Playing by the Rules: A Formal Specification of BoGL's Loops and Type System

By
Alex Grejuc

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 21, 2021
Commencement June 2021

AN ABSTRACT OF THE THESIS OF

Alex Grejuc for the degree of Honors Baccalaureate of Science in Computer Science presented on May 21, 2021. Title:
Playing by the Rules: A Formal Specification of BoGL's Loops and Type System

Abstract approved:

Martin Erwig

BoGL is a programming language created for the purpose of computer science education that is specific to the domain of board games. Although there is a language grammar and an existing implementation that is currently used by students, a complete and formal language standard does not yet exist. In the format of inference rules, this thesis specifies the language's while loops, which are at the intersection of functional and imperative programming, as well as its type system. This formal specification has helped resolve language design issues, continues the effort to standardize the language, and supports future work on it, such as the creation of systematic and beginner-friendly type error message reporting.

Key Words: Type Systems, Programming Languages, Functional Programming, Domain Specific Languages, Educational Programming Languages

Corresponding e-mail address: grejuca@oregonstate.edu

©Copyright by Alex Grejuc
May 21, 2021

Playing by the Rules: A Formal Specification of BoGL's Loops and Type System

By
Alex Grejuc

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 21, 2021
Commencement June 2021

Honors Baccalaureate of Science in Computer Science project of Alex Grejuc presented on May 21, 2021.

APPROVED:

Martin Erwig, Mentor, representing Computer Science

Eric Walkingshaw, Committee Member, representing Computer Science

---

Danila Fedorin, Committee Member, representing Computer Science

---

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

Alex Grejuc, Author

ACKNOWLEDGEMENTS

Thank you to my parents, who are always ready and willing to help me and who encouraged me to read and to value my education. They have done more for me than I can ever express and that has helped me get to where I am today. Thank you also to Rebeca, Vasile, and Daniel for growing up with me and for helping me navigate the world as a Romanian-American.

Thank you to my advisor, professor Martin Erwig. Your willingness to engage with undergraduate students and to take them seriously is remarkable. Thank you for hiring me as a research assistant. Thank you for all the time and effort you have invested into my education and research, especially for answering my many questions and for always providing clear and direct feedback.

Thank you to professor Eric Walkingshaw, not only for serving on my committee, but also for introducing me to the field of programming languages. The care and enthusiasm you put into teaching CS 381 sparked my interest in the field. Without your work, I do not think I would have pursued this path. Also, the effort you put into teaching CS 581 and 583 expanded my knowledge of the field and improved my abilities as a functional programmer, both of which directly helped me with this thesis.

Thank you to Daniel (Danila) Fedorin not only for serving on my committee, but also for your support in entering the programming languages group at OSU as the only other undergraduate student at the time. Thank you for your willingness to discuss and answer my many questions about computer science over our years at OSU together.

Thank you to the other students who helped get the BoGL alpha version off the ground. Thank you to the other group, made up of Johannes Freischuetz, Thomas Croll, Benjamin Warschauer, Fern Bostelman-Rinaldi, and Jackson Bizjak for contributing to the early growth of the language. Thank you to Kai Gay for doing

much of the early work on the current language implementation, thank you to Calvin Gagliano for creating the initial web interface, and thank you to Ben Friedman for continuing to develop the software and design the language over the last year with me.

Finally, thank you to all my friends for being a source of joy and comfort. I would especially like to thank Jonah and Daniel for being by my side throughout college as well as Anna and Kyle for their support even as we've lived in different cities.

Contents

1 Introduction **1**

2 Background **1**

2.1 A Brief History of BoGL 2

2.2 The BoGL Implementation 3

2.3 Preliminaries 3

2.4 BoGL: The Board Game Language 6

3 While Loops **9**

3.1 Transformation 10

3.2 Semantics 16

3.3 Renaming 17

3.4 Additional Examples 18

4 Type System **19**

4.1 Syntax 19

4.2 Typing Rules 20

4.2.1 Freshness and Explicit Values 21

4.2.2 Type Expansion 22

4.2.3 Type Definitions 23

4.2.4 Typing Environment Initialization 24

4.2.5 Type Signatures and Equations 25

4.2.6 Subtyping 26

4.2.7 Expressions 28

4.3 Value Extraction 32

4.4 Type Checking 33

5 Related Work **34**

6 Conclusion 34

7 Appendix 36

7.1 BoGL Grammar 36

7.2 Listing of Built-In Functions and Values 37

7.3 A Complete Example Program 37

1 Introduction

BoGL[1] (Board Game Language) is an educational domain specific programming language that models the subject of board games. The primary purpose of the language is to teach concepts in computer science to beginners. This thesis contains a formal specification of how local state is created for BoGL while loops as well as a formal specification of the BoGL type system. The purpose of these formal specifications is threefold: to uncover problems with the existing language and to devise a correct formalization of it[2]; to provide a modular, declarative, and implementation-agnostic specification of BoGL's while loops and its type system so that it is well-defined and so that others may understand it[3]; and to support further efforts to formalize and extend BoGL.

Section 2 covers some background information, both about the BoGL language as well as preliminary topics needed to understand the formal specification in the rest of the thesis. Section 3 describes BoGL's while loops, including the syntactic transformation that makes them possible, their semantics, as well as some examples of how they are used. Section 4 describes the BoGL type system with typing rules. Section 5 touches on related work and Section 6 concludes this thesis. Section 7 contains the BoGL grammar, a listing of built-in functions and values, and an example program of the game tic-tac-toe.

2 Background

BoGL incorporates many ideas from functional languages. Its syntax is similar to Haskell [7], programs are written in an equational, expression-based style, and variables are immutable. However, for the sake of simplicity, it does not have first class

[1] Pronounced with a long o, as in "no."

[2] Indeed, work on this thesis led to changes to the type system and to while loops

[3] That is, without having to read through code whose quality is affected by the realities of life. The corresponding prose descriptions will hopefully be elucidating for readers unfamiliar with the metalanguage of type systems.

functions, nor does it have pattern matching. The complete grammar and a listing of built-in functions and values can be found in Section 7. A description of the language, including its syntax and semantics, can be found in [5]. It should be noted that this document was created at a time when tuple let expressions were not part of the language. Section 2.4 briefly covers the aspects of BoGL that are of particular relevance to this thesis.

2.1 A Brief History of BoGL

Work on the language began in September 2019 as a project for the Oregon State University course CS 461: Senior Software Engineering. Professor Martin Erwig recruited two groups of students to help design and implement the language. Each group concluded their project at the end of the academic year in June 2020 in the form of a prototypical interpreter. One of these interpreters went on to be used in pilot sessions with students of Linus Pauling Middle School and even in a special version of the OSU course CS 160: Computer Science Orientation. At the time of writing, 160 students have used the language.

However, given that the language was rapidly evolving, the only part of the language to be formally specified at the end of the project phase was its syntax. The BoGL type system and semantics were only textually described and implemented as Haskell code. This was due to two practical reasons. The first is that informal specification enables rapid development, which provides answers to known unknowns and allows the discovery of unknown unknowns in a design space. So much changed about the language since its inception that a formal specification at this stage would have long become useless. The second reason is that undergraduate computer science students at Oregon State are not expected to understand the metalanguages used for formal specification of type systems or semantics.

Unsurprisingly, this informal specification resulted in in-the-moment choices about language implementation for underspecified features as well as bugs. Additionally,

student and educator feedback as well as continued language design led to changes in the specification, including tweaks to the BoGL types; the addition of tuple projections, and loop binding contexts. With this set of changes to the concept of BoGL, it has reached a stable phase that is suitable for a more rigorous specification.

2.2 The BoGL Implementation

There is a Haskell implementation of the language, although it should be noted that it does not exactly match the specification that is in this document, as it was created first.[4] There is also a front end in the form of a text editor and a REPL (read-evaluate-print loop), which is written in TypeScript.[5] Together, these two form a web-accessible implementation of the language, which is how students currently write an execute BoGL programs.[6] There is also a BoGL documentation page.[7]

Figure 1 displays an in-progress game of tic-tac-toe on the web interface. Each time a user enters an input expression in the REPL on the right, it is evaluated in the context of the program on the left. In Figure 3, the value equation `result` has been entered. Evaluating it requires user input, which the REPL requests and provides to the BoGL interpreter.

