\square; \overline{\varphi_i}]_0^n$$

where, as before, $\varphi_i = \{\textsc{Get}(c), \textsc{Put}(c)\}$. We call this generation rule (G-PingPong) because it subsumes two processes exchanging messages, while possibly participating in other interleaved communication patterns.

(G-FanOut). In the previous rule, we started with:

$$\textsc{Spawn}(\textsc{Put}(c)); \textsc{Get}(c)$$

which is terminating if c is a fresh channel. We can generalize this in a different direction by introducing another channel and process:

$$\textsc{Spawn}(\textsc{Put}(c_1)); \textsc{Spawn}(\textsc{Put}(c_2)); \textsc{Get}(c_1); \textsc{Get}(c_2)$$

This effect is terminating because there is exactly one get and one put on each of the channels c_1 and c_2, and the operations on c_1 can synchronize and afterwards the operations on c_2 can synchronize. We can generalize this to any number of channels and processes:

$$[\textsc{Spawn}(\textsc{Put}(c_i))]_0^n; [\textsc{Get}(c_i)]_0^n$$

This effect is terminating by the same argument as above. As before, we can use the notion of dual effect to abstract over where the get and put operations happen:

$$[\textsc{Spawn}(\varphi_i)]_0^n; [\overline{\varphi_i}]_0^n$$

where $\varphi_i = \{\textsc{Get}(c_i); \textsc{Put}(c_i)\}$ and each channel c_i is fresh. Finally, we can interleave the effect with other terminating effects:

$$[\textsc{Spawn}(\square; \varphi_i; \square)]_0^n; [\square; \overline{\varphi_i}]_0^n$$

where, as before, $\varphi_i = \{\textsc{Get}(c_i); \textsc{Put}(c_i)\}$ and each channel c_i is fresh. We call this generation rule (G-FanOut) since it subsumes a process that spawns several sub-processes to do work before collecting all their results. The (G-PingPong) and (G-FanOut) rules look strikingly similar, but yet they are different: The (G-PingPong) rule has 1 channel and 2 processes whereas the (G-FanOut) rule has n channels and $n + 1$ processes.

(G-Pipeline). It is easy to see that the effect:

$$\textsc{Spawn}(\textsc{Get}(c_0); \textsc{Put}(c_1)); \textsc{Put}(c_0); \textsc{Get}(c_1)$$

is terminating if c_0 and c_1 are fresh channels. The put operation, on c_0, in the original process causes the get operation, on c_0, in the spawned process to proceed. This in turn causes the put operation on c_1 in the spawned process to trigger which finally can synchronize with the original process. We can generalize this to a pattern where a message is sent from the original process through a pipeline of other processes before finally being returned to the original process:

$$[\textsc{Spawn}(\textsc{Get}(c_{i-1}); \textsc{Put}(c_i))]_1^n; \textsc{Put}(c_0); \textsc{Get}(c_n)$$

Here each spawned process i waits to a receive a message on channel i and once it does it sends it along on channel $i + 1$. The original process starts the pipeline with a put on c_0 and receives the final result on c_n. As before, we can interleave terminating effects:

$$[\textsc{Spawn}(\square; \textsc{Get}(c_{i-1}); \square; \textsc{Put}(c_i); \square)]_1^n; \textsc{Put}(c_0); \square; \textsc{Get}(c_n)$$

We call this rule (G-Pipeline) as it models a pipeline of processes where each process hands of its message to the next in the pipeline.

$$\frac{}{\varphi \rightarrow_E \varphi + \varphi}\ (\text{E-Choice-Dup}) \qquad \text{Get}(c) \rightarrow_E (\text{SelGet}(c, \epsilon)) \oplus (\text{SelGet}(c, \epsilon))\ (\text{E-Get-Select})$$

$$\text{Put}(c) \rightarrow_E (\text{SelPut}(c, \epsilon)) \oplus (\text{SelPut}(c, \epsilon))\ (\text{E-Put-Select}) \qquad \frac{\varphi_1, \varphi_2 \in \text{Generation}}{\varphi \rightarrow_E \varphi_1; \varphi; \varphi_2}\ (\text{E-Seq})$$

$$\frac{i \in \{1, 2\}}{\varphi_1^{sr} \oplus \varphi_2^{sr} \rightarrow_E \varphi_1^{sr} \oplus \varphi_i^{sr} \oplus \varphi_2^{sr}}\ (\text{E-Select-Dup})$$

Fig. 11. Effect Expansion Rules.

