Implementing a Logical Type System in the Villain Compiler Framework

William Chung
University of Maryland
College Park, Maryland, USA

1 Introduction

Type systems in programming languages help programmers avoid mistakes, reduce run-time errors, force documentation, and reduce type checking during run-time. Developing and implementing a type system for existing untyped scripting languages provides a way for programmers of these languages an easy way to access these benefits. A good model for such an implementation is Typed Racket, a sister language for Racket that allows and enforces statically-checked type annotations. Typed Racket's main, distinguishing feature is occurrence typing, a type discipline that uses common reasoning methods to assign variable occurrences a type based on predicates prior to the occurrence.

The rest of the paper will talk about three extension languages that I designed that act as a bridge to get the compiler for Villain and the dynamically typed languages mentioned in the CMSC838E notes that it extended off of closer to implementing a language similar to Typed Racket. The three extensions are Typed Alpha which implements static type checking, Typed Bumble which implements union types and typed annotated one-parameter functions, and Typed Control which implements occurrence typing. Each section explaining the three extensions provides the type system's syntax, typing rules, implementation, and examples to evaluate it. In addition to talking about how the implementations are tested for correctness, the paper will talk about how the languages intuitively compare with counterparts, what the limitations are, and what could be added in the future. All code for each implementation for the three programming languages is in its respective folder in the GitHub repository attached to this paper.

2 Related Work

Three main sources will be used for the implementation of these extensions.

CMSC8383E Notes: These lecture notes for CMMSC838E [1] include much source code that will be useful as the base for the extensions. Specifically, Con will be used as the dynamic counterpart of Typed Alpha since it includes integers, Booleans, conditionals, and predicates. The code from the addition of function definitions and calls in Iniquity is included in the dynamic counterpart of Typed Bumble. The function definitions and calls in Iniquity allow multiple arguments for function definitions and calls while Typed Bumble will guarantee only one.

Types and Programming Languages: Types and Programming languages [2] is a book published on type systems. Chapter 8 of the book showcases a typing system that includes natural numbers, Booleans, conditionals, and predicates. Due to the Con language mentioned earlier having all these features, this type system would be the perfect type system to base the starting extension language, Typed Alpha, on. In addition, chapter 9 showcases a simply-typed lambda-calculus version of this language which inspires some of the rules in Typed Bumble. Chapter 10 showcases an ML implementation of the type checker of this language that will act as the basis of the "typecheck.rkt" file for Typed Alpha and Typed Bumble.

Logical Types for Untyped Programming Languages: This paper [3] reformulates occurrence typing by discussing a proof system that helps determines types of variable occurrences from the environment of propositional logical formula called λ_{TR} which Typed Control will be a simplified version of. This paper also helped inspire the format of the rest of the paper in that it talks about syntax, typing rules, and works through an example for its language. In addition, there are many interesting facts the paper mentions about occurrence typing and the implementation that helped the implementation of Typed Control. For instance, section 7.4 mentions how simplifying logical formulas and refining the type environment each time a new propositional formula is added reduces to cost of the proof system on the implementation. The implementation of Typed Control uses this tip.

3 Typed Alpha

The first implementation of a type system is Typed Alpha. The type system is based on the type system laid out in chapter 8 of Types of Programming Languages. The implementation of the type checker in the file "typecheck.rkt" is loosely based on the ML implementation of Types of Programming Languages with the other files such as "parse.rkt", "compile.rkt", and "ast.rkt" based on Con in the lecture notes of CMSC838E. All these files are in the "TypedAlpha" folder of the GitHub repository. As the starting point for the languages, the language includes the basic types of Numbers and Booleans with basic predicates and if expressions. Numbers will only be integers in this programming language. In the upcoming sections, I will explain the formalities of the type system, the implementation, and the examples.

$$
\begin{array}{lr}
e ::= (\text{if } e\ e\ e) \mid (\text{c } e) \mid \#\text{t} \mid \#\text{f} \mid n & \text{Expressions} \\
c ::= add1 \mid sub1 \mid zero? & \text{Primitive Operations} \\
\tau_f ::= \mathbf{N} \mid \mathbf{B} & \text{First-Class Types} \\
\tau ::= \tau_f \mid \tau_f \rightarrow \tau_f & \text{Types}
\end{array}
$$

Figure 1. Typed Alpha Syntax

3.1 Syntax

The syntax, being mainly based on the system in Types of Programming Languages, of basic expression and type syntax and is described in Figure 1. It consists of numeric and Boolean constants, conditions, and basic primitive. "ast.rkt" loosely presents the syntax.

For **types**, **N** describes the type of numeric, integer values. **B** describes the type of the Boolean constants of true (#t) and false (#f). τ_f represents first-class types that can be used as inputs and output types for functions. $\tau_f \rightarrow \tau_f$ describes the type of functions that are not first-class, and will only be the type of primitive operators. The reason functions are not first-class is to make it easier for the implementation as the implementation mainly follows Con in the lecture notes which does not implement first-class functions.

3.2 Typing Rules

$$\frac{}{\vdash n : \mathbf{N}}(\textit{T-NUM})$$

$$\frac{}{\vdash \#t : \mathbf{B}}(\textit{T-TRUE})$$

$$\frac{}{\vdash \#f : \mathbf{B}}(\textit{T-FALSE})$$

$$\frac{}{\vdash c : \delta_\tau(c)}(\textit{T-CONST})$$

$$\frac{\vdash c : \tau \rightarrow \tau' \qquad \vdash e : \tau}{\vdash (c\ e) : \tau'}(\textit{T-PRIM})$$

$$\frac{\vdash e_1 : \mathbf{B} \qquad \vdash e_2 : \tau \qquad \vdash e_3 : \tau}{\vdash (if\ e_1\ e_2\ e_3) : \tau}(\textit{T-IF})$$

Figure 2. Typed Alpha Typing Rules

This subsection will elaborate off of the typing rules in Figure 2.