2.3 Preliminaries

Rule systems, also called formal or deductive systems, are a syntactic tool for deriving theorems; their typical application in programming languages theory is for the definition of semantics and type systems [8]. There are other applications as well, such as the definition of a syntactic transformation in Section 3.1. They are defined with *inference rules*, which are themselves described with *judgments* or assertions.[8]

[4]https://github.com/The-Code-In-Sheep-s-Clothing/bogl

[5]https://github.com/The-Code-In-Sheep-s-Clothing/bogl-editor

[6]https://bogl.engr.oregonstate.edu/

[7]https://bogl.engr.oregonstate.edu/tutorials/

[8]Typically, they are actually defined with inference rule *schemas*, that is, inference rules defined in terms of metavariables from which concrete inference rules can be created. Perhaps for brevity,

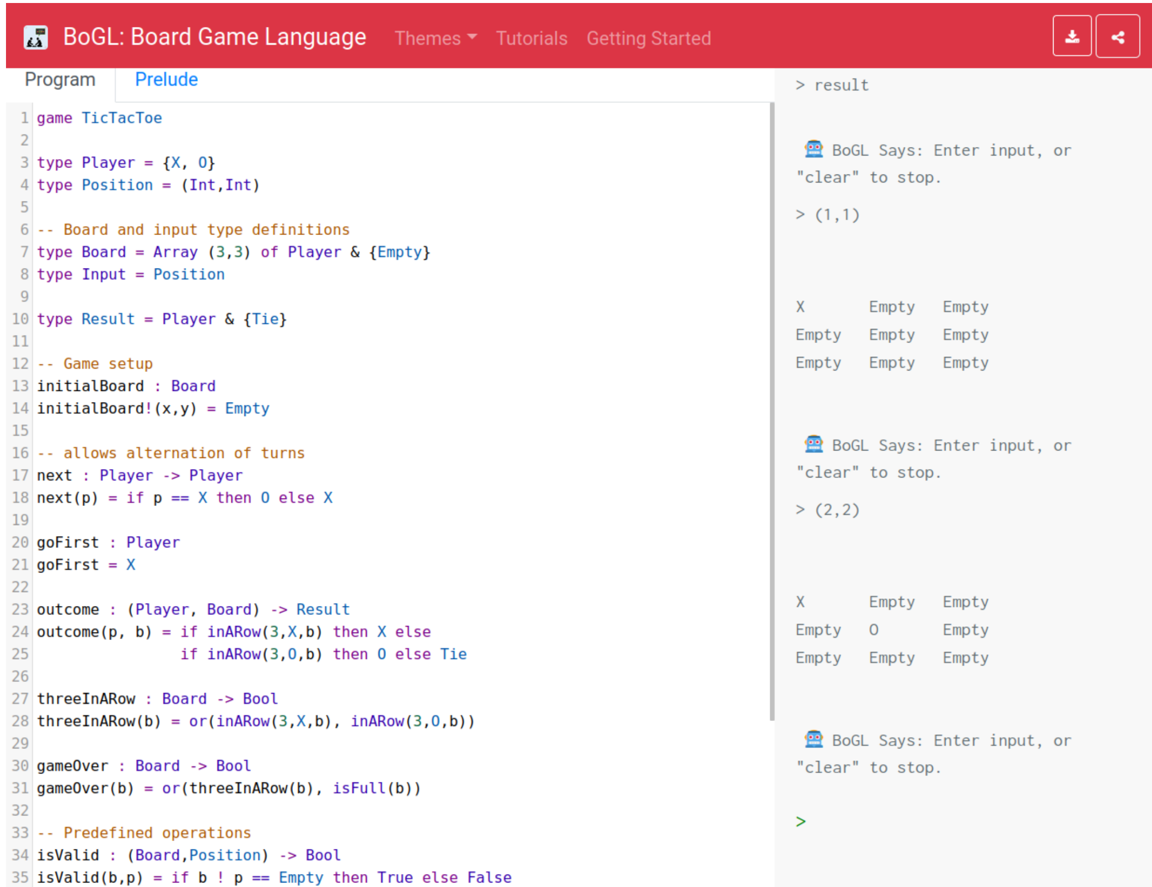

Figure 1: The online BoGL interface

An inference rule consists of a potentially-empty set of premise judgments that must all be satisfied for a conclusion judgment to hold and is typically read as an if-then statement. One way of representing them is with a horizontal line, above which are the premises and below which is the conclusion. For example, in Figure 2, R-Zero can be understood as an axiom that states that 0 is a natural number and R-Succ can be understood as stating that, if x is a natural number, then $x + 1$ is also a natural number. For a brief introduction to this and other notation in programming languages theory, see [10].

$$\textsc{R-Zero}\;\frac{}{0 \in \mathbb{N}} \qquad\qquad \textsc{R-Succ}\;\frac{x \in \mathbb{N}}{x + 1 \in \mathbb{N}}$$

Figure 2: Example Inference Rules

A *type* is a collection values related by a common behavior or property. A *typing rule* is a formal specification that assigns a type to a *term*, such as an expression or a function, of a language. A *type system* is specified with a collection of such rules, which may be defined using a rule system. For the purpose of practical programming language implementation, a type system is a static (occurring before program execution) and conservative (possibly rejecting valid programs) approach for determining the absence of a set of programming errors. A *type checker* is an implementation of a type system and is a common component of a compiler or interpreter. For an overview of type systems, including a discussion of some of the above terms, descriptions of the relevant notation, and examples, see [2]. For a comprehensive introduction to types and programming languages, see [8].

this distinction is often ignored and rule schemas are themselves referred to as rules, a convention that is followed in this thesis.

2.4 BoGL: The Board Game Language

BoGL contains the predefined types `Int` and `Bool`. It also contains the reserved type names `Board`, `Content`, and `Input`, which are only accessible in programs that contain their respective type definitions. To support the creation of more sophisticated types, BoGL contains tuple type constructors, enumeration type constructors, extended type constructors, and board type constructors. These may be used in a *typedef* to define a new type and assign it a name.

Tuple type constructors in BoGL use the same parenthesis syntax as tuple expressions, and they must be written in terms of (tuples of) defined types. For example, `({X, O}, {Player1, Player2})` is not a valid tuple type, but `(Piece, Player)` is, given that definitions for the components have been provided.

Enumeration type constructors allow the introduction of new *symbolic values*, that is, ones that are referred to by a *Name*. These exist in BoGL to allow a conceptually lightweight application of naming and abstraction. For example, the type definitions below might appear in the definition of a game of tic-tac-toe.

```
type Piece  = {X, O}
type Result = {XWins, OWins, Stalemate}
```

A symbolic value that appears in an enumeration type, such as `X` and `O` above, is considered *declared*. The type system requires that the values of an enumeration type do not clash with existing type names or symbolic values. For example, `BoardContent`, which might represent all of the values that appear on a tic-tac-toe board, is an invalid type definition given the earlier definition of `Piece` since it redeclares `X` and `O`.

```
type BoardContent = {X, O, Empty}
```

This problem can be solved by extended type constructors, which allow the creation of tagged union types, that is, ones that are composed of two or more types and admit values of any one of those types. These may be used to extend existing type definitions. For example, given `Piece`, `BoardContent` below is a valid type definition.

```
type BoardContent = Piece & {Empty}
```

The *base* of an extended type, that is, the left-most type, may be any type. Any other types in an extended type must either be literal enumerated types, the names of declared enumerated types, or the names of declared types that are composed solely of enumerated types. For brevity, these will all be referred to as enumerated types in what follows. This restriction means that the type definitions below are not valid.

```
type IntBoard   = Int & Board                    -- Invalid!
type PairTriple = (Int, Int) & (Int, Int, Int) -- Invalid!
```

A board type definition may appear at most once in a program. It declares the size of a board as well as the type of values that it may store. These constraints are then applied to every board value in the program. Ignoring the earlier type definition, the type definition below creates a board type of three rows and three columns, which may store the values `X`, `O`, or `Empty` for a game of tic-tac-toe. The board type definition also implicitly defines the type synonym `Content` for the type expression that follows `of` and the values `height` and `width` for the dimensions of the board. In the example below, `Content` is a synonym for `Piece & Empty` and the bindings `width = 3` and `height = 3` are introduced.

```
type Empty = {E}
type Piece = {X, O}
type Board = Array (3,3) of Piece & Empty
```

BoGL has while loops that are at the intersection of functional and imperative languages. These loops can only modify a local state and, since they are expressions, they evaluate to a value. They operate in a way that is is reminiscent of the state monad. That is, variables are threaded behind the scenes to simulate state, which allows BoGL to mimic the stateful looping constructs found in imperative languages. For example, `loop` is a function that plays a game until a termination condition is reached, which is written in terms of the user-defined functions `gameOver` and `tryMove`.

```
loop : (Player, Board) -> (Player, Board)
loop(p,b) = while not(gameOver(b)) do tryMove(p,b)
```

This feature allows repetition to be taught without recursion, although it may also be be viewed as recursion with training wheels, given that BoGL loops can be mechanically translated to recursive BoGL functions. They are described in detail in Section 3.

User input in BoGL is provided through the input variable, which changes over time during program execution. It is both modeled and implemented as a queue of expressions that are provided to the BoGL interpreter in addition to the program itself. It may only be an *int* or a *Value* and the program must have a type definition for `Input` before it can be used. Prior to execution, the entire input queue is type checked to ensure that it has the type defined by `Input` in the program. For example, given the input queue `[1,2,3]`, `sum` below dequeues three times and evaluates to `6`.

```
type Input = Int

sum : Int
sum = input + input + input
```

Execution terminates with an error if more input is used than is available during execution, such as `sum` with an input queue of `[1,2]`. However, a front end REPL may hide this detail from the user and instead request more input and re-execute the program, as the current implementation does.

Similar to the way that input is simulated by providing a queue to the BoGL interpreter alongside the program, boards are produced as an output of the BoGL interpreter for each use of the `place` function. It is up to the interpreter interface to handle this. The current implementation filters and displays this to the user, as can be seen in Figure 1.

3 While Loops

An imperative loop updates state variables until a condition is met. The functional counterpart in BoGL simulates this by shadowing variables during execution. These variables form a *loop state* and are only shadowed within the scope of their respective loop, which iterates until its condition is false and then returns the values of its state.