(G-Select). Consider the following effect:

$$\varphi_0 = \text{Spawn}(\text{Put}(c_1)); \text{Spawn}(\text{Put}(c_2))$$

This effect can be safely combined with a process that will read exactly once on channel c_1 and once on channel c_2, *in any order*. The effect of such a process could for example be the following:

$$\varphi_1 = (\text{SelGet}(c_1, \text{Get}(c_2))) \oplus (\text{SelGet}(c_2, \text{Get}(c_1)))$$

The combined effect $\varphi_0; \varphi_1$ is a terminating effect. The same reasoning can be applied to more channels. For example, consider the following effect:

$$\varphi_0' = \text{Spawn}(\text{Put}(c_1)); \text{Spawn}(\text{Put}(c_2)); \text{Spawn}(\text{Put}(c_3))$$

This can safely be combined with:

$$\begin{aligned}\varphi_1' = &(\text{SelGet}(c_1, \text{Get}(c_2); \text{Get}(c_3)))\\ &\oplus (\text{SelGet}(c_1, \text{Get}(c_3); \text{Get}(c_2)))\\ &\oplus (\text{SelGet}(c_2, \text{Get}(c_1); \text{Get}(c_3)))\\ &\oplus (\text{SelGet}(c_2, \text{Get}(c_3); \text{Get}(c_1)))\\ &\oplus (\text{SelGet}(c_3, \text{Get}(c_1); \text{Get}(c_2)))\\ &\oplus (\text{SelGet}(c_3, \text{Get}(c_2); \text{Get}(c_1)))\end{aligned}$$

Or with any select composed of a subset of these branches. This is exactly what the (G-Select) rule generalizes: it communicates m times over n channels, and generates m select constructs composed of an arbitrary number of branches. Each select will communicate exactly once over each channel.

4.2 Effect Expansion

The second step is to expand the terminating effect to make it more interesting. Figure 11 shows the expansion rules. These rules are given as a term rewriting system and can be applied to any sub-effect of a given effect. The expansion rules preserve termination, but not necessarily semantics. This makes sense since our goal is to generate terminating programs. Not all rules are in themselves interesting, but they may enable further rules.

(E-Choice-Dup). The rule simply copies an effect φ into a choice effect $\varphi + \varphi$. Clearly if φ is terminating then so is the $\varphi + \varphi$.

$$\frac{}{\varphi_1^{sr} \oplus \varphi_2^{sr} \rightarrow_R \varphi_2^{sr} \oplus \varphi_1^{sr}}\ \text{(R-Select)}$$

$$\frac{}{(\textsc{Get}(c_1); \varphi_1) + (\textsc{Get}(c_2); \varphi_2) \rightarrow_R (\textsc{SelGet}(c_1, \varphi_1)) \oplus (\textsc{SelGet}(c_2, \varphi_2))}\ \text{(R-ChoiceSelectGetGet)}$$

$$\frac{}{\textsc{Spawn}(\varphi_1); \textsc{Spawn}(\varphi_2) \rightarrow_R \textsc{Spawn}(\varphi_2); \textsc{Spawn}(\varphi_1)}\ \text{(R-Spawn-1)}$$

$$\frac{}{\textsc{Spawn}(\varphi_1); \textsc{Spawn}(\varphi_2) \rightarrow_R \textsc{Spawn}(\textsc{Spawn}(\varphi_2); \varphi_1)}\ \text{(R-Spawn-2)}$$

Fig. 12. Effect Reordering Rules.

(E-Select-Dup). The rule duplicates a case of a select effect. This has no semantic effect, but allows us to generate programs such as:

```
select {
    case x <- c1 => ...
    case x <- c2 => ...
    case x <- c2 => ...
}
```

where the same channel is selected from multiple times.

(E-Get-Select) and (E-Put-Select). The rule replaces a get (resp. put) operation $\textsc{Get}(c)$ (resp. $\textsc{Put}(c)$) by a select operation with two identical cases that have empty bodies. This is semantically equivalent to an ordinary get (resp. put) operation.

(E-Seq). The rule expands a terminating effect to a sequence of terminating effects.

4.3 Effect Reordering

The third step is to reorder the terminating effect. Figure 12 shows the reordering rules. Similar to the expansion rules, these are rewrite rules that can be applied inside any generated effect and do not influence termination.

(R-Select). The rule states that in a select effect, the order of the two cases can be swapped without affecting termination. This rule is interesting, because if we have two select expressions:

```
select {
    case x <- c1 => ...
    case y <- c2 => ...
}
```

```
select {
    case x <- c1 => ...
    case y <- c2 => ...
}
```

then reordering the last select expressions means that if any locks are taken on the channels c_1 and c_2 the implementation has to be very careful about the order in which they are acquired, so as to not cause a deadlock between the two select expressions.

(R-Choice-SelectGetGet). The rule states that if we have a choice between two effects of the shape $(\textsc{Get}(c_1); \varphi_1)$ and $(\textsc{Get}(c_2); \varphi_2)$, then it can be replaced by the form $(\textsc{SelGet}(c_1, \varphi_1)) \oplus (\textsc{SelGet}(c_2, \varphi_2))$. This is because a choice provides a stronger guarantee than a select: for a choice to terminate, both branches should terminate, whereas for a select to terminate only the branch

selected based on which channel is ready to synchronize need to terminate. It is then safe to transform a choice into a selection. The opposite is not true: To preserve termination, a select can *not* be replaced by a choice.