Constants: The simplest rule is T-NUM which states that a numeric constant n is of type **N**. The next two simple rules are T-TRUE and T-FALSE. T-TRUE states that the Boolean constant true is of type **B**. T-FALSE states that the Boolean constant false is of type **B**. For the primitive constants, using T-CONST, the type is determined by a $\delta_\tau(c)$ function that assigns types to these constants. As shown in figure 10, $\delta_\tau(add1)$ and $\delta_\tau(sub1)$ both equate type $\mathbf{N} \rightarrow \mathbf{N}$. $\delta_\tau(zero?)$ equates to $\mathbf{N} \rightarrow \mathbf{B}$. Other primitives not in the syntax of Typed Alpha, but in other languages are also listed in the figure and will be used of future results.

Primitive Expressions: In chapter 8 of Types of Programming Languages, the system uses three rules to represent primitive operations of T-SUCC, T-PRED, T-ISZERO. Instead, I made it into a single rule T-PRIM that takes in (c *e*). The typing rule determines the type of the primitive, *c*, which has to be of a function type, which would have to use T-CONST since that is the only typing rule that returns a functional type τ -> τ'. After that, *e* would need to be the same type of τ. T-CONST would then state if *c* is of type τ -> τ', and *e* is of type τ, the expression (c *e*) would be of type τ'.

Conditionals: The T-IF rule states that given the syntax (if e_1 e_2 e_3) , if e_1 is of type **B** and e_2 and e_3 are both of type τ, then the expression (if e_1 e_2 e_3) is of type τ. Many programming languages have different typing rules for if expressions for the typing of the predicate and whether the branch statements have to be the same. Racket lets both the predicates and branches for any type. However, chapter 8 in Types of Programming Languages uses these rules, so I will follow this restriction for this basic programming language. In the upcoming extensions, with the addition of the union types, I will make the typing rules for conditionals more similar to Racket's implementation.

3.3 Implementation

Many files were taken from Con and have remained mostly unchanged. The biggest change to "ast.rkt" was the addition of type structures of Integer to represent **N** and Boolean to represent **B**. "compile.rkt" was also changed as before compiling the expression, the type checker gets called to return the type of the expression. If the type checker returns an error, the compiler throws a Type Mismatch error saying what was expected and what it got. Since the type checker runs before assembly code generation, the type checker saves the compiler from compiling a badly typed expression which removes type-related run time exceptions, which would be all the exceptions.

The main file added was "typecheck.rkt" which implements the type checker. This is loosely based on the type

checker in Chapter 10 of Types and Programming Language. The function "get-type" gets called by the compiler in "compile.rkt" before code generation. The function uses pattern matching and some recursive functions that also calls *get-type* to determine what typing rule to use for e.

Constants: If e is patterns match to an Integer, it gets type **N** represented by the Integer structure which represents T-NUM. If e pattern matches a Boolean, it gets type **B** represented by the Boolean structure which represents T-TRUE and T-FALSE. These are simple cases that do not need function calls.

Primitive Operations: If e is matched to the primitive expression (c e), *get-type-prim*1 is called on c and e which simulates T-PRIM. First, get-type-op1 is called on c, which simulates T-CONST as the results from δ_τ are coded in, which gets c's type *in-ty*. *Get-type* is called on e to get e's type. If e's type is the same as *in-ty*, then this a well-typed program. If not, the type checker throws an error and print "type mismatch" and tells the user the type was expected and what it was given. The printing of types is implemented by print-type and the checking of type equality is implemented by *type-eq*?.

Conditionals: If e patterned matches the syntax (if $e1$ $e2$ $e3$), *get-type-if* is called on $e1$, $e2$, and $e3$ which simulates T-IF. First, the function checks if the type of $e1$ matches with the type Boolean with a get-type call and pattern matching. If it is not, it throws a type mismatch error, an error explained early. Then, it checks if the type of $e2$ matches with the type of $e3$. If it does, it returns their common type, and if it does not, a type mismatch error gets thrown.

3.4 Examples and Evaluation

We will look at an example of an expression that will pass the type checker and successfully compile. Then, we will look at an expression that will fail the type checker and throw a type mismatch error. The expression we will look at first is:

(if #f (sub1 7) 4)

To get the type of this expression first, we would first use T-IF. Using T-IF, we would use T-FALSE to get the type of #f which is **B**. After validating the predicate of #f is **B**, we would use T-PRIM to get the expression of (sub1 7). We would then T-CONST to get the type of sub1 which is $\mathbf{N} \rightarrow \mathbf{N}$. Then, we would use T-NUM to get the type of 7 which is **N**. Since sub1 is $\mathbf{N} \rightarrow \mathbf{N}$ and 7 is N, we can use T-PRIM to show that (sub1 7) is **N**. Then, we would use T-NUM to get the type of 4 which is **N**. Because the predicate #f is **B** and (sub1 7) and 4 are both **N**, this expression's type is of type **N** using T-IF meaning the expression is well-typed and would pass the type checker. The type checker goes through this algorithm with its functions that represent the type systems that were mentioned earlier and from the tester function "compile.rkt" in the test folder that uses "test-runner.rkt", the compiler does return the correct result of 4 without any type mismatch errors.

An example of an expression that is not well-typed is

(add1 (zero? 7))

To try to see if we could get the type of this (add1 (zero? 7)) Since T-PRIM is the only typing rule that handles primitives, we would need to see if we can use T-PRIM. We would then use T-CONST to get the type of add1 which $\mathbf{N} \rightarrow \mathbf{N}$. Then, we would use T-PRIM to get the type of (add1 t). Using T-CONST, we would know that zero? is $\mathbf{N} \rightarrow \mathbf{B}$. We would use T-NUM to determine 7 is **N**. Since 7 is **N** and zero? is $\mathbf{N} \rightarrow \mathbf{B}$, using T-PRIM, we can determine (zero? 7) is of type **B**. Since, add1 is $\mathbf{N} \rightarrow \mathbf{N}$ and (zero? 7) is **B** and not **N**, we can determine that we can not use T-PRIM to get the type of (add1 (zero? 7)) meaning there is no rule we can use to get the type. This means this expression is not well-typed and the type checker would return an error that will happen if compile.rkt tried to run it. This expression is also commented in "test-runner.rkt" which would cause a type mismatch error in "compile.rkt" in the test folder as a similar algorithm runs in the implementation with its functions that represent the type systems.