For example, `ten` is a value equation containing a loop that increments to `10`. It has a loop state initialized to `1`. This state is updated with each iteration until its state is `x = 10`, at which point the condition is `False` and the value of the state is returned. On the other hand, `ten'` has a loop that iterates until its state is `x = 9`. Outside the loop, the binding `x = 1` has not been changed, so it too evaluates to `10`.

```
ten : Int
ten = let x = 1 in while x < 10 do x + 1

ten' : Int
ten' = let x = 1 in (while x < 9 do x + 1) + x
```

Let expressions are not the only way to create loop state; it is also possible to do so with a function equation. For example, `succToTen` below generalizes the `ten` example above to a function that may begin counting at any integer. It will return an argument unchanged if it is ten or greater. The call `succToTen(2)` will initialize the loop state to `x = 2` and evaluate to `10`.

```
succToTen : Int -> Int
succToTen(x) = while x < 10 do x + 1
```

The above examples raise the question of what happens to loop state when function equations and let expressions are mixed. `stepToTen` below, which generalizes `succToTen` with a parameterized step value, is a function that demonstrates the answer: more shadowing. The call `stepToTen(1, 2)` evaluates to `11`. This occurs because `(m = 1, step = 2)` is shadowed by `x = 2`, which the let expression intro-

duces.[9]

```
stepToTen : (Int, Int) -> Int
stepToTen(m, step) = let x = m in while x < 10 do x + step
```

However, it is not the loop state that is shadowed, but rather the *binding context*. This is a notion that can be understood as a loop state that is passed around during syntactic transformation. That is, state is used during type checking and evaluation and is only associated with one loop, while a binding context is detected during syntactic transformation, may be used to create a loop state, and may be associated with many loops. The example `fifteen` below, which evaluates to `15`, clarifies this distinction.

```
fifteen : Int
fifteen = let x = 1 in
          (while x < 10 do x + 1) + (while x < 5 do x + 1)
```

`let x = 1` introduces a single binding context that spans both loops. Each expression is transformed to have its own loop state of `x = 1`, which enables it to be type checked and evaluated. The left loop evaluates to `10` and the right loop evaluates to `5`.

These concepts and the syntactic transformation that uses them are explained more precisely in Section 3.1. The semantics of while loops is described in Section 3.2. To further clarify the behavior of while loops, more examples are presented and described in Section 3.4.

3.1 Transformation

Loop state is not explicitly included in the concrete syntax of BoGL. Instead, it is encoded in terms of let expressions by a a syntactic *loop transformation* from an implicit binding context, which is a mapping from variable names x to expressions

[9]This may be the source of nontermination errors. An implementation may detect conditions that are not written in terms of loop state variables and warn the user about this.

e. Binding contexts are lexically scoped and are introduced by function equations and let expressions; they are references to the names introduced by these constructs. Their syntax is shown in Figure 3.

$$s \quad ::= \quad \{x_1 = e_1, \ldots, x_n = e_n\} \qquad (\textit{potentially empty context, } n \geq 0)$$

Figure 3: Binding Context Syntax

A while loop that is within a binding context will then be transformed to have a local, modifiable state based on that context. This state is encoded as a let expression that directly encloses a while loop. For example, consider this function equation, which was presented earlier:

```
succToTen(x) = while x < 10 do x + 1
```

A first approach to transforming this function is to detect a binding context of $\{x = x\}$. That is, since loop transformation occurs statically, a context is created as a reference to a parameter. This context can then be used to create the loop state `let x = x`. If the function is called during execution, this state will transitively refer to the argument. Thus, the first transformed version of `succToTen` is:

```
-- Not generalizable!
succToTen(x) = let x = x in while x < 10 do x + 1
```

In fact, this approach works here. However, it does not generalize to all BoGL while loops since it is missing variable renaming. Thus, given that `x'` is not in the free variables of the loop, the actual version that the transformation produces is:

```
succToTen(x) = let x' = x in while x' < 10 do x' + 1
```

That is, the binding context $\{x' = x\}$ is instantiated as the loop state `let x' = x` and instances of `x` have been renamed to `x'`. Variables are renamed to ensure that nested loops have separate state. For now, this detail should be accepted without more explanation, which uses the loop semantics and is thus provided in Section 3.3.

As noted above, let expressions both introduce binding contexts and encode loop state. The example below is provided to show how these two roles combine. It

evaluates to 21. A let expression introduces a binding context for two loops and binds x to 1 for use outside of the loops.

```
let x = 1 in (while x < 10 do x + 1) +
             (while x < 10 do x + 1) + x
```

This loop is transformed so that each loop gets its own state and the original let binding is preserved, allowing the final x to refer to it.

```
let x = 1 in (let x' = 1 in while x' < 10 do x' + 1) +
             (let x' = 1 in while x' < 10 do x' + 1) + x
```

Loop transformation is the process of enclosing while loops in a let-bound state and renaming their variables through the use of binding contexts.[10] That is, in a binding context of the form $\{x = e\}$, stateless loops of the form while e_c do e_b are transformed to loops of the form let x = e in while e'_c do e'_b.[11] This transformation introduces the loop state let x = e. It also potentially renames x in e_c, e_b and recursively transforms those sub-expressions. This process is described below using the judgments in Figure 4.

$Q \hookrightarrow Q'$	The equation Q is loop transformed to Q'
$s, e \mapsto e'$	In the binding context s, the expression e is loop transformed to e'

Figure 4: Loop Transformation Judgments

The equation judgment does not include a context, since all equations begin in an empty context. The expression transformation judgment includes a context, since expressions may inherit (but also shadow) a binding context. Figures 5 and 6 contain the loop transformation rules for equations and expressions, respectively. These rules are applied to every equation in a program as well as to the input expression.[12]

[10] An implementation may forgo transformation and instead use a stack to create loop state with a context-sensitive parser. Renaming is not necessary if loop state is directly encoded in the syntax of while expressions. For example, instead of representing loop state as let x = 1 in while x < 10 do x + 1, a while node in the AST might instead carry its loop state without a let expression as while {x = 1} x < 10 do x + 1.

[11] The case of empty contexts is handled in Figure 6 and subsequently explained.

[12] Not to be confused with user input, which cannot contain a while loop as it is limited to literal expressions only.

Terms that do not contain value expressions, such as signatures and type definitions, are unaffected, so rules for these cases are omitted.

L-FEQ
$$\frac{\{x'_1 = x_1, \ldots, x'_n = x_n\}, e \mapsto e'}{f\,(x_1, \ldots, x_n)\ \texttt{=}\ e \hookrightarrow f\,(x_1, \ldots, x_n)\ \texttt{=}\ e'} x'_1, \ldots, x'_n \notin FV(e)$$

L-VEQ
$$\frac{\{\}, e \mapsto e'}{x\ \texttt{=}\ e \hookrightarrow x\ \texttt{=}\ e'}$$

L-BEQ
$$\frac{\{\}, e \mapsto e'}{x!\,(p_x, p_y)\ \texttt{=}\ e \hookrightarrow x!\,(p_x, p_y)\ \texttt{=}\ e'}$$

Figure 5: Loop Transformation Rules for Equations

L-FEQ is for function equations, so the binding context is created as a set of references to parameters. It should be noted that this rule only introduces a binding context. It does not instantiate it as a loop state, since the function body may contain let expressions that shadow the function binding context. The variables of the context must not be in the free variables of e to ensure that loops within loops retain a reference to parameters rather than to the variables of the outer loop. This requires that instances of parameters within loops in e in the binding context of the function equation are renamed to their respective names in the binding context. This occurs if the binding context is instantiated in L-WHILE. The two remaining rules specify that neither type of equation introduces a binding context, although their expressions must still transformed.

L-LET specifies that let bindings are used to create the context for the body of a let expression. The bound expression is transformed in the original binding context while the body expression is transformed in a new binding context that refers to this transformed bound expression. The names that are introduced in the new binding context must not be in the free variables of e, just as in L-FEQ. The additional rule for a single-variable let expression is omitted since it differs in a non-substantive way from L-LET.

L-While is where a binding context is instantiated as loop state. It uses a substitution operation, written as $[x'_1/x_1, \ldots, x'_n/x_n]e$ to indicate that occurrences of x'_i replace occurrences of x_i in e. To transform a loop, its component expressions must be recursively transformed, variables must be renamed according to its binding context, and it must be enclosed in a let expression created from that binding context.

L-Empty handles the case of a while loop that is written in an empty binding context. In this case, it is not possible to create a loop state for the expression, so it is returned unchanged. The type system must reject while loops of this form, as explained in Section 4.2.

The remaining cases in Figure 6 are a straightforward structural threading of the context to subterms. Neither a syntactic transformation nor a context change occurs at the top level in them.

$$\frac{}{s, a \mapsto a}\ \text{L-ATOMIC}$$

$$\frac{s, e_1 \mapsto e_1' \quad \dots \quad s, e_n \mapsto e_n'}{s, (e_1, \dots, e_n) \mapsto (e_1', \dots, e_n')}\ \text{L-TUPLE}$$

$$\frac{s, e_1 \mapsto e_1' \quad \dots \quad s, e_n \mapsto e_n'}{s, f(e_1, \dots, e_n) \mapsto f(e_1', \dots, e_n')}\ \text{L-APP}$$

$$\frac{s, e_1 \mapsto e_1' \quad s, e_2 \mapsto e_2'}{s, e_1\ o\ e_2 \mapsto e_1'\ o\ e_2'}\ \text{L-BINOP}$$

$$\frac{s, e \mapsto e'}{s, e\ \texttt{\#}\ m \mapsto e'\ \texttt{\#}\ m}\ \text{L-PROJ}$$

$$\frac{s, e \mapsto e'}{s, e\ \texttt{\#}\ (m_1, \dots, m_n) \mapsto e'\ \texttt{\#}\ (m_1, \dots, m_n)}\ \text{L-PROJECT}$$

$$\frac{s, (e_1, \dots, e_n) \mapsto (e_1', \dots, e_n') \quad \{x_1' = x_1, \dots, x_n' = x_n\}, e \mapsto e'}{s, \texttt{let}\ (x_1, \dots, x_n)\ \texttt{=}\ (e_1, \dots, e_n)\ \texttt{in}\ e \mapsto \texttt{let}\ (x_1, \dots, x_n)\ \texttt{=}\ (e_1', \dots, e_n')\ \texttt{in}\ e'}\ x_1', \dots, x_n' \notin FV(e)\ \text{L-LET}$$

$$\frac{s, e_1 \mapsto e_1' \quad s, e_2 \mapsto e_2' \quad s, e_3 \mapsto e_3'}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mapsto \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3'}\ \text{L-IF}$$

$$\frac{\begin{array}{c}\{x_1' = x_1, \dots, x_n' = x_n\}, [x_1'/x_1, \dots, x_n'/x_n]e_c \mapsto e_c' \\ \{x_1' = x_1, \dots, x_n' = x_n\}, [x_1'/x_1, \dots, x_n'/x_n]e_b \mapsto e_b'\end{array}}{\{x_1' = x_1, \dots, x_n' = x_n\}, \texttt{while}\ e_c\ \texttt{do}\ e_b \mapsto \texttt{let}\ (x_1', \dots, x_n')\ \texttt{=}\ (x_1, \dots, x_n)\ \texttt{in}\ \texttt{while}\ e_c'\ \texttt{do}\ e_b'}\ \text{L-WHILE}$$

$$\frac{}{\{\}, \texttt{while}\ e_c\ \texttt{do}\ e_b \mapsto \texttt{while}\ e_c\ \texttt{do}\ e_b}\ \text{L-EMPTY}$$