(R-Spawn-1). The rule states that a sequence of two spawns $\textsc{Spawn}(\varphi_1)\,;\,\textsc{Spawn}(\varphi_2)$ can be replaced by an effect where the order of spawns are swapped: $\textsc{Spawn}(\varphi_2)\,;\,\textsc{Spawn}(\varphi_1)$.

(R-Spawn-2). The rule states that a sequence of two spawns: $\textsc{Spawn}(\varphi_1)\,;\,\textsc{Spawn}(\varphi_2)$ can be replaced by an effect where one spawn is nested inside the other: $\textsc{Spawn}(\textsc{Spawn}(\varphi_2)\,;\,\varphi_1)$.

4.4 Termination Theorems

Theorem 4.4 (Generated Effect Terminates). *If $\varphi \in$ Generation, then φ is a terminating effect.*

Theorem 4.5 (Rewrite Rules Preserves Termination). *If φ is terminating, applying any rewrite rule of Figure 11 (expansion), or Figure 12 (reordering) to φ preserves termination.*

The detailed proofs are available in the technical report[1].

5 IMPLEMENTATION

To evaluate our approach, we have implemented program generators for the Go, Kotlin, Crystal, and Flix programming languages. The tool generates an effect using the rules described in Section 4. The effect is translated into a λ_{chan} expression which is then translated into a program in each of the target languages.

5.1 Effect Generation

The implementation closely mirrors the rules outlined in Section 4. We assign a weight to each rule. The weight determines how likely the rule is to be applied: the heavier the weight, the more often the rule is chosen. We discuss the choice of weights in Section 6.2.

5.2 Program Generation

Given a terminating effect φ, we generate an expression $e : \mathsf{unit} \,\&\, \varphi$ by following the type and effect system of Section 3. Each type and effect describes a set of expressions. We randomly generate a specific program for the target programming language from a type and effect as follows:

(1) We recursively translate the type and effect into expressions (or statements) of the target language corresponding to the appropriate channel operations. Whenever a type and effect maps to multiple possible expressions or statements, we choose one at random (see below).
(2) We collect all channels that occur in the effect and let-bind them such that they are visible in the generated expression (or statement) from Step 1. It is safe to do so, since channels are given static names, as discussed in Section 3.
(3) We wrap the entire generated expression (or statement) from Step 2 in an appropriate main method with appropriate imports and so forth for the target language.

Example I. Given the (non-terminating) effect:

$$\textsc{Spawn}(\textsc{Put}(c_1) + \textsc{Put}(c2))\,;\,\textsc{Get}(c_1)$$

we non-deterministically translate it into the Flix expression:

```
spawn { if (true) c1 <- () else c2 <- () }; <- c1
```

or

[1] Available at https://soft.vub.ac.be/Publications/2020/vub-tr-soft-20-15.pdf

```
spawn { if (false) c1 <- () else c2 <- () }; <- c1
```

The entire generated Flix program, in the first case, is then:

```
def main(): Unit =
  let c1 = chan 0;
  let c2 = chan 0;
  spawn { if (true) c1 <- () else c2 <- () };
  <- c1
```

Given the terminating effect $\textsc{Spawn}(\textsc{Put}(c) + \textsc{Put}(c)) ; \textsc{Get}(c)$ it is equally valid to generate expressions such as:

```
spawn { if (true) c <- () else c <- () }; <- c
spawn { if (false) c <- () else c <- () }; <- c
spawn { if (1 == 1) c <- () else c <- () }; <- c
spawn { if (1 != 1) c <- () else c <- () }; <- c
spawn { if (true) c <- 123 else c2 <- 456 }; <- c
spawn { if (Random.nextBool()) c <- 123 else c <- 456 }; <- c
```

And so forth. The type and effect system, together with the proofs, guarantees that such programs must terminate.

