An Adventure in Symbolic Execution (extended abstract)

Gergő Érdi
Standard Chartered Bank
gergo@erdi.hu

ABSTRACT

ScottCheck is a verifier for text adventure games based on symbolic execution. Its implementation is based on an idiomatic concrete interpreter written in Haskell. Even though Haskell is a general-purpose functional language, the changes required to transform it into a symbolic interpreter turned out to be fairly small.

ACM Reference Format:
Gergő Érdi. 2020. An Adventure in Symbolic Execution (extended abstract). In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL '20).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Interactive fiction is a format of computer programs that can broadly be described as a textual back-and-forth between a human player and an automated world simulation. A subset of them, text adventure games, are characterized by having explicit win and failure states, tracing their lineage back to 1976's *Colossal Cave Adventure*. The usual implementation strategy of text adventure games is to use a domain-specific language for describing the specifics of individual game worlds, and then create interpreters for this language, targeting whatever platforms the game is to be released on.

An adventure game is essentially a puzzle, and a puzzle that has no solution can be a frustrating experience for the player. Starting from the initial state, there should always be a way to get to a winning state.

We can use symbolic execution of the game world description to check if there is a sequence of player inputs that result in a winning end state. One approach is to take an off-the-shelf interpreter, and compile it into symbolically executed code: our interest in this topic was sparked by previous work[3] in which the `scottfree` interpreter, itself is written in C, is compiled with SymCC[6] into symbolic form. Another possible approach would be to implement the interpreter in an environment with ambient symbolic evaluation, such as Rosette[8].

Our work explores the low-tech approach of using the general-purpose functional programming language Haskell, implementing a concrete interpreter idiomatically, and then changing it just enough to be able to execute it symbolically and pass it to an SMT solver to find input that satisfies the winning condition.

2 STRUCTURE AND INTERPRETATION OF ADVENTURE GAMES

Following previous work in [3], we focus on the format of Scott Adams's text adventure games, originating from his first game, 1978's *Adventureland*. The game world is modeled as a space of discrete *rooms*, connected with each other in the six cardinal directions. Each room comes with a textual description to present to the player. The rooms also contain *items*, which are objects the player can manipulate. Most notably, items can be moved around either directly by the player (by taking them, moving to another room and dropping them), or by various world simulation events.

Beside the data describing rooms, their connections, items, and their starting locations, the game files also contain *scripts* in a simple language. Each script line consists of a set of *conditions* (e.g. *is item #4 currently in the same room as the player character?*) and a sequence of *instructions* (e.g. *swap locations of items #5 and #2*).

Player input is processed by parsing against two small dictionaries of verbs and nouns. Script lines can either be automatic, executing in every turn regardless of user input; or keyed to some combination of a verb and a noun index.

Unlike more elaborate winning conditions in other games, the Scott Adams adventure games all uniformly use the concept of collecting *treasure* items as the goal. One room is marked as the *treasury*; the `SCORE` command shows the current number of treasures in the treasury, and finishes the game if it is equal to the number of all treasure items in the game.

3 MONAD TRANSFORMERS FOR CONCRETE INTERPRETERS

The concrete interpreter is based on the traditional stack of monad transformers[4]: a *Reader* giving access to the world description, a *Writer* collecting the output messages, and a *State* consisting of the current item locations, including the location of the player-controlled avatar:

```
type GameData = ...
data St = St
  { currentRoom :: Int16
  , itemLocations :: Array Int16 Int16
  }
type Engine = ReaderT GameData (WriterT [String] (State S))
```

Each turn of the game takes three steps: world simulation, user input, then response to the player input. This means the interaction model itself is monadic as well: the player can see all previous output before deciding on their next input. We implement this

Gergő Érdi is employed by Standard Chartered Bank. This paper has been created in a personal capacity and Standard Chartered Bank does not accept liability for its content. Views expressed in this paper do not necessarily represent the views of Standard Chartered Bank.

IFL '20, September 2020, The Internet,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

structure by doing the first and the third step inside *Engine*. This means we have a purely functional core, with an external, thin layer of IO only to take care of showing output and getting input.

4 SYMBOLIC EXECUTION AND PUZZLE TESTING

To turn the interpreter into a solver, we change it from concrete to symbolic execution. SBV[2] is a Haskell library providing types that support symbolic evaluation. The resulting symbolic constraints are then passed to an SMT solver; in our case, we use the open-source solver Z3[1].