Figure 6: Loop Transformation Rules for Expressions

3.2 Semantics

The semantics of a while loop is defined by the evaluation judgment $E; e \Downarrow v$, which says that, in the evaluation environment E, the expression e evaluates to the value v.[13] It is specified by the two rules in Figure 7. The loop state is $\texttt{let}\ x\ \texttt{=}\ e$. The first rule says that, if the condition e_c evaluates to `False`, then the loop evaluates to the value of the loop state. The second rules says that, if e_c evaluates to `True`, then the state is updated to $\texttt{let}\ x\ \texttt{=}\ e_b$ and the loop is recursively evaluated. It should be noted that extending E with a new binding for x instead of encoding it as a let expression in the third premise would not be correct, since it would separate the loop state from the loop expression. This would result in a stuck term since it is not possible to express the false case of a while loop without access to its state. Figure 8 shows the evaluation of a while loop with this semantics. It omits the derivations of non-loop expressions.

$$\frac{E; e \Downarrow v \qquad E, x = v; e_c \Downarrow \texttt{False}}{E; \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in while}\ e_c\ \texttt{do}\ e_b \Downarrow v}$$

$$\frac{E; e \Downarrow v_1 \qquad E, x = v_1; e_c \Downarrow \texttt{True} \qquad E, x = v_1; \texttt{let}\ x\ \texttt{=}\ e_b\ \texttt{in while}\ e_c\ \texttt{do}\ e_b \Downarrow v_2}{E; \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in while}\ e_c\ \texttt{do}\ e_b \Downarrow v_2}$$

Figure 7: While Loop Semantics

$$\frac{E;\texttt{1}\Downarrow\texttt{1} \qquad E,\texttt{x}=\texttt{1};\texttt{x < 2}\Downarrow\texttt{True} \qquad \dfrac{E,\texttt{x}=\texttt{1};\texttt{x + 1}\Downarrow\texttt{2} \qquad E,\texttt{x}=\texttt{1},\texttt{x}=\texttt{2};\texttt{x < 2}\Downarrow\texttt{False}}{E,\texttt{x}=\texttt{1};\texttt{let x = x + 1 in while x < 2 do x + 1}\Downarrow\texttt{2}}}{E;\texttt{let x = 1 in while x < 2 do x + 1}\Downarrow\texttt{2}}$$

Figure 8: A partial loop evaluation

[13]This is an illustrative semantics. It does not include user input, which is necessary for evaluation. It also does not include board output, which is produced as a result of evaluating a BoGL program. These are omitted since they are incidental to understanding loop behavior.

3.3 Renaming

Renaming is needed to ensure that nested loop states are independent of each other. First consider loops `eleven` and `four`. The former increments in steps of `2` from `x = 1` to its final value `x = 11`. The latter increments in steps of `1` from `x = 1` to `x = 4` until the condition reaches `11 + x = 15`.

```
eleven : Int
eleven = let x = 1 in while x < 10 do x + 2

four : Int
four = let x = 1 in while 11 + x < 15 do x + 1
```

We then combine the two into `nested`, noting that two let binding contexts have intentionally been merged into one function binding context. The previous two examples were provided to clarify how `nested` executes rather than as exact components of it. The loop in `nested` is similar to the one in `four` except that the constant `11` has been replaced by a loop similar to the one in `11`.

```
nested : Int -> Int
nested(x) = while (while x < 10 do x + 2) + x < 15 do x + 1
```

It can then be transformed without renaming. That is, $\{x = x\}$, a binding context without renaming, creates loop states of `let x = x` for both the inner and outer loops.

```
nested : Int -> Int
nested(x) =
  let x = x in while
 (let x = x in while x < 10 do x + 2) + x < 15 do x + 1
```

However, with this transformation, the second loop state depends on the first. That is, `x` within the condition loop will have an initial loop state based on `x` in the outer loop rather than its binding context. We can observe this after one iteration of the outer loop with the call `nested(1)`.

```
 let x = x + 1 in while
(let x = x in while x < 10 do x + 2) + x < 15 do x + 1
```

The parameter `x` is bound to the argument `1`. However, due to the binding `let x = x + 1`, the inner loop does not begin at `1`, but at `2` and thus evaluates to `10` rather than `11`. This causes `nested(1)` to evaluate to `5` rather than `4`. If we instead create a binding context with a fresh variable `x'`, the transformed version is:

```
nested : Int -> Int
nested(x) =
  let x' = x in while
 (let x' = x in while x' < 10 do x' + 2) + x' < 15 do x' + 1
```

After one iteration of the outer loop with the call `nested(1)`, we then have:

```
 let x' = x' + 1 in while
(let x' = x in while x' < 10 do x' + 2) + x' < 15 do x' + 1
```

In this case, the inner loop reference to the parameter is preserved and it still begins at `1` and evaluates to `11`. Thus, with renaming, `nested(1)` evaluates to `4`.

Whether nested loop states should affect each other or not is a language design question that should take into account the guiding value of BoGL, which is that it should be the best choice for teaching computer science. If it is better to allow loop states to be shared, renaming can be removed from L-While.

3.4 Additional Examples

The previous examples in Section 3 are all intentionally arithmetic for the sake of uniformity, however, BoGL loops are not restricted to this purpose. For example, they may be used to get an input that matches some criteria, such as the equation below, which loops until the input is `42`. The initial value `x = 0` is not used, but it must provided so that a loop state can be created.

```
theAnswer : Int
theAnswer = let x = 0 in while x /= 42 do input
```

Additionally, as the example in Section 2.4 indicates, a common use of this feature is to create game loops that run until an end condition is reached, which itself often requires looping over the values of a board to check for some property. For more examples of while loops, readers are referred to the online tutorials.[14]

4 Type System

BoGL is statically typed, has subtyping, and does not have parametric polymorphism [8]. Section 4.1 introduces the syntax and judgments, Section 4.2 describes the type system with typing rules, Section 4.3 describes the problem of value extraction, and Section 4.4 explains how these rules can be used to type check a program.

4.1 Syntax

A grammar for the BoGL types described above is shown in Figure 9. N ranges over upper and lowercase names, including the names of the atomic types. V and W range over uppercase names and are only used to refer to symbolic values. f and x range over *name*, e over *expr*, m over *int*, o over *binop*, and p over *pos*. S ranges over sets of *Names* as well as over Γ and Δ.

$$
\begin{array}{rcll}
b & ::= & N \mid \texttt{(}b\texttt{,}\ \ldots\texttt{,}\ b\texttt{)} & (\textit{base type}) \\
c & ::= & \{V\texttt{,}\ \ldots\texttt{,}\ V\} & (\textit{enumeration type}) \\
t, u & ::= & b \mid c \mid t\ \texttt{\&}\ N \mid t\ \texttt{\&}\ c & (\textit{type expression})
\end{array}
$$

Figure 9: Types

A grammar for the typing environments is shown in Figure 10. While it is not strictly necessary, we use two typing environments for clarity of presentation. Γ is an environment for language terms; it maps symbolic values, variable names, and function names to types. Δ is an environment for type declarations; it maps type names to type expressions.

[14] https://bogl.engr.oregonstate.edu/tutorials/

$$
\begin{array}{lcll}
\Gamma & ::= & \varnothing & (\textit{empty term environment}) \\
 & | & \Gamma,\ V : N & (\textit{value typing}) \\
 & | & \Gamma,\ x : b & (\textit{variable typing}) \\
 & | & \Gamma,\ f : b_1\ \texttt{->}\ b_2 & (\textit{function typing}) \\
\\
\Delta & ::= & \varnothing & (\textit{empty definition environment}) \\
 & | & \Delta,\ N = t & (\textit{type definition})
\end{array}
$$

Figure 10: Typing Environments

Figure 11 shows the judgments that are used to specify the type system. They are individually explained in greater detail throughout Section 4.2.