Example II. Given the terminating effect:

$$\textsc{Spawn}(\textsc{Get}(c_1)); \textsc{Spawn}(\textsc{Put}(c_2)); ((\textsc{Put}(c_1); \textsc{Get}(c_2)) + (\textsc{Get}(c_2); \textsc{Put}(c_1)))$$

we non-deterministically translate it into Crystal as:

```
spawn do c1.receive end;
spawn do c2.send(nil) end;
if b
  c1.send(nil); c2.receive
else
  c2.receive; c1.send(nil)
end
```

or with a different ordering on the conditional branches:

```
spawn do c1.receive end;
spawn do c2.send(nil) end;
if b
  c2.receive; c1.send(nil)
else
  c1.send(nil); c2.receive
end
```

The entire generated program, in the first case, can then be:

```
def main():
  c1 = Channel(Nil).new
  c2 = Channel(Nil).new
  spawn do c1.receive end;
  spawn do c2.send(nil) end;
  if true
    c2.receive; c1.send(nil)
  else
    c1.send(nil); c2.receive
  end
```

Example III. Given the terminating effect:

$$\textsc{Spawn}(\textsc{Put}(c_1); \textsc{Get}(c_2)); \textsc{Get}(c_1); \textsc{Put}(c_2) \oplus \textsc{Put}(c_1); \textsc{Get}(c_2)$$

we non-deterministically translate it into Kotlin as:

```
launch { c1.send(Unit); c2.receive() } };
select<Unit> {
  c1.onReceive { _ -> c2.send(Unit) }
  c1.onSend { _ -> c2.receive() }
}
```

or with a different ordering on the select clauses:

```
launch { c1.send(Unit); c2.receive() } };
select<Unit> {
  c1.onSend { _ -> c2.receive() }
  c1.onReceive { _ -> c2.send(Unit) }
}
```

The entire generated program, in the first case, is then:

```
object Main {
  fun main(...): Unit = runBlocking<Unit> {
    val c1 = Channel<Unit>();
    val c2 = Channel<Unit>();
    launch { c1.send(Unit); c2.receive() } };
    select<Unit> {
      c1.onReceive { _ -> c2.send(Unit) }
      c1.onSend { _ -> c2.receive() }
    }
  }
```

5.3 Detecting Bugs

Each generated program is executed with a timeout of 30 seconds. If the program crashes, we have found a bug in the programming language implementation. If the program times out, we flag it for manual inspection to determine if it has deadlocked.

Manual inspection was performed by the two authors, and entailed the following:

(1) We rerun the program multiple times with a higher timeout: if any execution fails to terminate, it is likely that the program contains a deadlock. A small-sized program rarely takes several minutes to terminate (unless something is wrong).
(2) When available, we use language tools such as the JVM runtime deadlock detection tool and manual inspection of threads and locks to assess whether the program language runtime deadlocked. When not available (in the case of Crystal), we rely solely on manual inspection.
(3) In addition to proving the correctness of the generation rules, we manually inspect the generated program to convince ourselves that all of its executions must terminate. A sample of the programs are included, so the reader is free to verify this for him or herself.

We experimentally choose a timeout of 30 seconds. A low timeout may result in many programs being flagged for manual inspection, as the timeout includes the time to compile and execute the program. A high timeout, on the other hand, will reduce the throughput of the program generator.

6 EVALUATION

We have proved the soundness of our technique. A program with an effect generated by the rules must always terminate. Our primary experimental research question is then whether the technique is useful for finding bugs in real-world programming languages.

To evaluate the technique, we consider three experimental research questions:

- **RQ1:** is the technique able to find bugs in real-world programming languages?
- **RQ2:** are the generation, expansion, and reordering rules useful?
- **RQ3:** what weight heuristics increase the chance of finding bugs?

We ran our technique on the four languages listed in Table 1. The table also shows the number of GitHub stars and the rank of each programming language in the TIOBE index as of August 2020 [TIOBE 2020]. The maturity of the channel and process implementation is also indicated by the number of years since it was integrated into the language. In total we tested 164 different compiler versions.

All experiments were run on a docker environment set up with 16GB of RAM, on a 2015 Dell PowerEdge R730 with 2 Intel Xeon 2637 processors with a frequency of 3GHz.

Table 1. Tested Programming Languages.

Language / Library	Versions	Total Versions	Channels Since	GitHub Stars	TIOBE
Flix	0.5.0 – 0.8.1	7	Feb. 26, 2019	530	n/a
kotlinx-coroutine	0.19 – 1.3.3	39	Feb. 7, 2017	8,000	29
Crystal	0.19.0 – 0.32.1	33	Apr. 30, 2015	15,100	51-100
Go	1 – 1.13.6	85	Nov. 10, 2009	75,700	11

6.1 RQ1: Finding Bugs

We ran our technique on every version of Go, Kotlin, Crystal, and Flix listed in Table 1. As explained earlier, we execute each generated program with a timeout of 30 seconds. If the program crashes, we have found a bug in the programming language implementation. If the program times out, we flag it for manual inspection to determine if it has deadlocked. Each bug we found was discovered in a few minutes by our tool, after generating a couple of hundred programs. *In every single case we observed, when a program timed out it was due to a bug in the programming language implementation.*

We have looked through the GitHub issue trackers for the respective languages to discover if our tool missed in any known concurrency bugs within the scope of our paper. At the time of writing, we did not discover any such bugs.

We now report on the results for each programming language.