Many other tests are listed in the test folder in "test-runner.rkt". The commented tests result in type mismatch errors. "compile.rkt" runs these. Outside the test folder is "text.rkt" has the first working expression and "compile-file.rkt" runs this. From the evaluation of running "compile.rkt", the compiler passed all the tests that are supposed to pass the type checker and throw an error for the tests that fail the type checker.

4 Typed Bumble

The next implementation of a type system is Typed Bumble. This type system is the first step that transitions from the type system laid out in chapter 8 of Types of Programming Languages to Typed Racket. The two main additions of the language are one-parameter functions with type annotation and Union types like the Any type. These additions affect "parse.rkt", "compile.rkt", and "ast.rkt" which will be loosely based on Iniquity from the CMSC838E lecture notes. In addition, "ast.rkt" and "typecheck.rkt" will have some new additions from Typed Alpha based on the new union and typed annotated one parameter function types.

4.1 Syntax

The additions in the syntax from Typed Alpha include program syntax represented by p which is represented by a list of definitions and the main expression. A definition is represented by d which is represented by a variable name and the abstraction. An abstraction is represented by the input variable name, the first-class type of the input variable, the body expression, and the first-class type of the output

$$
\begin{array}{lr}
p ::= (\text{begin } \overrightarrow{d}\, e) & \text{Program} \\
d ::= (\text{define } f\ a) & \text{Definition} \\
a ::= \lambda x^{\tau_f}.e \rightarrow \tau_f & \text{Abstraction} \\
e ::= (\text{if } e\ e\ e) \mid (\text{c } e) \mid (f\ e) \mid \#\text{t} \mid \#\text{f} \mid n \mid x & \text{Expressions} \\
c ::= add1 \mid sub1 \mid zero? \mid number? \mid boolean? & \text{Primitive Operations} \\
\tau_f ::= \mathbf{N} \mid \mathbf{\#t} \mid \mathbf{\#f} \mid (\cup\, \overrightarrow{\tau_f}) & \text{First-Class Types} \\
\tau ::= \tau_f \mid \tau_f \rightarrow \tau_f & \text{Types} \\
\Gamma ::= \emptyset \mid \Gamma, x : \tau_f & \text{Variable Environment} \\
\Gamma_f ::= \emptyset \mid \Gamma, x : \tau & \text{Function Environment}
\end{array}
$$

Figure 3. Typed Bumble Syntax

that the expression is supposed to return. The variable reference, x, has been added in the expression which are symbols that are not primitives/functional constants. There are also function references, f, which are also symbols that are not primitives/function constants. Syntactically, there are no differences from variables references, but they will be taken care of in the typing rules to enforce that functions are not first-class. The new functional constants are *number*? and *boolean*?ȷ that do dynamic type checking on an expression of the Any type, which will be explained below.

For **types**, the new addition is the union type of $(\cup\, \overrightarrow{\tau_f})$ which represents the union of any type in its list $\overrightarrow{\tau_f}$ which means that an expression of this type could be any type in $\overrightarrow{\tau_f}$. In addition, union types will follow conventional union rules. We abbreviate $(\cup\ \ \#t\ \#f)$ as **B**, booleans. We also abbreviate $(\cup$ (list of all the non-union first-class types)) which is equivalent to $(\cup$ (list of all the first-class types)) as **T**, which represents Any, the supertype of all first-class types.

For **environments**, they are represented by the variable environment that handles variables and the type environment that handles functions defined in the list of definitions in the program. A type environment is a list of bindings of the name of the function or variable with its type. One thing to note is that due to the nature of only having one parameter function definitions, the variable environment will only have zero or one bindings.

4.2 Typing Rules

Typed Bumble has new typing rules and subtyping rules as shown in figure 4 and 5.

Program: T-PROG is added where it checks that each abstraction in each function definition is well-typed given the type bindings of every function that was defined is added to the type environment and ends up returning the type of the main expression given that every function that was defined is added to the function type environment.

Abstraction: T-ABS is added where it takes an abstraction consisting of the input variable name, the first-class type of the input variable, the body expression, and the first-class type of the output variable. It then gets the type of the body expression with the binding pair of the name and type of the input variable added to the type environment. If the expression's type is the same as the output type, the abstraction is the output type.

$$\frac{\overrightarrow{\qquad\qquad\qquad} \qquad \Gamma \vee \Gamma_f, \overrightarrow{f : \tau^i} \vdash e : \tau'}{\Gamma \vee \Gamma_f \vdash (\text{begin } \overrightarrow{d}^{\,i}\ e) : \tau'}(\textit{T-PROG})$$

$$\frac{\Gamma, x : \tau \vee \Gamma_f \vdash e : \tau'}{\Gamma \vee \Gamma_f \vdash \lambda x^{\tau}.e \rightarrow \tau' : \tau \rightarrow \tau'}(\textit{T-ABS})$$

$$\frac{\Gamma \vee \Gamma_f \vdash e : \tau \qquad \vdash \tau <: \tau'}{\Gamma \vee \Gamma_f \vdash e : \tau'}(\textit{T-SUBSUME})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash n : \mathbf{N}}(\textit{T-NUM})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash \#t : \mathbf{\#t}}(\textit{T-TRUE})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash \#f : \mathbf{\#f}}(\textit{T-FALSE})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash c : \delta_\tau(c)}(\textit{T-CONST})$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vee \Gamma_f \vdash x : \tau}(\textit{T-VAR})$$

$$\frac{\vdash c : \tau \rightarrow \tau' \qquad \vdash e : \tau}{\Gamma \vee \Gamma_f \vdash (c\ e) : \tau'}(\textit{T-PRIM})$$

$$\frac{\Gamma \vee \Gamma_f \vdash e_1 : \tau \qquad \Gamma \vee \Gamma_f \vdash e_2 : \tau' \qquad \Gamma \vee \Gamma_f \vdash e_3 : \tau'}{\Gamma \vee \Gamma_f \vdash (if\ e_1\ e_2\ e_3) : \tau'}(\textit{T-IF})$$

$$\frac{f : \tau \rightarrow \tau' \in \Gamma_f \qquad \Gamma \vee \Gamma_f \vdash e : \tau}{\Gamma \vee \Gamma_f \vdash (f\ e) : \tau'}(\textit{T-APP})$$

Figure 4. Typed Bumble Typing Rules

$$\frac{}{\vdash \tau <: \tau}(\textit{S-REFL})$$

$$\frac{\overrightarrow{\tau_i <: \tau'^i}}{\vdash (\cup \overrightarrow{\tau}^i) <: \tau'}(\textit{S-UNIONSUB})$$

$$\frac{\exists i. \vdash \tau <: \tau'_i}{\vdash \tau <: (\cup \overrightarrow{\tau'}^i)}(\textit{S-UNIONSUPER})$$

Figure 5. Typed Bumble Subtyping Rules

Constants: Nothing notable was changed.