$N \star S$	N is fresh in S
$\lfloor t \rfloor = \bar{V}_n$	$\bar{V}_n$ are the explicit values of t
$\Delta^*(t) = t'$	t has a valid expansion to t' in the context of Δ
$\Delta; \Gamma \vdash \textit{typedef} \rightsquigarrow \Delta; \Gamma$	A type definition extends Δ and Γ
$\Delta, \Gamma \vdash \diamond$	Δ and Γ are valid typing environments
$\Delta; \Gamma \vdash \textit{signature} \rightsquigarrow \Delta; \Gamma$	A type signature extends Γ
$\Delta; \Gamma \vdash \textit{equation}$	An equation matches its signature in the context of Δ and Γ
$t_1 <: t_2$	t_1 is a subtype of t_2
$\Delta; \Gamma \vdash e : b$	e has type b in the context of Δ and Γ

Figure 11: Judgments

4.2 Typing Rules

The typing rules in BoGL are used to type check an input expression in the environments that are created from a program as well as the user input in an empty environment (since it may only contain literal expressions). They are specified and described below. Some of the rules below are formulated only in terms of tuples of types where a single type may also occur. For conciseness and only in the typing rules, we allow singleton tuple types and consider them to be equivalent to the type that they contain. This form is syntactically invalid in an actual BoGL program.

4.2.1 Freshness and Explicit Values

Figure 12 defines *freshness*. We write more shortly $\bar{S}_n$ for the sequence $S_1, \ldots, S_n$. $\bar{N}_k \star \bar{S}_n$ says the names $N_1, \ldots, N_k$ are *fresh* with respect to the sequences or the domains of relations $S_1, \ldots, S_n$. That is, the first rule below applies to a sequence of names (such as the explicit values of a type expression), whereas the second rules applies to the names in the domains of either of the typing environments (such as type names and variable names).

For example, $\texttt{Empty} \star \Delta, \texttt{Piece = \{X,O\}}; \Gamma, \texttt{X} : \texttt{Piece}, \texttt{O} : \texttt{Piece}$ holds since `Empty` does not appear in the domains of Δ or Γ. However, $\texttt{X} \star \Delta, \texttt{Piece = \{X,O\}}; \Gamma, \texttt{X} : \texttt{Piece}, \texttt{O} : \texttt{Piece}$ does not hold since `X` appears in the domain of Γ. The latter case occurs if a program tries to reuse a symbolic value that has already been declared.

$$\frac{N \notin S}{N \star S} \qquad \frac{N \notin dom(S)}{N \star S} \qquad \frac{N \star S_1 \qquad \ldots \qquad N \star S_n}{N \star \bar{S}_n} \qquad \frac{N_1 \star \bar{S}_n \qquad \ldots \qquad N_k \star \bar{S}_n}{\bar{N}_k \star \bar{S}_n}$$

Figure 12: Freshness

Figure 13 defines $\lfloor t \rfloor = \bar{V}_n$, which says that the (pair-wise different) values $V_1, \ldots, V_n$ are *explicitly mentioned* in type t. ϵ denotes empty sequences. For example, using the type definitions below, $\lfloor \texttt{Empty \& \{X, O\} \& \{A, B\}} \rfloor = \texttt{X}, \texttt{O}, \texttt{A}, \texttt{B}$, whereas $\lfloor \texttt{Piece} \rfloor = \epsilon$ since the type name `Piece` does not explicitly contain any symbolic values.

```
type Empty = {E}
type Piece = Empty & {X, O} & {A, B}
```

$$\lfloor b \rfloor = \epsilon \qquad \frac{}{\lfloor \texttt{\{}V_1\texttt{,} \ldots \texttt{,}V_n\texttt{\}} \rfloor = \bar{V}_n} \qquad \frac{\lfloor t \rfloor = \bar{V}_n \qquad \lfloor t' \rfloor = \bar{W}_m}{\lfloor t \texttt{ \& } t' \rfloor = \bar{V}_n, \bar{W}_m}$$

Figure 13: Explicit values

4.2.2 Type Expansion

Figure 14 contains the rules for type expansion, which converts types to a dereferenced normal form that is used for subtyping. We write $\Delta(N)$ for the type expression that is defined for type name N, and we write $\Delta^*(t)$ for the repeated substitution of names by their definitions in type t. We also use η to range over enumeration types as well as extended types made up solely of enumerated types. Type expansion fails if a type contains an undefined name or if a type is extended by anything other than an enumerated type. With these stipulations, type expansion takes on an additional role of type validity; that is, only a type that can be successfully expanded is *valid*.

$$\frac{N \in \{\texttt{Int}, \texttt{Bool}, \texttt{Board}\}}{\Delta^*(N) = N} \qquad \frac{\Delta(N) = t \qquad \Delta^*(t) = t'}{\Delta^*(N) = t'}$$

$$\frac{\Delta^*(b_1) = t_1 \qquad \ldots \qquad \Delta^*(b_n) = t_n}{\Delta^*(\texttt{(}b_1\texttt{,} \ldots \texttt{,}b_n\texttt{)}) = \texttt{(}t_1\texttt{,} \ldots \texttt{,}t_n\texttt{)}} \qquad \frac{i \neq j \implies V_i \neq V_j}{\Delta^*(\texttt{\{}V_1\texttt{,} \ldots \texttt{,}V_n\texttt{\}}) = \texttt{\{}V_1\texttt{,} \ldots \texttt{,}V_n\texttt{\}}}$$

$$\frac{\Delta^*(t) = t' \qquad \Delta^*(u) = \eta \qquad \lfloor t' \rfloor \star \lfloor \eta \rfloor}{\Delta^*(t \texttt{ \& } u) = t' \texttt{ \& } \eta}$$

Figure 14: Type Expansion

For example, given the previous type definitions of `Empty` and `Piece`, the type expansion below dereferences `Empty` to its type expression $\{\texttt{E}\}$ and combines it with the right side of the type expression.

$$\frac{\Delta^*(\texttt{Empty}) = \{\texttt{E}\} \qquad \Delta^*(\{\texttt{X, O}\} \texttt{ \& } \{\texttt{A, B}\}) = \{\texttt{X, O}\} \texttt{ \& } \{\texttt{A, B}\} \qquad \texttt{E} \star \texttt{X}, \texttt{O}, \texttt{A}, \texttt{B}}{\Delta^*(\texttt{Empty \& } \{\texttt{X, O}\} \texttt{ \& } \{\texttt{A, B}\}) = \{\texttt{E}\} \texttt{ \& } \{\texttt{X, O}\} \texttt{ \& } \{\texttt{A, B}\}}$$

4.2.3 Type Definitions

ENV-TYPEDEF shows that a type definition d is valid if its type expression t has a valid type expansion and if its type name N has not been used before and does not clash with any of the explicit values of t. If this is the case, then Δ is extended by d and Γ is extended such that the explicit values of t have type N.

For example, the two type definitions below are valid and thus extend the typing environments.

```
type Piece        = {X, O}
type BoardContent = Piece & {Empty}
```

In the empty typing environments, the first definition results in the judgment:

$$\varnothing;\varnothing \vdash \texttt{type Piece = }\{\texttt{X, O}\} \rightsquigarrow \varnothing, \texttt{Piece} = \{\texttt{X, O}\}; \varnothing, \texttt{X} : \texttt{Piece}, \texttt{O} : \texttt{Piece}$$

That is, the definition environment is extended by a type definition for `Piece` and the term environment is extended by typings for the values `X` and `O`, which are assigned the type `Piece`. The second type definition further extends these typing environments:

$$\begin{gathered}\varnothing, \texttt{Piece} = \{\texttt{X, O}\}; \varnothing, \texttt{X} : \texttt{Piece}, \texttt{O} : \texttt{Piece} \vdash \\ \texttt{type BoardContent = Piece \& }\{\texttt{Empty}\} \rightsquigarrow \varnothing, \texttt{Piece} = \{\texttt{X, O}\}, \texttt{BoardContent} = \\ \texttt{Piece \& }\{\texttt{Empty}\}; \varnothing, \texttt{X} : \texttt{Piece}, \texttt{O} : \texttt{Piece}, \texttt{Empty} : \texttt{BoardContent}\end{gathered}$$

It should be noted that values are only assigned a single type in the term environment. The definition `BoardContent` does not change the typings of `X` or `O`.

ENV-BOARDDEF shows that a board type definition is syntactic sugar that introduces the type definitions `Content` and `Board`. The definition `Board` = `Board` is necessary, since `Board` is atomic in BoGL, that is, it cannot be expressed in terms of other types, given that there are no general collection type constructors in the language.

Env-TypeDef

$$\frac{\Delta^*(t) = t' \qquad \lfloor t' \rfloor = \bar{V}_n \qquad N, \bar{V}_n \star \Delta, \Gamma \qquad N \notin \{\bar{V}_n\}}{\Delta; \Gamma \vdash \texttt{type}\ N\ \texttt{=}\ t \leadsto \Delta, N = t; \Gamma, V_1 : N, \ldots, V_n : N}$$

Env-BoardDef

$$\frac{\Delta; \Gamma \vdash \texttt{type Content = } t \leadsto \Delta'; \Gamma'}{\Delta; \Gamma \vdash \texttt{type Board = Array (}m_x\texttt{, }m_y\texttt{) of } t \leadsto \Delta', \texttt{Board} = \texttt{Board}; \Gamma'}$$

Figure 15: Type Definitions

4.2.4 Typing Environment Initialization

The terms `True`, `False`, `not`, `or`, and `and` are always available in a BoGL program and have typings in the initial term environment. The terms `width`, `height`, `place`, `count`, and `longestRow` are added only to the term environment of programs that contain a board type definition. Similarly, `input` is added only to programs that contain an input type definition.