6.1.1 Flix. We ran our implementation on Flix from version 0.5.0 (the first version to support channels) until version 0.8.1. In total, we tested 7 versions of Flix.

We found two bugs in the channel and process implementation, which have been confirmed and fixed in version 0.8.1. The first bug, which we refer to as Flix I, was discussed in Section 2 and causes a deadlock. Figure 13 shows a program that demonstrates the second bug, which we call Flix II. When this program is compiled with Flix 0.5.0 and 0.6.0, the compiler emits illegal JVM bytecode which causes the program to crash. Neither of these bugs were previously discovered, despite an extensive test suite for channel and processes with more than 345 unit tests spanning more than 3,500 lines of code.

Flix
```
def main(): Unit = {
  let c1 = chan Unit 0;
  // ... channels c2 to c4 ...
  spawn {
    select {
      case _ <- c3 => c4 <- ()
      case _ <- c3 => c4 <- ()
    }; ()
  };
  spawn { <- c2; c3 <- () };
  spawn { <- c1; c2 <- () };
  c1 <- ();
  <- c4
}
```

Fig. 13. A Generated Program that Crashes in Flix.

6.1.2 Crystal. We ran our implementation on all available versions of Crystal between version 0.19.0 (the first version to have the channels API), and version 0.32.1. In total, we tested 33 versions of Crystal.

We found a bug that results in a deadlock with Crystal starting at version 0.19.0 (released on September 2, 2016), until it was fixed in version 0.23.0 (released on July 27, 2017). This bug remained in the official releases of Crystal for almost a year (330 days), and impacted 13 releases. The bug was reported on the Crystal issue tracker on January 8, 2017[2].

Figure 14 shows a program that triggers the bug. The program contains three channels, `c1`, `c2`, and `c3`, and spawns three threads (that we number 1 to 3), in addition to the main thread (that we number 0). The situation before the `select` of thread 0 is executed is the following, assuming other threads have progressed as much as they can: thread 1 is trying to receive on channel `c1`, thread 2 is trying to receive on channel `c3`, and thread 3 is trying to send on channel `c2`. When executing the `select` of thread 0, all communications can be fulfilled: a value is sent to `c3`, allowing thread 2 to finish its execution, a value is then sent to `c1`, allowing thread 1 to finish its execution, and a value is read from `c2`, allowing thread 3 to finish its execution. Thread 0 can then also finish its execution. Hence, no deadlock is present in the code.

However, running this with a version of Crystal between 0.19.0 and 0.22.0 results in a deadlock. The fix for this bug consists of a single line, changing the definition of an *empty* channel in Crystal, from being a channel with no value, to being a channel with no value *and* no senders.

We believe our work is the first that tries to fuzz concurrency runtimes. We did, however, run the popular state-of-the-art C fuzzer AFL[3] on a buggy version of Crystal. We ran AFL for 72 hours, generating more than 2 million programs, without discovering the bug we found or any other bugs. If AFL had managed to generate a non-terminating program that, however, would not by itself tell us anything. Either the program does not terminate because it exposes a bug in the language runtime or it does not terminate because the program itself contains a deadlock.

Our work has the critical property that every generated program must terminate. Hence we can use termination as an oracle for correctness. This is simply not possible with generic fuzzers: A generic fuzzer cannot automatically find the deadlock bugs found by our work.

[2] https://github.com/crystal-lang/crystal/issues/3862
[3] http://lcamtuf.coredump.cx/afl

Crystal

```
def main()
  c1 = Channel(Nil).new
  c2 = Channel(Nil).new
  c3 = Channel(Nil).new
  # Thread 1
  spawn do c1.receive end
  spawn do # Thread 2
    select
      when _ = c3.receive
        nil
      when _ = c3.receive
        nil
    end
  end
  # Thread 3
  spawn do c2.send(nil) end
  select # Thread 0
  when _ = c3.send(nil)
    c1.send(nil)
    c2.receive
  when _ = c3.send(nil)
    c1.send(nil)
    c2.receive
  end
end
```

Fig. 14. A Generated Program that Deadlocks in Crystal.

6.1.3 Kotlin. We ran our implementation on all versions of the kotlinx-coroutine library available on Maven between version 0.19 (the first version of the library available on Maven) and version 1.3.3. This makes a total of 39 versions of the library that were tested[4].

We found a bug that result in a crash with the kotlinx-coroutine library starting at version 0.19 (released in September 2017), until it was fixed in version 0.26.0 (released in September 2018). The bug causes the program to crash with a stack overflow exception. This bug remained in the official releases of the kotlinx-coroutine for a year, and impacted 20 releases. The bug was reported on the Kotlin issue tracker on August, 21, 2018[5]. Notably, the bug report speculates that there could be a bug, but does not include a program fragment that reproduces the problem.