Variables: T-VAR was added which states that if the variable type environment contains the pairing of $x : \tau$, x has type τ.

Primitive Operations: T-PRIM remains unchanged from before.

Subsumption: Subsumption is mainly useful to determine subtypes for union types. There is the standard rule of S-REFL which states a type is a subtype of itself. In addition, there is S-UNIONSUB which states a union is a subtype of another type if each variable in the union is a subtype of the other type. S-UNIONSUPER states that a type is a subtype of a union type if there exists of type in the list of types in the union that is a supertype of that other type. There is then T-SUBSUME which states if e is of type τ and τ is a subtype of τ', e is also of type t'.

Conditionals: The T-IF rule given the syntax (if e_1 e_2 e_3) no longer requires e1 to be a Boolean as it can now be any type which is its only direct change.

Application: T-APP was added given the syntax (f e). It states that if the functional environment proves that f has the type $\tau \rightarrow \tau'$ and e is of type τ, the expression (f e) is of type τ'.

4.3 Implementation

Many changes to "compile.rkt", "parse.rkt", and "ast.rkt" from Typed Alpha were based on iniquity. In addition, "ast.rkt" had the Fun structure to represent the function type and the *Union* structure to represent the Union type. It includes the *Binding* structure with the symbol x and the type ty that is used for the context. Also, the $Defn$ structure from iniquity was modified to include a list of type bindings for the input variable as opposed to just a list of names for the input variables. The "any" and "boolean" constants were defined to represent how we shorthanded them. The *simp-union* function is used to get a simplified version of the list of types in a Union type based on Union rules. The file, "parse.rkt", also accounts for type annotations for the one-parameter user-defined functions similar to Typed Racket.

The main file, "typecheck.rkt", was modified to account for type rules. In addition, print-type was updated to include the Union types.

Program: Since programs are no longer just expressions, get-type now matches with the *Progstructure*. It simulates T-PROG by first initializes its function environment with *get-type-funs* which gets every name and type of every definition in the program. It then runs get-type-defines to see that every type given by T-ABS for the definitions matches the intended function type. After doing that, it will then call get-type-e to return the type of the main expression. Environments: The type environment that handles variables is represented in other functions as *env*, and the type environment that handles functions is represented as $fenv$. Both are lists of the *Binding* structures.

Subsumption: The act of checking if something is a subtype is handled by *is-subtype*?, which replaces *type-eq*?, where it checks that the second type $ty2$ is a subtype of first type $ty1$. S-REFL is handled by matchings where the two types are the same type. S-UNIONSUB is handled by the matching that $ty2$ is of a union type which calls *is-subtype-union*2? that implements the rule to check if $ty2$ is a subtype of $ty1$. S-UNIONSUPER and S-wUNIONSUB is handled by the matching that $ty1$ is of a union type which calls *is-subtype-union*1? that implements the rule to check if $ty2$ is a subtype of $ty1$. Instead of a rule for subsumption, *get-supertype* is implemented to the most simplified common supertype which either one of the types of the Union of the two types.

Abstraction: T-ABS is simulated with *get-type-define*. It first adds its given type binding for its one parameter to the variable type environment and calls *get-type-e* on the body expression e that the definition structure has and checks if it returns a subtype of the intended output type. If this case holds, it will return the intended function type. This return type will not be used anywhere since functions are not first-class.

Constants: *get-type-op*1 was updated to include *number*? and *boolean*?.

Variables: T-VAR is handled by pattern matching with the Var structure with the symbol v and a call to the function, *lookup-type*, with the parameter v and the variable type environment. *lookup-type* takes a symbol and an environment and checks if the symbol is in a pairing in the list. If that pairing exists, the associated type gets returned and if it does not, a lookup error gets thrown.

Primitive Operations: *get-type-op*1 does not have a specific, non-universal change from Typed Alpha.

Conditionals: The functional implementation of T-IF, *get-type-if*, which takes no longer throws a type mismatch error if e1 is not of type boolean. Also, since every type has a common supertype, the function now just returns the most simplified, common supertype with a call to *get-supertype* with *e*2 and *e*3.

Application: T-APP is implemented with get-type-app. "get-type-app" calls lookup-type on the symbol f and the function type environment to get the type of f. This type gets matched with the (*Fun in-ty out-ty*) structure where it checks if the argument is a subtype of *in-ty* with *is-subtype*?. If it is, *out-ty* will get returned. If it is not, a type mismatch error is thrown.

4.4 Examples and Evaluation:

Like before, we will take a look at an example of an expression that will pass the type checker and successfully compile. Then, we will take a look at an expression that will fail the type checker and throw a type mismatch error.