Figure 16 specifies how Γ is initialized with these built-ins. The first rule specifies that the empty typing environments are valid. The second rule specifies that a typing environment produced by a valid type definition is also valid. The remaining rules specify the conditions for including each of the built ins in the typing environments.

$$\frac{}{\varnothing, \varnothing \vdash \diamond} \qquad \frac{\Delta, \Gamma \vdash \diamond \qquad \Delta; \Gamma \vdash \texttt{type}\ N\ \texttt{=}\ t \leadsto \Delta'; \Gamma'}{\Delta'; \Gamma' \vdash \diamond} \qquad \frac{\Delta, \Gamma \vdash \diamond}{\Delta; \Gamma, \texttt{True} : \texttt{Bool} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond}{\Delta; \Gamma, \texttt{False} : \texttt{Bool} \vdash \diamond} \qquad \frac{\Delta, \Gamma \vdash \diamond}{\Delta; \Gamma, \texttt{not} : \texttt{(Bool -> Bool)} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond}{\Delta; \Gamma, \texttt{or} : \texttt{(Bool, Bool) -> Bool} \vdash \diamond} \qquad \frac{\Delta, \Gamma \vdash \diamond}{\Delta; \Gamma, \texttt{and} : \texttt{(Bool, Bool) -> Bool} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond \qquad \texttt{Board} \in dom(\Delta)}{\Delta; \Gamma, \texttt{width} : \texttt{Int} \vdash \diamond} \qquad \frac{\Delta, \Gamma \vdash \diamond \qquad \texttt{Board} \in dom(\Delta)}{\Delta; \Gamma, \texttt{height} : \texttt{Int} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond \qquad \texttt{Board} \in dom(\Delta)}{\Delta; \Gamma, \texttt{place} : \texttt{(Content, Board, (Int, Int)) -> Board} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond \qquad \texttt{Board} \in dom(\Delta)}{\Delta; \Gamma, \texttt{count} : \texttt{(Content, Board) -> Int} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond \qquad \texttt{Board} \in dom(\Delta)}{\Delta; \Gamma, \texttt{longestRow} : \texttt{(Content, Board) -> Int} \vdash \diamond}$$

$$\frac{\Delta, \Gamma \vdash \diamond \qquad \texttt{Input} \in dom(\Delta)}{\Delta; \Gamma, \texttt{input} : \texttt{Input} \vdash \diamond}$$

Figure 16: Typing Environment Initialization

4.2.5 Type Signatures and Equations

Figure 17 shows that a type signature extends the term environment with a typing if the name that it introduces is fresh and if the type that it assigns to that name is valid.

The equation rules in Figure 18 check that the name and type of an equation match its type signature. Env-Feq says that parameters are assigned index-wise types based on the function input in the type signature. The body expression must

$$\frac{x \star \Gamma \qquad \Delta \vdash b}{\Delta;\Gamma \vdash x\ :\ b \leadsto \Delta;\Gamma,\ x : b}\ \textsc{Sig-Val} \qquad \frac{f \star \Gamma \qquad \Delta \vdash b \qquad \Delta \vdash b'}{\Delta;\Gamma \vdash f\ :\ b\ \texttt{->}\ b' \leadsto \Delta;\Gamma,\ f : b\ \texttt{->}\ b'}\ \textsc{Sig-Fun}$$

Figure 17: Type Signature Rules

match the output type of the type signature in its lexical environment. Env-Veq says that, if the body expression of a value equation has the type declared by its signature, then that value equation is valid. Env-Beq is a rule for checking board equations. It must be used to check all of the board equations in a board definition individually. It says that a board equation is valid if it places expressions of type `Content` on a board that exists in the term environment.

$$\frac{f : (b_1, \ldots, b_k)\ \texttt{->}\ b \in \Gamma \qquad \Delta;\Gamma,\ x_1 : b_1, \ldots, x_k : b_k \vdash e : b}{\Delta;\Gamma \vdash f\texttt{(}x_1\texttt{,}\ \ldots\texttt{,}\ x_k\texttt{)}\ \texttt{=}\ e}\ \textsc{Env-Feq}$$

$$\frac{x : b \in \Gamma \qquad \Delta;\Gamma \vdash e : b}{\Delta;\Gamma \vdash x\ \texttt{=}\ e}\ \textsc{Env-Veq} \qquad \frac{x : \texttt{Board} \in \Gamma \qquad \Delta;\Gamma \vdash e : \texttt{Content}}{\Delta;\Gamma \vdash x\texttt{!(}p_x, p_y\texttt{)}\ \texttt{=}\ e}\ \textsc{Env-Beq}$$

Figure 18: Equation Rules

4.2.6 Subtyping

Subtyping allows a type system to accept a larger set of well-behaved programs through the principle of safe substitution. That is, if $t_1 <: t_2$, it is safe to use an expression of type t_1 where an expression of type t_2 is expected [8]. Its purpose in BoGL is to allow the components of an extended type to be admitted where the extended type is expected in function application and for type unification in conditional expressions and equality comparison. It is used directly only by rule T-Subsumption in Figure 23, where types must be expanded before being used in the subtype relation.

BoGL subtyping has a subset semantics; if the set of types in t_1 is a subset of the set of types in t_2, then $t_1 <: t_2$. This is made precise by the rules in Figure 19. The subtyping relation is reflexive (by S-Refl) and transitive (by S-Trans). Tuple subtyping is defined in an element-wise manner. S-And allows extended types to be subtypes of other extended types, importantly allowing permutations to be compared. S-Left and S-Right allow components of a type expressions to be subtypes of that type expression.

Subtyping can be formulated with a bottom type, that is, one that is the subtype of all other types and/or with a top type, that is, one that is the supertype of all other types. BoGL does not include either of these, since they are not necessary for the formalization nor for writing programs. Additionally, subtyping in functional languages is typically defined for functions, but this is not done in BoGL since they are not first class values.

Figure 20 contains an example program excerpt that requires subtyping. It defines `TR`, which is a subtype of `T`, but is not defined in terms of it and is defined such that a component-wise comparison will not suffice to derive the subtype relationship between the two. Figures 21 and 22 show a partial derivation that assigns the type `T` to `f(x)` in the definition of `h` by subsumption so that the program excerpt is well-typed. Rule names are abbreviated in Figure 21.

S-Refl
$$t <: t$$

S-Trans
$$\frac{t <: t_1 \qquad t_1 <: t_2}{t <: t_2}$$

S-And
$$\frac{t_1 <: t \qquad t_2 <: t}{t_1 \texttt{ \& } t_2 <: t}$$

S-Left
$$t_L <: t_L \texttt{ \& } t_R$$

S-Right
$$t_R <: t_L \texttt{ \& } t_R$$

S-Tuple
$$\frac{t_1 <: t'_1 \qquad \ldots \qquad t_n <: t'_n}{(t_1, \ldots, t_n) <: (t'_1, \ldots, t'_n)}$$

Figure 19: The Subtype Relation

```
type TA = {A}
type TB = {B}
type TC = {C}
type TD = {D}

type TAB = TA & TB
type TABC = TAB & TC

type TCD = TC & TD
type TCDE = TCD & {E}

type T = TAB & TCDE
type TR = TD & TABC

f : Int -> TR
f(x) = ...

g : T -> Int
g(x) = ...

-- requires that TR <: T
h : Int -> Int
h(x) = g(f(x))
```

Figure 20: A program excerpt that requires subtyping

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{\{D\}} <: \texttt{\{D\}}}\text{S-R}_{\text{F}}}{\texttt{\{D\}} <: \texttt{\{D\} \& \{E\}}}\text{S-L}}{\texttt{\{D\}} <: \texttt{\{A\} \& \{B\} \& \{C\} \& \{D\} \& \{E\}}}\text{S-R} \quad \dfrac{\dfrac{}{\texttt{\{A\} \& \{B\} \& \{C\}} <: \texttt{\{A\} \& \{B\} \& \{C\}}}\text{S-R}_{\text{F}}}{\texttt{\{A\} \& \{B\} \& \{C\}} <: \texttt{\{A\} \& \{B\} \& \{C\} \& \{D\} \& \{E\}}}\text{S-L}}{\texttt{\{D\} \& \{A\} \& \{B\} \& \{C\}} <: \texttt{\{A\} \& \{B\} \& \{C\} \& \{D\} \& \{E\}}}\text{S-A}$$

Figure 21

$$\dfrac{\dfrac{\cdots}{\Delta;\Gamma \vdash \texttt{f(x)} : \texttt{TR}} \quad \dfrac{\cdots}{\Delta^*(\texttt{TR}) = \texttt{\{D\} \& \{A\} \& \{B\} \& \{C\}}} \quad \dfrac{\cdots}{\Delta^*(\texttt{T}) = \texttt{\{A\} \& \{B\} \& \{C\} \& \{D\} \& \{E\}}} \quad \text{Figure 21}}{\Delta;\Gamma \vdash \texttt{f(x)} : \texttt{T}}\text{T-Subsumption}$$

Figure 22

4.2.7 Expressions

Figure 23 contains the expression typing rules. It should be noted that expressions can only be assigned a base type, that is, either a type name or a tuple of type names.

For example, given the now-familiar type definition for `Piece` and the expression `let x = X in foo(x)`, `x` has type `Piece`, not {`X, O`}. This requirement simplifies the typing rules and enforces that the purpose of type expressions is to define types and to allow structural comparison, whereas base types provide a way of referring to types. Structural comparison is needed for subtyping, so types are expanded before being used with this relation.

T-Int, T-Tuple, T-Var, T-App, and T-Let are the usual rules for these features.

T-Symbol specifies that, if a symbolic value has type information in Γ, then it is assigned that type. The rule assigns at most one type to a symbolic value. If a symbolic value is part of multiple type definitions, T-Subsumption can be used to assign it one of those types.

T-Subsumption specifies that, if an expression has some type, it also has all of the supertypes of that type. This rule is needed to admit some $t <: t'$ in a context where t' is required. Since subtyping is formulated in terms of expanded type expressions, Δ^* is used directly in the premises.

T-If specifies that, if the conditional of an if expression has type `Bool` and the then and else branches have the same type b, then the if expression also has type b. It should be noted that b may be a supertype of the types of the then and else expressions. This is discussed in more detail in T-Eq.