Figure 15 shows a program that triggers the bug. The program has five channels (`c1` to `c5`), along with five created processes that each perform an operation on one of the channel, either a receive or a send. The main thread performs the dual operation on all five channels using a `select` expression. The `select` expression contains multiple branches, but all branches will correctly perform the dual operations that is performed by the other threads for each channel, as discussed in Section 4. Hence, the program must terminate. However, it results in a crash with `kotlinx-coroutine` version 0.19 until 0.26.0. This bug was indirectly fixed as part of a major rework of the concurrency model[6].

[4]In Kotlin, channels are implemented as part of the standard library using coroutines.
[5]https://github.com/Kotlin/kotlinx.coroutines/issues/504
[6]https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0

Kotlin

```
object Main {
  fun main(...): Unit = runBlocking<Unit> {
    val c1 = Channel<Unit>();
    // ... channels c2 to c5 ...
    launch { // Thread 1
      select<Unit> {
        c5.onReceive { _ -> Unit }
        c5.onReceive { _ -> Unit }
        c5.onReceive { _ -> Unit }
      }
    };
    launch { c4.receive() };  // Thread 2
    launch { c3.receive() };  // Thread 3
    launch { c2.send(Unit) }; // Thread 4
    launch { c1.send(Unit) }; // Thread 5
    select<Unit> { // Thread 0
      c2.onReceive { _ ->
        c4.send(Unit);  c1.receive(); c5.send(Unit);  c3.send(Unit)
      }
      c5.onSend(Unit) {
        c2.receive();  c3.send(Unit); c1.receive();  c4.send(Unit)
      }
      c3.onSend(Unit) {
        c1.receive();  c2.receive(); c4.send(Unit); c5.send(Unit)
      }
    }
  }
}
```

Fig. 15. A Generated Program that Crashes in Kotlin.

6.1.4 Go. We ran our technique on Go ranging from Go 1 to Go 1.13.6, without finding any bugs. We looked through the GitHub issue tracker for Go and we were not able to find any bug report of a defect in its concurrency runtime. It seems that the concurrency runtime was developed "in-house" before being open-sourced, i.e., the first relevant commit seems to contain a complete and finished implementation. It is possible that the implementation was already battle-tested internally before being made publicly available. Thus, to the best of our knowledge, there is no relevant bug in the Go concurrency runtime for us to reproduce.

Summary for RQ1. We found two previously unknown bugs in Flix, and reproduced two bugs; one in Crystal and one in Kotlin. While such a number may seem low compared to other fuzzing approaches, we should remember that these bugs are:

- **Critical** because they may impact *every* user of the programming language that relies on channel- and process-based concurrency,
- **Difficult to discover** because they involve non-determinism, and
- **Semantically deep** because they involve complicated implementations (e.g., concurrency, shared mutable memory, complex locking, etc.).

Table 2. Experimental Results with Different Heuristics for the Rule Weights.

	(G-Seq)	(G-Choice)	(G-Spawn)	(G-PingPong)	(G-FanOut)	(G-Pipeline)	(G-Select)	Expansion	Reordering	Flix I	Flix II	Crystal	Kotlin
Heuristic A	1	1	1	1	1	1	1	1	1	99	32	28	48
Heuristic B	1	1	1	1	1	1	1	1	**0**	98	0	13	149
Heuristic C	1	1	1	1	1	1	1	**0**	1	47	0	0	0
Heuristic D	1	1	1	1	1	1	**0**	1	1	0	36	0	0
Heuristic E	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	600	0	86	148
Heuristic F	1	1	1	**2**	**2**	**2**	**15**	1	1	400	12	63	155

Moreover, these bugs were found across multiple languages in trusted components that we expect to be well-tested. We are not aware, at the time of writing, of any bug report for Flix, Crystal, Kotlin, and Go that fits within the scope of our tool and which was not found by it. Of course there could still be bugs unknown to anyone. We answer Q1 by concluding that our technique is useful for fuzzing channel- and process based concurrency runtimes.

6.2 RQ2: Usefulness of Rules

We have found four bugs in the Kotlin, Crystal, and Flix programming languages using all the generation, expansion, and reordering rules described in Section 4. We now want to determine whether all of these rules are useful.

We choose a buggy version of each compiler/runtime (Kotlin with kotlinx-coroutine 0.19, Crystal 0.19, Flix 0.5.0). We then choose a subset of the rules and generate and execute 5,000 programs. We measure the number of programs that reveal the bug in the compiler/runtime. Our goal is to understand whether selectively enabling (or disabling) a rule increases (or decreases) the chance of discovering a bug.

We use individual weights for the generation rules, but group the expansion and reordering rules into one category since they are fairly uniform. A weight of 0 means that the rule is excluded.