The expression we will look at first is:

((begin (define (f (x : Boolean)) : (U Number Boolean)
(if x 5 #t)) (f #t))

Since this is a program, we must apply T-PROG. Since (f (x : Boolean)) : (U Number Boolean) (if x 5 #t)) is the only definition in the list of definitions, we would only need to add f: **B** → (∪ **N B**) to the functional environment and get the type of the expression (if x 5 #t) based on the type environment and given a variable environment with x:**B** added to it. Using T-IF, we would check if x was well-typed which using T-VAR where we would check x:**B** is in the variable environment, we would conclude x is of type **B** . Then, we w use T-NUM to show 5 is of type **N**. We would then use T-TRUE to show #t is of type **#t**. We would then use S-UNIONSUPER to show that #t is a supertype of **B** which is (∪ **#t #f**) since the list contains **#t** which is a supertype of **#t** through T-refl. Using S-UNIONSUPER to show (∪ **N #t**) is a supertype of both of **N** and **#t** and then T-SUBSUME to show 5 and #t are both of type (∪ **N #t**) meaning (if x 5 #t) is of type (∪ **N #t**). Using S-UNIONSUB and UNION-SUPER, we can show (∪ **N #t**) is a subtype of (∪ **N B**) meaning through T-ABS, (f (x : Boolean)) : (U Number Boolean) (if x 5 #t)) would be of type **B** → (∪ **N B**). To complete, T-PROG, we would need to find the type of (f #t) given the type environment added f : **B** → (∪**N B**). Using T-APP, we would get that f is of type **B** → (∪ **N B**). Using, T-FALSE, S-UNIONSUPER, and T-SUBSUME, we can show #f is of type **B**. Completing T-APP, that would make (f #t) of type (∪ **N B**) as its type and completing T-PROG would make the program of type (∪ **N B**).

The implementation works a similar way by calling functions that simulate the typing rules. The implementation has the type checker pass this test and many others that are listed in "test-runner.rkt".

An example of an expression that is not well-typed is:

(begin (define (f (x : Any)) : (U Boolean Number) (add1 x))
(f 5))

The process works similarly to before. Since this is a program, we must apply T-PROG. Since (f (x : Any)) : (U Boolean Number) (add1 x)) is the only definition in the list of definitions, we would only need to add f: **T** → (∪ **N B**) to the functional environment and get the type of the expression (add1 x) based on the type environment and given a variable environment with x : **T** added to it. Using T-PRIM, we would get the expression of (add1 x). Using T-CONST, we would learn that add1 is of type **N** → **N**. This would mean for (add1 x) to be a type, x would need to be of type **N**. However, this is not the case as the only rule that can be applied on x is T-VAR which would make x type **T** and any supertype of **T**. However, **T** is also known as (∪ **N #t**) **#f**) which is not a subtype of **N** as for it to be a subtype, every type in the list in the union must be a subtype of **N** which is not true. **#t** is not a subtype of **N** for example. Since there is no other rule to handle primitives, there is no way to determine the type of (begin (define (f (x : Any)) : (U Boolean Number) (add1 x)) (f 5)) which is why the implementation throws a Type Mismatch error after going through its type rule functions.

Like before, many other tests are listed in the test folder in "test-runner.rkt" in the same format as before which Typed Bumble passes or for the commented ones, throws a Type Mismatch error. Another interesting program that we can write in this language that the type checker accepts is:

(begin (define (f (x : Number)) : Number
(if (zero? x) 0 (add1 (add1(f (sub1 x)))))) (f 7))

This program recursively calculates 7 * 2. I will not elaborate on how the program type checks, but this program highlights the expressiveness of the Typed Bumble. It is both in "test-runner.rkt" and "text.rkt". It can be compiled using Typed Bumble with "compile-file.rkt".

5 Typed Control

The final implementation of type system is Typed Control. This type system will be based on $_{TR}$ described in Logical Types for Untyped Languages. The main addition is occurrence typing which uses common reasoning methods to assign variable occurrences a type based on predicates prior to the occurrences. To do this, the judgment of the type system is

$$\Gamma \vee \Gamma_f \vdash p : \tau; \psi+|\psi-; o$$

Which states that in environment $\Gamma \vee \Gamma_f$, the program p (or expression e) has type t, comes with then proposition $\psi+$ and else proposition $\psi-$, and references object o. If p evaluates to true, then the proposition $\psi+$ holds and if it evaluates to false, else proposition $\psi-$, holds. Due to the simplicity of the language, most of the propositions will be quite basic. I will now summarize the syntax, rules, and implementation.

p ::= (begin $\overrightarrow{d}$ e) — Program
d ::= (define f a) — Definition
a ::= $\lambda x^{\tau_f}.e \rightarrow \tau_f$ — Abstraction
e ::= (if e e e) | (c e) | (f e) | #t | #f | n | ch | x — Expressions
c ::= $add1$ | $sub1$ | $zero?$ | $number?$ | $boolean?$ | $char?$ | $char\text{->}integer$ | $integer\text{->}char$ — Primitives
τ_f ::= **N** | **#t** | **#f** | **C** | ($\cup$ $\overrightarrow{\tau_f}$) — First-Class Types
τ ::= τ_f | $x : \tau_f \frac{\psi+|\psi-}{o} \tau_f$ | **E** — Types
ψ ::= τ_x | $\bar{\tau}_x$ | $\psi \vee \psi$ | tt | ff — Propositions
o ::= $\varnothing$ | x — Objects
Γ ::= Ø | Γ, ψ — Variable Environment
Γ_f ::= Ø | Γ, ψ — Function Environment

Figure 6. Typed Control Syntax

5.1 Syntax

The expression syntax, as listed in figure 6, has not changed much other than characters being added. A new thing not mentioned is that (begin (list of d) e) can be simplified to (begin d_1 (begin d_2 ... (begin d_n e) ...)). This means that instead of a program being equivalent to a specific version of let-rec, it is equivalent to a specific version of the regular let* expression meaning there is no recursion in Typed Control. One change to types is that characters were added and T would include that.

A slight change to **types** is that function type was modified where it also includes the name of its input parameter, the if proposition, the else proposition, and the object that would be outputted with the output type in the judgment if this function was ever applied. Another change was the addition of the Explosion type represented by **E** which is a special type that is due to always-false (ff) proposition being in the variable environment which would lead to an explosion of the system making every expression being type Explosion. Expression has type **E** if it is every type at the same time which can only happen if the environment explodes due to the contradiction proposition of ff. In addition, the character type, **C** was added which would affect the definition of **T**.

Some **propositions** are familiar as they include conjunctions, always-true (tt), and always-false (ff). There are also type propositions, τ_x that states x has type τ and $\bar{\tau_x}$ that states x does not have type τ. These propositions work as expected with short-circuiting happening for conjunctions and type propositions working together as they would intuitively.