T-While specifies that the condition expression of a while loop must have type `Bool` and that the body expression must have the same type as its let-bound loop state. The restriction on the body expression ensures that the loop state can be properly updated and that the false case of a while loop returns the same type as any subsequent iterations of that loop. There is no case to handle a while loop that is not enclosed in a let expression, which is the result of a loop written in an empty binding context. The lack of this case means that it is not possible to derive a typing for a program with such a loop; in other words, it is a type error.

T-Arithmetic specifies that arithmetic operations are well-typed if they are used with expressions of type `Int`. If it is desired for language usability purposes, this rule could be reformulated to support arithmetic operations between expressions of the same potentially nested integer tuple type, such as `((1,1),(2,2) + ((1,1),(2,2)) = ((2,2),(4,4))`. In this case, the semantics would need to support position-wise arithmetic operations for tuples.

T-Get specifies that board access expression has type `Content` if its left side has type `Board` and its right side has type `(Int, Int)`. Since board dimensions are known statically, limited bounds checking may also be performed for board access.[15]

T-Eq specifies that (in)equality comparison is only allowed for expressions that have the same type. It should be noted that, due to T-Subsumption, if there exists a supertype of two distinct types, then they can be compared. One implication of this is that an ill-typed equality expression may be well typed if a suitable type definition is provided. For example, `1 == A` is ill-typed in a program that only includes the type definition `type T = {A}`, but well typed in a program that instead has `type T = Int & {A}`. If this property proves to be counterproductive to the educational goals of BoGL, the final two premises may instead be $\Delta; \Gamma \vdash e_1 : b_1$ and $\Delta; \Gamma \vdash e_2 : b_2$. That is, the two expressions need not be required to have the same type, so long as the semantics is formulated to accommodate this.

T-Compare specifies that relative comparison between expressions is well-typed if they have type T-Int. As with T-Arithmetic, this rule could be expanded to support potentially nested integer tuple types.

T-Project specifies that, if all of the indices are within the bounds of the input tuple of a projection, then its type is a tuple of the types of the respective input tuple elements. It should be noted that a projection may produce a non-tuple type in the case of a single index, and it may also produce an output tuple that is of higher cardinality than the original. However, projection of an expression with a non-tuple

[15] The implementation checks literal expressions.

type is not type correct.

$$\textsc{T-Int}\quad \frac{}{\Delta;\Gamma \vdash m : \texttt{Int}} \qquad \textsc{T-Symbol}\quad \frac{V : N \in \Gamma}{\Delta;\Gamma \vdash V : N} \qquad \textsc{T-Tuple}\quad \frac{\Delta;\Gamma \vdash e_1 : b_1 \;\ldots\; \Delta;\Gamma \vdash e_k : b_k}{\Delta;\Gamma \vdash \texttt{(}e_1, \ldots, e_k\texttt{)} : \texttt{(}b_1, \ldots, b_k\texttt{)}}$$

$$\textsc{T-Subsumption}\quad \frac{\Delta;\Gamma \vdash e : b \qquad \Delta^*(b) = t \qquad \Delta^*(b') = t' \qquad t <: t'}{\Delta;\Gamma \vdash e : b'}$$

$$\textsc{T-App}\quad \frac{f : b_1 \texttt{ -> } b_2 \in \Gamma \qquad \Delta;\Gamma \vdash e : b_1}{\Delta;\Gamma \vdash f\texttt{(}e\texttt{)} : b_2} \qquad \textsc{T-Var}\quad \frac{x : b \in \Gamma}{\Delta;\Gamma \vdash x : b}$$

$$\textsc{T-If}\quad \frac{\Delta;\Gamma \vdash e_1 : \texttt{Bool} \qquad \Delta;\Gamma \vdash e_2 : b \qquad \Delta;\Gamma \vdash e_3 : b}{\Delta;\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : b}$$

$$\textsc{T-Let}\quad \frac{\Delta;\Gamma \vdash \texttt{(}e_1, \ldots, e_n\texttt{)} : \texttt{(}b_1, \ldots, b_n\texttt{)} \qquad \Delta;\Gamma,\; x_1 : b_1, \ldots, x_n : b_n \vdash e : b}{\Delta;\Gamma \vdash \texttt{let (}x_1, \ldots, x_n\texttt{) = (}e_1, \ldots, e_n\texttt{) in } e : b}$$

$$\textsc{T-While}\quad \frac{\begin{array}{c}\Delta;\Gamma \vdash \texttt{(}e_1, \ldots, e_n\texttt{)} : \texttt{(}b_1, \ldots, b_n\texttt{)} \\ \Delta;\Gamma,\; x_1 : b_1, \ldots, x_n : b_n \vdash e_c : \texttt{Bool} \qquad \Delta;\Gamma,\; x_1 : b_1, \ldots, x_n : b_n \vdash e_b : \texttt{(}b_1, \ldots, b_n\texttt{)}\end{array}}{\Delta;\Gamma \vdash \texttt{let (}x_1, \ldots, x_n\texttt{) = (}e_1, \ldots, e_n\texttt{) in while } e_c \texttt{ do } e_b : \texttt{(}b_1, \ldots, b_n\texttt{)}}$$

$$\textsc{T-Arithmetic}\quad \frac{o \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\} \qquad \Delta;\Gamma \vdash e_1 : \texttt{Int} \qquad \Delta;\Gamma \vdash e_2 : \texttt{Int}}{\Delta;\Gamma \vdash e_1 \; o \; e_2 : \texttt{Int}}$$

$$\textsc{T-Get}\quad \frac{\Delta;\Gamma \vdash e_1 : \texttt{Board} \qquad \Delta;\Gamma \vdash e_2 : \texttt{(Int,Int)}}{\Delta;\Gamma \vdash e_1 \texttt{ ! } e_2 : \texttt{Content}}$$

$$\textsc{T-Eq}\quad \frac{o \in \{\texttt{==}, \texttt{/=}\} \qquad \Delta;\Gamma \vdash e_1 : b \qquad \Delta;\Gamma \vdash e_2 : b}{\Delta;\Gamma \vdash e_1 \; o \; e_2 : \texttt{Bool}}$$

$$\textsc{T-Compare}\quad \frac{o \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\} \qquad \Delta;\Gamma \vdash e_1 : \texttt{Int} \qquad \Delta;\Gamma \vdash e_2 : \texttt{Int}}{\Delta;\Gamma \vdash e_1 \; o \; e_2 : \texttt{Bool}}$$

$$\textsc{T-Project}\quad \frac{\Delta;\Gamma \vdash e_1 : b_1 \qquad \ldots \qquad \Delta;\Gamma \vdash e_n : b_n \qquad \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}}{\Delta;\Gamma \vdash \texttt{(}e_1, \ldots, e_n\texttt{) \# (}i_1, \ldots, i_k\texttt{)} : \texttt{(}b_{i_1}, \ldots, b_{i_k}\texttt{)}}$$

Figure 23: Expression Typing Rules

4.3 Value Extraction

A current problem with the BoGL language is that some values cannot be efficiently extracted from extended types. Although this has not proven problematic for the programs that users are currently writing, it is nonetheless an issue that should be resolved. Two approaches that are in consideration are briefly sketched out. One is a refinement approach that the Hack language [12] and Flow JavaScript static type checker [13] use. This would require introducing typing rules that track values excluded by the condition of an if expression so that extended variable types can be reduced to their components in Γ. For example, `Empty` can be removed from the potential values of `x` and its type can be reduced to `Int` in the else branch below.

```
type T     = {Empty}
type Board = Int & T

increment : Content -> Content
increment(x) = if x == Empty then Empty else x + 1
```

This has the advantage of not introducing new syntax, although it complicates the formalization and implementation of the language. It is also syntactically unwieldy to reduce to a base type if it is extended by many symbolic values. This bloat can be alleviated with an operator that is syntactic sugar that tests for equality with all the values of an enumerated type.

```
type T     = {A, B, Empty}
type Board = Int & T

increment : Content -> Content
increment(x) = if x is T then x else x + 1
```

However, this may pose problems for students since it may not be clear why this can only be done with enumerated type names. Another approach is to introduce a typecase construct [1]. This has the advantages of keeping if expressions simple, of being more general, and of not introducing syntactic bloat, although it comes at the

cost of a significant extension of the language. An example is given below:

```
increment' : Content -> Content
increment'(x) = typecase x of
                    (x : Int) -> x + 1
                    (x : T)   -> x
```

4.4 Type Checking

A minimal type checker for BoGL takes as input a (program, input buffer, input expression) triple and returns either a *btype* or a type error, hereafter referred to as a *result type*. An implementation may produce a more sophisticated output, such as informative messages, position information, and a curated set of type errors.

The type checking process is shown in Figure 24. The first phase verifies that the program is well typed and initializes the typing environments. That is, it checks that all type definitions are valid and adds them to Δ, introduces typings in Γ for built-ins according to these type definitions, and further extends Γ with typings from signatures if they all correspond to their equations. It also checks that all elements of the input buffer have type `Input`. If this succeeds, then the input expression is type checked in the typing environments produced by the first phase. A type error occurs if either the first phase fails or if the input expression is ill typed.