This code transformation is surprisingly straightforward and painless. The solver-specific parts begin only after the game data has been read and parsed; we can keep the parser as-is. The interpreter state is changed to use SBV's symbolic types (prefixed with an *S*):

```
data S = S
  { currentRoom :: SInt16
  , itemLocations :: Array Int16 SInt16
  } deriving (Generic, Mergeable)
```

Here, *SInt16* is SBV's 16-bit integer type. *itemLocations* is still a static array of symbolic values, since the set of items remains constant during play-through for a given game: only the locations of items (i.e. the elements of the array) change. We let data-generic instance deriving[5] write the instance for SBV's *Mergeable* typeclass; this typeclass enables branching in symbolic results, which is crucial when interpreting conditions that check item locations.

Arithmetic works without change, since SBV types implement the *Num* typeclass. Because in standard Haskell, operators like == are not overloaded in their return type, the Boolean operators have SBV-specific versions.

This takes care of data. For control, we can write *Mergeable* instances for *ReaderT*, *WriterT* and *State* since these are all just typed wrappers around bog-standard function types. This allows us to define symbolic versions of combinators like *when*, or **case** with literal matches. Thus, we can build up the kit that enables writing quite straightforward monadic code, just by replacing some combinators with their symbolic counterpart. Here's an example of the code that runs a list of instruction codes in the context of their conditions; even without seeing any other definitions, it should be fairly straightforward what it does:

```
execIf :: [SCondition] → [SInstr] → Engine SBool
execIf conds instrs = do
  (oks, args) ← partitionEithers ⟨$⟩ mapM evalCond conds
  let ok = sAnd oks
  sWhen ok (exec args instrs)
  return ok
```

5 NOTIONS OF ADVENTURING AND MONADS

At this point, we have a symbolic interpreter which can consume user input line by line:

```
stepPlayer :: (SInt16, SInt16) → Engine (SMaybe Bool)
stepPlayer (verb, noun) = do
  perform (verb, noun)
  isFinished
```

The question then is, how do we keep turning the crank of this and let the state evolve for more and more lines of symbolic input, until we get an *sJust sTrue* result, meaning the player has won the game? SBV's monadic *Query* mode provides a way to do this incrementally: at each step, fresh free symbolic variables standing for the next input line are fed to the state transition function, yielding a new symbolic state and return value. Then, satisfiability of this new return value being *sJust sTrue* is checked with the SMT solver; if there's no solution yet, we keep this process going, letting the next *stepPlayer* call create further constraints. Furthermore, since the *Query* monad allows IO, we can recover the behavior of our original, concrete interpreter. Instead of using free variables for the input at each step, we read and parse the player's input into *SInt16* variables containing concrete values. Since the only potentially symbolic arguments to the *Engine* are the player inputs, if those are concrete, everything further downstream will also be concrete. In particular, the output messages, while their type is *SString*, contain concrete values which can be extracted into the standard *String* type for printing. This allows the same interpreter implementation to be used for both solving and interactive playing.

6 CONCLUSION

The full code of our symbolic Scott Adams adventure game interpreter is available under the terms of the MIT license from https://github.com/gergoerdi/scottcheck.

The combination of Haskell, a general-purpose functional language, and SBV, a library for SMT-based verification, allowed rapid development of a symbolic interpreter with acceptable real-world performance: *ScottCheck* was written from scratch in a single week, by an author previously unfamiliar with symbolic execution techniques. In terms of performance, with the Z3 SMT solver backend, it can successfully find a solution (consisting of 14 steps) for the fourth tutorial adventure from the *ScottKit* suite[7] in three and a half minutes. Further testing with more complicated adventures remains future work.

REFERENCES

[1] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
[2] L. Erkök. SBV: SMT based verification in Haskell, 2011. URL https://leventerkok.github.io/sbv/.
[3] M. M. Lester. Program transformations enable verification tools to solve interactive fiction games. In *7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, 2020.
[4] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995.
[5] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. *ACM Sigplan Notices*, 45(11):37–48, 2010.
[6] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.
[7] M. Taylor. ScottKit - a toolkit for Scott Adams-style adventure games, 2009. URL https://rdoc.info/github/MikeTaylor/scottkit.
[8] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.