An **object** is a portion of the runtime environment that can either be a variable or an empty object.

Environments have changed in that now that both environments are lists of propositions as opposed to lists of type bindings as they were before. If two environments can prove each other, they are the same environment.

5.2 Typing Rules

The universal modification to the new typing rules and subtyping rules as shown in figure 7, 8, and 9 were based on having more information from the judgment. The other changes will be elaborated on.

Program: Since programs are now equivalent to a let* expression where there are bindings of only abstractions and the main expression, the program can be simplified as a nested define statement with T-PROG. After doing that, it can do T-DEFINE where it gets the judgment information that gives Type τ, $\psi+|\psi-$, and o from the abstraction body of the single definition. It would then get the judgment information of the main expression with τ_f added to the function environment which would be Type τ', $\psi+'|\psi-'$, and o'. Given this information, the judgment information of this defined statement that is recursively nested from T-DEFINE is τ', $\psi+'|\psi-'$, and o' with o replacing every instance of the function's name f.

Subsumption: This has remained the same except S-ESUB and S-ESUP which both state E is the subtype and supertype of every type. In addition, SO-REFL and SO-TOP were added for object subsumption.

Explosion: This rule of T-EXPL states that if the variable environment proves ff, then the expression e is type **E**.

Abstraction: T-ABS now has its input variable name, a then proposition, else proposition, and object where it is the same as the input variable name if proposition, else proposition, and object from the body expression in the abstraction.

Constants: Only thing that changed was the judgment information was added in all the constant typing rules and the addition of T-CHAR to handle characters.

Variables: T-VAR had its judgment information changed, so we know the then proposition is that x is not #f and the else proposition is that x is #f. It is the only typing rule that explicitly returns a variable for its x.

Primitive Operations: T-PRIM stays relatively the same, but with added judgment information.

Conditionals: T-IF displays how occurrence typing works. Given (if e_1 e_2 e_3), it gets the judgment information of e1

$$\frac{\Gamma \vee \Gamma_f \vdash (\text{begin } d_1 \text{ (begin } d_2 \text{ ... (begin } d_n \text{ e) ...)) } : \tau; \psi+|\psi-; o}{\Gamma \vee \Gamma_f \vdash (\text{begin } \overrightarrow{\text{d}}\ e) : \tau; \psi+|\psi-; o}(\textit{T-PROG})$$

$$\frac{\Gamma \vee \Gamma_f \vdash f : \tau; \psi+|\psi-; o \qquad \Gamma \vee \Gamma_f, \tau_f \vdash e : \tau'; \psi+'|\psi-'; o'}{\Gamma \vee \Gamma_f \vdash (\text{begin (define } f\ a) \text{ e}) \ : \ \tau'[o/f]; \psi+'[o/f]|\psi-'; o'[o/f]}(\textit{T-DEFINE})$$

$$\frac{\Gamma \vee \Gamma_f \vdash e_1 : \tau; \psi_1+|\psi_1-; o \qquad \Gamma, \psi_1+ \vee \Gamma_f \vdash e_2 : \tau'; \psi_2+|\psi_2-; o' \qquad \Gamma, \psi_1- \vee \Gamma_f \vdash e_3 : \tau'; \psi_3+|\psi_3-; o'}{\Gamma \vee \Gamma_f \vdash (if\ e_1\ e_2\ e_3) : \tau'; \psi_2+ \vee \psi_3+|\psi_2- \vee \psi_3-; o'}(\textit{T-IF})$$

Figure 7. Typed Control Typing Rules of T-PROG, T-DEFINE, and T-IF

which is $\tau; \psi_1+|\psi_1-; o$. It then gets the judgment information of e_2 with ψ_1+ added to the variable environment which ends up being $\tau'; \psi_2+|\psi_2-; o$. It would then get the judgment information of e_3 with ψ_1- added to the variable environment which ends up being $\tau; \psi_3+|\psi_3-; o$. The ending judgment information of the whole expression is $\tau'; \psi_2+ \vee \psi_3+|\psi_2- \vee \psi_3-; o'$

Application: T-APP is similar to before but now integrated with the judgment information and substitution. Given (f e), it gets the type of f which is $(\tau \frac{\psi_f+|\psi_f-}{o_f} \tau')$. It then gets the judgment information of e which is $\tau; \psi+'|\psi-'; o'$. From this, it will give the judgment information of $\tau'; \psi_f+; \psi_f-; o_f+$ where o' replaces all instances of x.

5.3 Implementation

The main changes were to "ast.rkt" and "typecheck.rkt". "ast.rkt" was mainly changed to have structures that implemented the syntax changes that will be explained below.

The main file, "typecheck.rkt", was modified to account for type rules that also changed. In addition, the helper functions were transferred to another file called "typecheck.rkt". In addition, all functions were changed to return a judgement structure, *Judge*, consisting of the type, the then proposition, else proposition, and object.

Environments: Because of the simplicity of only having type propositions and conjunction propositions and how there can only be one variable in a proposition due to variables only appearing in one parameter functions, a variable environment can be able to have one proposition that variable x has type τ for each variable x due to implication. This simplification is done any time a proposition is added with the function "add-prop" that makes use of other helper functions that handle adding propositions that say variable x is type τ, variable x is not type τ or a conjunction proposition. This function and helper functions take advantage of rules relating to union types to make the single proposition that x has type τ. The helper function that takes care of adding conjecture propositions. *add-prop-or* takes advantage of rules specific to conjunctions. Lastly, instead of checking every time if ff is in the list of propositions, the

$$\frac{\Gamma, x : \tau \vee \Gamma_f \vdash e : \tau'; \psi+|\psi-; o}{\Gamma \vee \Gamma_f \vdash \lambda x^{\tau}.e \rightarrow \tau' : x : \tau \frac{\psi+|\psi-}{o} \tau'}(\textit{T-ABS})$$