Table 2 shows the result of this experiment. The table shows the weights assigned to every rule or rule group. The table also shows the number of programs generated, out of 5,000, that reveal a bug for each of the programming languages. For example, Heuristic A assigns every rule a weight of one. The results show that Heuristic A generates 99/5,000 Flix programs that reveal bug I, 32/5,000 Flix programs that reveal bug II, 28/5,000 Crystal programs that reveal a bug, and 48/5,000 Kotlin programs that reveal a bug. We can see that a significant number of programs are required to discover a bug. For example, for Crystal the ratio 28/5,000 is less than 1%.

Based on the experiments, we observe that:

- Heuristic A, which has all rules with a weight of one, is able to discover 4/4 bugs.
- Heuristic B, which omits the reordering rules, is able to discover 3/4 bugs, but not the Flix II bug. This suggests that the reordering rules are useful.
- Heuristic C, which omits the expansion rules, is only able to discover 1/4 bugs. This suggests that the expansion rules are useful.
- Heuristic D, which omits the (G-Select) generation rule, is only able to discover 1/4 bugs.
- Heuristic E, which omits all generation rules except for the (G-Select) rule, is able to find 3/4 bugs. This, together with Heuristic D, supports our claim that the select construct is the source of significant complexity and is where implementations are often incorrect.

Summary for RQ2. The experiments suggest that the generation, expansion, and reordering rules are all useful for discovering bugs. We cannot exclude expansion or reordering rules without missing bugs. We also cannot limit ourselves to the (G-Select) rule without missing at least one bug. We did not try all combinations of the rules, since there are exponentially many, but the experimental results suggests that as a whole the rules are useful. It may be that some specific rule could be omitted and we would still be able to obtain the same results. But of course there is the danger of over fitting the rules to the results. We recommend that future implementations begin with all rules and then work from there. In summary, we answer Q2 by concluding that all rules or rule groups appear useful.

6.3 RQ3: Weight Heuristics

We now consider the question of how the individual weights assigned to each rule influence the chance to a find a bug. We use the same experiment and data from Table 2.

Heuristic F has not been mentioned before. It assigns a weight of 1 to most rules, a weight of 2 to the (G-PingPong), (G-FanOut), (G-Pipeline) rules, and a weight of 15 to the (G-Select) rule. Heuristic F is the rule we used while developing the tool before we considered any of the other heuristics from Table 2. We choose the weights based on "eye-balling" the generated programs, i.e. some subjective measure of "does this program look interesting".

In the following, we shall use the term "success rate" to refer to the ratio of programs that reveal a bug to those that do not. Based on the experiments, we observe that:

- Heuristic E has the highest success rate for Flix I.
- Heuristic D has the highest success rate for Flix II, followed by Heuristic A.
- Heuristic E has the highest success rate for Crystal, followed by Heuristic F
- Heuristic F has the highest success rate for Kotlin, followed by Heuristic B and F.

The results suggest that there is no unequivocally best heuristic. The results also show that changing the weights may significantly increase or decrease the success rate. Based on Heuristic A, Heuristic E, and Heuristic F it seems that, except for Flix II, increasing the weight of the (G-Select) rule does improve the success rate across the board. If we compare Heuristic A (every rule has weight one) and Heuristic E (our eye-balled weights), it appears that there is no strong reason to try "guess" good weights from the outset. While Heuristic F has a high success rate for Flix I, Crystal, and Kotlin the major worry is that it might miss Flix II. It is important that the success rate is balanced so that we are most likely to discover all bugs. Consequently, we recommend that future implementations start by giving every rule equal weight.

Summary for RQ3. The experiments suggest that there is no unequivocally best heuristic. Tuning the weights may significantly increase or decrease the chance to discover one bug, but may be to the detriment of discovering another bug.

7 RELATED WORK

Concurrency. Hoare introduced the communicating sequential process calculus (CSP) which is used as the foundation for the concurrency features of Go, Kotlin, Crystal, and Flix [Hoare 1978].

Type and Effect Systems. Nielson et al. present a type and effect system for a calculus with channels and processes [Nielson et al. 2015]. We build on their work and extend it with the select construct which is strictly more powerful than ordinary get and put operations, since it is able to wait on multiple channels simultaneously.

Compiler Testing. Yang et al. present Csmith, a program generation technique to discover bugs in C compilers [Yang et al. 2011]. Csmith generates a C program, compiles it using several compilers,

runs the executables, and compares their results. If the results differ there is likely a bug. In our work, we use termination as the criteria to determine if an execution was correct, but given that Go, Kotlin, Crystal, and Flix have similar semantics, it would be interesting to investigate if differential testing can be applied to these languages.

Le et al. present the idea of equivalence modulo inputs (EMI), a technique to transform a program while keeping it semantically equivalent to the original program when given the same input [Le et al. 2014]. This is done by modifying unreachable parts of the program, and if the execution of the mutated program changes, there is likely a bug in the compiler. We speculate that applying the EMI technique to our setting will be difficult due to non-determinism.