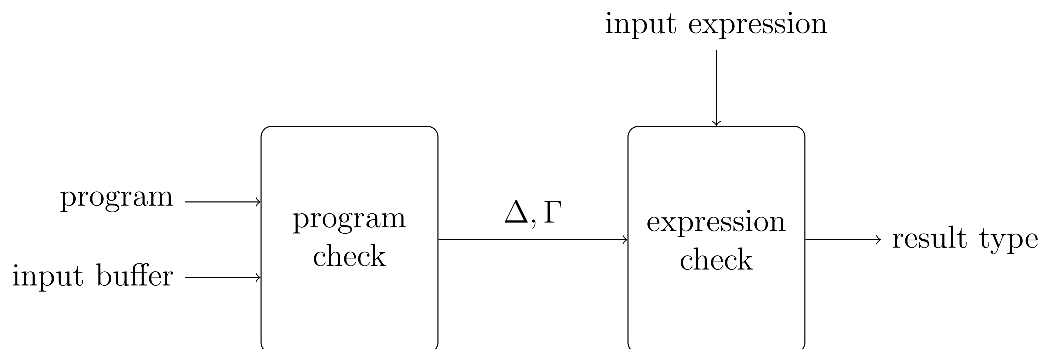

Figure 24: Type Checker Operation

5 Related Work

Other functional languages have imperative-style iteration constructs. For example, Scheme has the do and named let constructs [9]. The former is similar in function to BoGL while loops, except that execution terminates on a true condition and variables must be explicitly provided to each loop expression rather than via binding context. The latter is similar in form in that it provides an iteration mechanism through a let expression, however, this construct implicitly creates a function that must be recursively called within the loop body. Additionally, the Guile implementation of Scheme includes a while construct [4]. This is differs from BoGL's loops in two ways: it has the additional continue and break control flow functions and it either returns the value of its boolean condition or the arguments provided to the break function. Although Haskell does not have a language-level feature that is similar, it does have the loops [11] and monad-loops libraries [3]. These both provide iteration constructs in a purely functional language with monadic variants that abstract away the process of manually threading state.

Igarashi and Nagira propose union types as a mechanism for subtyping in object-oriented languages that is not based on inheritance so that independently-developed types can be grouped together, which is the motivation behind extended types in BoGL [6]. Their work goes beyond the simplicity of BoGL's extended types to factor out common fields and methods of classes to support type-safe member access and exhaustive case analysis.

6 Conclusion

This thesis continues the formalization of the BoGL educational domain-specific programming language. It presents functional while loops that use a static binding context to simulate local, mutable state with immutable variables, which can serve as a gentle introduction to writing recursive functions. It also presents a type system

with subtypable extended and enumeration types that form a conceptually lightweight mechanism for abstraction and representation. The work presented herein contributes to an ongoing effort to produce a language standard, which will enable scientific communication about the language, support educators in their effort to learn about and develop BoGL teaching material, and facilitate work on the current implementation as well as allow for other implementations.

There are other static analyses that can be further developed and formalized, such as limited bounds checking for boards and coverage checking for board definitions. Similarly, the BoGL semantics can also be formalized to support language standardization. Additionally, the problem of type extraction mentioned in Section 4.3 can be resolved with one of the proposed solutions, though it should take into consideration the primary concern of language simplicty and teachability. Finally, with a formally specified type system, additional work can be done to improve type error messages.

7 Appendix

7.1 BoGL Grammar

```
     game ::= game Name typedef* valuedef*                      (game definition)
  typedef ::= type Type = type                                  (general type definition)
            | type Board = Array (int,int) of type              (board type definition)

    btype ::= Type | (btype, ... , btype)                       (base type)
    etype ::= {Value, ... , Value}                              (enumeration type)
     type ::= btype | etype | type & Type | type & etype        (type expression)
    ftype ::= btype -> btype                                    (function type)
    vtype ::= btype | ftype                                     (value type)

 valuedef ::= signature equation                                (value definition)
signature ::= name : vtype                                      (type signature)
 equation ::= name = expr                                       (value equation)
            | name(name, ... , name) = expr                     (function equation)
            | boardeq ... boardeq                               (board equations)
  boardeq ::= name!(pos,pos) = expr                             (board range definition)
      pos ::= int | name                                        (board position)

    aexpr ::= int | Value | name                                (value, name)
     expr ::= aexpr                                             (atomic expression)
            | (expr, ... , expr)                                (tuple)
            | name(expr, ... , expr)                            (function application)
            | expr binop expr                                   (infix application)
            | expr # int                                        (single tuple projection)
            | expr#(int, ... ,int)                              (tuple projection)
            | let name = expr in expr                           (local definition)
            | let (name, ... , name) = (expr, ... , expr) in expr   (local definitions)
            | if expr then expr else expr                       (conditional)
            | while expr do expr                                (while loop)
    binop ::= + | - | * | / | ! | == | /= | > | >= | < | <=     (binary operation)
```

Figure 25: BoGL Syntax

7.2 Listing of Built-In Functions and Values

```
input      : Input
width      : Int
height     : Int
True       : Bool
False      : Bool
```

Figure 26: Built-In Values

```
place      : (Content,Board,(Int,Int)) -> Board
count      : (Content,Board) -> Int
longestRow : (Content,Board) -> Int
not        :  Bool -> Bool
or         : (Bool,Bool) -> Bool
and        : (Bool,Bool) -> Bool
```

Figure 27: Built-In Functions

7.3 A Complete Example Program

```
game TicTacToe

{-
   A game of tic-tac-toe written in the BoGL language.
   Play by typing the word play into the REPL.
   Then enter integer pairs as inputs, for example (1, 1).
-}

-- declares the symbolic values X and O in an enumerated type
type Player = {X, O}

-- declares a 1-indexed width x height board
-- (1, 1) is the top left and (3, 3) is the bottom right
-- constrained to contain only X, O, or Empty values
-- & creates an extended type
```

```
type Board = Array (3, 3) of Player & {Empty}

-- the value returned at the end of a game
type Result = {XWins, OWins, Tie}

-- represents the current player and the pieces on the board
type State = (Player, Board)

-- a type synonym for a pair of integers
type Position = (Int, Int)

-- the type system will reject input type other than (Int, Int)
-- ex: (1, 1) will be accepted
-- only literal values are allowed
-- ex: expressions such as let a = 1 in (a, a) are rejected
type Input = Position

-- initialize all pieces on the board to Empty
initialBoard : Board
initialBoard!(x, y) = Empty

-- allows alternation of turns
next : Player -> Player
next(p) = if p == X then O else X

-- Comment out the definitions of inARow and isFull
-- to run this game in the current implementation

-- Checks if x or more pieces of c are in a row
inARow : (Int, Content, Board) -> Bool
inARow(x, c, b) = longestRow(c, b) >= x

-- Checks that a board does not contain Empty pieces
isFull : Board -> Bool
```

```
isFull(b) = count(Empty, b) = 0

-- determines the outcome of a game of tic-tac-toe
-- precondition: the game is over
outcome : (Player, Board) -> Result
outcome(p, b) = if inARow(3, X, b) then XWins else
                if inARow(3, O, b) then OWins else Tie

-- uses the built-in inARow function
-- determines if either player has 3 pieces in a row
threeInARow : Board -> Bool
threeInARow(b) = or(inARow(3, X, b), inARow(3, O, b))

-- determines if the game is over
gameOver : Board -> Bool
gameOver(b) = or(threeInARow(b), isFull(b))

-- checks whether a piece can be placed at a board position
isValid : (Board, Position) -> Bool
isValid(b, p) = if b ! p == Empty then True else False

-- prompts user for input until a a valid one is provided
-- changes the state by placing a piece and switching the player
tryMove : (Player, Board) -> (Player, Board)
tryMove(p, b) = let pos = input in
                if isValid(b, pos) then (next(p), place(p, b, pos))
                                   else (p, b)

-- runs a game until it ends
gameLoop : (Player, Board) -> (Player, Board)
gameLoop(p,b) = while not(gameOver(b)) do tryMove(p, b)

-- plays a game of tic-tac-toe given an initial state
-- and determines the result after it ends
```

```
playWith : (Player, Board) -> Result
playWith(a, b) = outcome(gameLoop(a, b))

-- starts a game of tic-tac-toe with the standard setup
-- returns the result of playing the game
play : Result
play = playWith(X, initialBoard)
```

References

[1] Martín Abadi et al. "Dynamic typing in a statically typed language". eng. In: *ACM Transactions on Programming Languages and Systems* 13.2 (1991), pp. 237–268.

[2] Luca Cardelli. "Type Systems". In: *CRC Handbook of Computer Science and Engineering*. Ed. by Allen Tucker. 2nd ed. CRC Press, 2004. Chap. 97.

[3] James Cook. *monad-loops: Monadic loops*. 2015. URL: https://hackage.haskell.org/package/monad-loops.

[4] The Guile Developers. *GNU Guile 3.0.5 Reference Manual*. 2021. URL: https://www.gnu.org/software/guile/manual/.

[5] Martin Erwig. *BoGL Syntax*. 2020. URL: https://bogl.engr.oregonstate.edu/docs/BoglSyntax.pdf.

[6] Atsushi Igarashi and Hideshi Nagira. "Union Types for Object-Oriented Programming". In: *Journal of Object Technology* 6.2 (2007), p. 47.

[7] Simon Marlow et al. *Haskell 2010 language report*. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/.

[8] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[9] Alex Shinn, John Cowan, and Arthur Glecker A., eds. *Revised7 Report on the Algorithmic Language Scheme*. 2013. URL: https://small.r7rs.org/attachment/r7rs.pdf.

[10] Jeremy Siek. *Crash Course on Notation in Programming Language Theory*. 2012. URL: http://siek.blogspot.com/2012/07/crash-course-on-notation-in-programming.html.

[11] Thomas Tuegel. *loops: Fast imperative-style loops*. 2014. URL: https://hackage.haskell.org/package/loops.

[12] *Types: Type Refinement*. URL: https://docs.hhvm.com/hack/types/type-refinement.

[13] *Union Types*. URL: https://flow.org/en/docs/types/unions/#toc-unions-refinements.