$$\frac{\Gamma \vee \Gamma_f \vdash e : \tau'; \psi+|\psi-; o \qquad \vdash \tau <: \tau' \qquad \vdash o <: o'}{\Gamma \vee \Gamma_f \vdash e : \tau'; \psi+|\psi-; o'}(\textit{T-SUBSUME})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash n : \mathbf{N}; tt|ff; \emptyset}(\textit{T-NUM})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash \#t : \#\mathbf{t}; tt|ff; \emptyset}(\textit{T-TRUE})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash \#f : \#\mathbf{f}; ff|tt; \emptyset}(\textit{T-FALSE})$$

$$\frac{}{\Gamma \vee \Gamma_f \vdash c : \delta_\tau(c); tt|ff; \emptyset}(\textit{T-CONST})$$

$$\frac{\tau_x \in \Gamma}{\Gamma \vee \Gamma_f \vdash x : \tau; \#\mathbf{f}_x|\#\bar{\mathbf{f}}_x; x}(\textit{T-VAR})$$

$$\frac{\Gamma \vdash ff}{\Gamma \vee \Gamma_f \vdash e : \mathbf{E}; tt|tt; \emptyset}(\textit{T-EXPL})$$

$$\frac{\vdash c : x : \tau \frac{\psi_f+|\psi_f-}{o_f} \tau'; \psi+|\psi-; o \qquad \vdash e : \tau; \psi+'|\psi-'; o'}{\Gamma \vee \Gamma_f \vdash (c\ e) : \tau'[o'/x]; \psi_f+|\psi_f-; o_f}(\textit{T-PRIM})$$

$$\frac{(\tau \frac{\psi_f+|\psi_f-}{o_f} \tau')_f \in \Gamma_f \qquad \vdash e : \tau; \psi+'|\psi-'; o'}{\Gamma \vee \Gamma_f \vdash (f\ e) : \tau'[o'/x]; \psi_f+[o'/x]|\psi_f-[o'/x]; o_f[o'/x]}(\textit{T-APP})$$

Figure 8. Typed Control Typing Rules except T-PROG, T-DEFINE, and T-IF

variable environment has a Boolean variable to determine whether it exploded or not and can give every type **E** to save runtime, so it would only check if ff is in the system once which loosely simulated by *find-contra*. Also, handling type

E gives the benefit of always saying that expressions that are guaranteed to never run will always be typed checked improving the expressiveness of the program.

Program: *get-judge* simulates T-PROG. Since programs are now equivalent to a let* expression where there are bindings of only abstractions and the main expression, T-PROG ends up calling $get\text{-}judge\text{-}help$, the equivalent to T-DEFINE, on the program by simplifying it to nested define expression. It then works as much as it is described in T-DEFINE, including the substitution of every instance of f with obj which is implemented by the functions *subs-type*, *subs-prop*, and $subs\text{-}obj$.

Subsumption: This has remained the same except S-ESUB and S-ESUP are now implemented in *is-subtype*?. In addition, SO-TOP is shown in $get\text{-}superobj$ which gets the common super object of two objects or nothing if they do not exist.

Abstraction: Implements T-ABS as described above with $get\text{-}judge\text{-}body$. It throws a type mismatch error if the system has not exploded and if the body expression is not the subtype of the expected output type.

Constants: Changes the functions that implement rules relating to constants to now return judgment structures with the correct information. Also, added pattern matching case in $get\text{-}judge\text{-}e$ to implement T-CHAR and the char function cases in the pattern matching of $get\text{-}judge\text{-}op1$

Variables: T-VAR was implemented through a pattern-matching case in $get\text{-}judge\text{-}e$ to now include the information from the judgment structure

Primitive Operations: $get\text{-}judge\text{-}prim1$ implements T-PRIM with added judgment information. It throws a type mismatch error if the system has not exploded and if the operative is not the subtype of the expected input type of the primitive operator.

Conditionals: $get\text{-}judge\text{-}if$ implements T-IF exactly as described.

Application: $get\text{-}judge\text{-}app$ implement T-APP as described. It throws a type mismatch error if the system has not exploded and if the argument is not the subtype of the expected parameter.

5.4 Example and Evaluation

For an example, we will take a look at one that takes advantage of occurrence typing The expression we will look at first is:

```
(begin ((begin (define (f (x : (U Number Char))) : Number
                (if (number? x) (add1 x) (char->integer x))) (f #\a))
```

Since this is a program, we must apply T-PROG. Since there is only one definition, it makes it pretty easy to simplify this program to use T-DEFINE. We use T-DEFINE to get the judgment information of this begin expression starting with the definition. We use T-ABS to get the judgment information of this body (x : (U Number Char))) : Number

$$\frac{}{\vdash o <: o}(SO\text{-}REFL)$$

$$\frac{}{\vdash o <: \emptyset}(SO\text{-}TOP)$$

$$\frac{}{\vdash \tau <: \mathbf{E}}(S\text{-}ESUP)$$

$$\frac{}{\vdash \mathbf{E} <: \tau}(S\text{-}ESUB)$$

Figure 9. Newly Added Subtyping Rules in Typed Control

$$\begin{aligned}
\delta_\tau(add1) &= \mathbf{N} \to \mathbf{N} = x{:}(\mathbf{N}\tfrac{tt|ff}{\emptyset}\mathbf{N}) \\
\delta_\tau(sub1) &= \mathbf{N} \to \mathbf{N} = x{:}(\mathbf{N}\tfrac{tt|ff}{\emptyset}\mathbf{N}) \\
\delta_\tau(zero?) &= \mathbf{N} \to \mathbf{B} = x{:}(\mathbf{N}\tfrac{tt|tt}{\emptyset}\mathbf{B}) \\
\delta_t(number?) &= x{:}\mathbf{T}\tfrac{\mathbf{N}_x|\bar{\mathbf{N}}_x}{\emptyset}\mathbf{B} \\
\delta_\tau(boolean?) &= x{:}\mathbf{T}\tfrac{\mathbf{B}_x|\bar{\mathbf{B}}_x}{\emptyset}\mathbf{B} \\
\delta_\tau(char?) &= x{:}\mathbf{T}\tfrac{\mathbf{C}_x|\bar{\mathbf{C}}_x}{\emptyset}\mathbf{B} \\
\delta_\tau(char\text{->}integer) &= \mathbf{C} \to \mathbf{N} = x{:}(\mathbf{C}\tfrac{tt|ff}{\emptyset}\mathbf{N}) \\
\delta_\tau(integer\text{->}char) &= \mathbf{N} \to \mathbf{C} = x{:}(\mathbf{N}\tfrac{tt|ff}{\emptyset}\mathbf{C})
\end{aligned}$$