Pałka et al. present a technique to generate random programs that are type correct by construction [Pałka et al. 2011]. The key idea is to read the type rules "backwards" to iteratively construct a type correct program in a goal-directed, bottom-up fashion.

Midtgaard et al., building on the work of Pałka et al., present a type and effect system for OCaml that captures evaluation order dependence of function arguments [Midtgaard et al. 2017]. Using the type and effect system, Midtgaard et al. implement a program generator that constructs programs where argument evaluation order is immaterial. The authors apply the technique to two OCaml backends finding several bugs in the process. Our work is inspired by Midtgaard et al., but our calculus and effect system are significantly richer.

Felleisen et al. present PLT Redex, a domain-specific language designed for specifying and debugging operational semantics [Felleisen et al. 2009]. PLT Redex can generate expressions that satisfy the grammar of a language, but this generation is not driven by the type and effect system of the language, and can generate many ill-typed terms. Fetscher et al. improve on this by generating well-typed terms from the typing judgments of a Redex specification [Fetscher et al. 2015].

Program Fuzzing. Miller et al. introduced blackbox fuzzing to automatically find bugs in programs, but in a way that is typically limited to shallow bugs [Miller et al. 1990]. Since then, fuzzing has become a popular technique for automatic test generation, and has been used to test compilers [Cummins et al. 2018; Dewey et al. 2014, 2015; Holler et al. 2012; Köroglu and Wotawa 2019; Lidbury et al. 2015].

Godefroid et al. introduced grammar-based whitebox fuzzing to overcome the limitations of fuzzing for programs that operate in multiple phases [Godefroid et al. 2008]. Our technique also relies on a grammar of *valid* programs. However, a whitebox approach is not a good fit for programs using channels, as the input of a whitebox fuzzer is a "sequential deterministic program under test" [Godefroid et al. 2008].

Zalewski introduced American Fuzzy Lop[7], a popular coverage-guided mutation-based fuzzer that has been used to implement multiple other fuzzers [Böhme et al. 2016; Lemieux et al. 2018; Stephens et al. 2016]. In contrast to these tools, our approach does not aim to maximize a property of the program under test such as coverage or time, but rather to generate non-deterministic, but terminating programs.

Mathis et al. introduced parser-directed fuzzing, a lightweight whitebox fuzzing approach that targets input parsers [Mathis et al. 2019]. In our work, we require not just a syntactically valid program, but a one that is type-correct and always terminates.

8 CONCLUSION

We have presented an automatic program generation technique to test programming language implementations of channel and process-based concurrency. The key idea is a type and effect system that describes programs composed of processes that communicate over channels, but

[7] http://lcamtuf.coredump.cx/afl

always terminate. The type and effect system can be used to drive random program generation to fuzz the implementation of concurrency runtimes. Using the effect language, we have implemented a program generator and applied it to Flix, Kotlin, Crystal, and Go. We discovered two previously unknown bugs in Flix, and reproduced two bugs; one in Crystal and one in Kotlin.

This paper has three important lessons: First, programming language implementors can use our fuzzing technique to test their concurrency runtimes during development and maintenance. We think such testing techniques are important since channel and process-based programming models seem to be gaining traction and are being implemented in many languages. Yet, such implementations are often tricky to get right. Second, type and effect systems can be used to drive program generation especially when the semantics of the program we wish to generate are complex. Third, We think that concurrency runtimes and programming language runtimes in general is an interesting domain for future fuzzing techniques.

REFERENCES

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1032–1043. https://doi.org/10.1145/2976749.2978428

Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. 95–105. https://doi.org/10.1145/3213846.3213848

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 725–730. https://doi.org/10.1145/2642937.2642963

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 482–493. https://doi.org/10.1109/ASE.2015.65

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. Mit Press.

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 383–405. https://doi.org/10.1007/978-3-662-46669-8_16

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 206–215. https://doi.org/10.1145/1375581.1375607

Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 445–458.

Yavuz Köroglu and Franz Wotawa. 2019. Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing. In *Proceedings of the 14th International Workshop on Automation of Software Test, AST@ICSE 2019, May 27, 2019, Montreal, QC, Canada*. 28–34. https://doi.org/10.1109/AST.2019.00010

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 216–226.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. 254–265. https://doi.org/10.1145/3213846.3213874

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 65–76. https://doi.org/10.1145/2737924.2737986

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 548–560. https://doi.org/10.1145/3314221.3314651

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven quickchecking of compilers. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 15.

Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel) (Lecture Notes in Computer Science)*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.), Vol. 1710. Springer, 114–136. https://doi.org/10.1007/3-540-48092-7_6

Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. Springer.

Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 91–97.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

TIOBE. 2020. TIOBE Index for August 2020. https://www.tiobe.com/tiobe-index//.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.