Figure 10. Constant Typing: Right is for Typed Control

(if (number? x) (add1 x) (char->integer x)). Using T-ABS, we would get the judgement information of (if (number? x) (add1 x) (char->integer x)) with the proposition that x has type $(\cup\ \mathbf{N}\ \mathbf{C})$ to the environment. Then, we would do T-IF where we first get the judgment information of (number? x). To get (number? x), we would use T-PRIM which uses T-CONST to get number? which would say it is $x{:}\mathbf{T}\frac{\mathbf{N}_x|\bar{\mathbf{N}}_x}{\emptyset}\mathbf{B}; tt|ff;$. It would then get judgment information of x which using T-VAR and T-SUBS is a subtype of T meaning that (number? x) is of $\mathbf{B}; \mathbf{N}_x|\bar{\mathbf{N}}_x; \emptyset$. Keeping with T-IF, we would then check (add1 x) with $textbf{N}_x$ added to the variable environment that included $(\cup\ \mathbf{N}\ \mathbf{C})_x$ which would simply to just simplify the variable environment to $\mathbf{N}_x$. Due to this, using T-CONST to get judgment information of add1 that includes it being type $x{:}\mathbf{N}\frac{tt|ff}{\emptyset}\mathbf{N}$ and T-VAR to get the prop $\mathbf{N}_x$ meaning it is of type **N**, we can determine (add1 x) is of type **N**. As we move on to the else statement in T-IF, we would check (char->integer) with $\bar{\mathbf{N}}_x$ added to the variable environment that has $(\cup\ \mathbf{N}\ \mathbf{C})_x$ which by simplification, the environment has $\mathbf{C}_x$. Using Due to this, using T-CONST to get judgment information of char->integer that includes it being type $x{:}\mathbf{C}\frac{tt|ff}{\emptyset}\mathbf{N}$ and T-VAR to get the prop $\mathbf{C}_x$ meaning it is of type **C**, we can determine (char->integer x) is of type **N**. Since (add1 x) and (char->integer x) are both **N**, f (x : (U Number Char)))

: Number (if (number? x) (add1 x) (char->integer x)) would return x:$(\cup\ \mathbf{N}\ \mathbf{C})^{\frac{tt|ff}{\varnothing}}\mathbf{N}$. Following T-DEFINE, we would get the judgment information of the expression, (f #\a) given we add f:$(\cup\ \mathbf{N}\ \mathbf{C})$ -> **N** to the function environment. We would use T-APP on (f #\a), where we would check the variable environment that f is type x:$(\cup\ \mathbf{N}\ \mathbf{C})^{\frac{tt|ff}{\varnothing}}\mathbf{N}$, and we use T-CHAR and T-SUBS to figure out #\a is of type **C** and therefore $(\cup\ \mathbf{N}\ \mathbf{C})$ meaning (f #\a) would return $\mathbf{N}; tt||ff; \varnothing$. The result is $\mathbf{N}; tt||ff; \varnothing$.

If we did not implement occurrence typing in Typed Control, this expression would not pass type-checking which shows the power of this language. Other expressive programs that test correctness are included in "test-runner.rkt" with similar formats to the last two languages.

6 Additional Evaluation and Future Work

In addition to the tests created to check correction of Typed Alpha, Typed Bumble, and Typed Control, I will explain the intuitive benefits of each language compared to its dynamic counterpart and other statically typed languages. Extensions will have the benefits of the previous extension.

Typed Alpha: Due to the addition of a type checker that checks if a language is well-typed, there can be no more type runtime errors or any automatic dynamic type checking which saves time for compilation. In addition, a correct type checker guarantees there will be no runtime errors since all the runtime errors in Con are type errors.

Typed Bumble: The addition of forced type annotations for user-defined functions enforces user documentation by the user which leads to more transparent code and fewer errors. In addition, the Union type provides more expressiveness than a basic static type system due to the implied addition of polymorphism which gives the option for functions to take in more expressions and output more expressions.

Typed Control: The addition of occurrence typing gives the user the option to write more expressive programs with expressions that are guaranteed to be safe by using propositional logic from previous predicates before the occurrence. The applications include anywhere from type checking certain variables to always saying "dead code" is well-typed due to the explosion type which would both save time for the type checker, but also allow more safe programs.

Limitations and Future Work: Since Typed Control is the last extension in the paper, I will mainly be talking about its limitations. Its main limitation is that it is too simple as it lacks many features from both λ_{TR} and Villain. Because of its simplicity, other propositions in λ_{TR} were not added like implications and disjunctions. Features that were mentioned in both Logical Types for Untyped Languages and are in the Villain compiler include first-class lambdas, pairs, local binding, and multiple arguments for functions. Another feature that was cut from the transition from Typed Bumble to Typed Control was recursive functions. These features were cut out due to time constraints but may be added in the future in the next extensions.

In addition, more work that could be done include writing a proof of type soundness with evaluation rules, a proof of progress, and proof of preservation. Having a proof like this would prove that any well-typed program in the systems will not get stuck which will formally show the benefits of my statically typed languages to its dynamically checked counterpoints.

7 Conclusion

This paper describes three extensions to the dynamically typed languages in the CMSC8383E to incrementally get features of Typed Racket. These features include static type checking, type annotated functions, union types, polymorphism, and occurrence typing. Having these features of both static type system and environments that use proposition logic leads to both transparent, safe programs that remove run type errors and automatic type checking that can also be as fairly expressive as safe programs dynamic typed languages.

References

[1] David Van Horn. 2021. Notes. https://www.cs.umd.edu/class/spring2021/cmsc838E/Notes.html

[2] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[3] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. *SIGPLAN Not.* 45, 9 (Sept. 2010), 117–128. https://doi.org/10.1145/1932681.1863561