2016

THE DRAG LANGUAGE

Weixi Ma
University of Kentucky, wma225@g.uky.edu
Digital Object Identifier: http://dx.doi.org/10.13023/ETD.2016.018

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Weixi Ma, Student

Dr. Raphael Finkel, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

THE DRAG LANGUAGE

THESIS

A thesis submitted in partial
fulfillment of the requirements for
the degree of Master of Science in
the College of Engineering at the
University of Kentucky

By
Weixi Ma
Lexington, Kentucky

Director: Dr. Raphael Finkel, Professor of Computer Science
Lexington, Kentucky 2016

THESIS

THE DRAG LANGUAGE

This thesis describes the Drag language.
Drag is a general purpose, gradually typed, lexically scoped, and multi-paradigm programming language. The essence of Drag is to build the abstract syntax trees of the programs directly and interactively.
Our work includes the language specification and a prototype program. The language specification focuses on the syntax, the semantic model, and the type system. The prototype consists of an interactive editor and a compiler that targets several platforms, among which we focus on the LLVM platform in this thesis.

KEYWORDS: Programming Languages, Abstract Syntax Tree, Interactive Programming, Direct Manipulation, Integrated Development Environment

Author's signature: Weixi Ma

Date: February 4, 2016

THE DRAG LANGUAGE

By
Weixi Ma

Director of Dissertation: Raphael Finkel

Director of Graduate Studies: Miroslaw Truszczynski

Date: February 4, 2016

To Languages

Acknowledgments

Drag would not be possible without the help of many people.

The first Thank You goes to my adviser, Dr. Raphael Finkel, for keeping a perfect balance between allowing me the freedom to explore my ideas and providing me with guidance, for teaching me the way of communicating precisely and effectively in research and for providing me with essential raw materials in his programming languages class and compiler class. More importantly, Dr. Finkel has set an example of an excellent educator and researcher I aspire to be.

I am grateful to Dr. Jerzy Jaromczyk and Dr. Henry Dietz for serving as my committee members and want to acknowledge how much I treasure their guidance and help.

A part of the inspiration of Drag comes from the Formal Semantic class of Dr. Greg Stump. In addition, Dr. Stump has also enlightened me to bring many methodologies originating from linguistics into my work.

I would like to thank Dr. Tingting Yu, who helped me realize the possibility of Drag being a software testing platform and guided me to implement the prototype of incremental symbolic execution in Drag.

Finally, but not least, I would like to thank my family, Ping Ma, Zhan Lu and Ruiting Zhao, who have been very important to me during this work — not only for their love and support, but also for providing different ideas and skepticism.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

1.1 Background and Terms

- `Programming Language` — In this thesis, we define **programming language** in a wide sense – a software tool that interfaces between clients, usually humans, and lower-level facilities, such as files or operating systems [1].
- `Programming Paradigm` — A **programming paradigm** is a style of computer programming. Some widely-known paradigms include: the imperative paradigm, which focuses on changing the states and the flow of control; the functional paradigm, which models program execution as the evaluation of expressions; and the declarative paradigm, which views programming as stating what is wanted but not necessarily how to compute it [1].

  A programming language may implement many paradigms, like OCaml and Scala. In this thesis, we call such a language multi-paradigm.
- `IDE` — An IDE, **integrated development environment**, is software that helps programmers write programs. An IDE is often able to scan and parse the source program to provide some aids for editing the programs, such as autocompletion, grammar highlight, refactoring, and declaration jumping.Two examples of IDEs are Visual Studio and IntelliJ. Some programming languages are highly integrated into their IDEs — for instance, Matlab.
- `AST` — An AST, **abstract syntax tree**, is a representation of the abstract syntax structure of a string-formed program. This representation serves as the central data structure for all post-parsing activities during compilation [2].
- `LLVM` — The LLVM, **low level virtual machine**, is a compiler tool-chain. The core of LLVM is its instructions set.
- `S-expression` — The **S-expression**, invented for the programming language LISP [3], is a nested list structure that represents LISP programs and data uniformly. An S-expression is recursively defined as

  – `an atom or`

  – `(a . d), where a and d are S-expressions`

1.2 Programming Language, Compiler and IDE

Drag is a programming language

Drag is a general purpose, gradually typed, lexically scoped, and multi-paradigm programming language.

Drag displays programs as graphical ASTs. Programmers write programs by directly modifying the structures of the ASTs. The graphical AST suppresses the complex textual details other langauge use, to ensure the simplicity and consistency of syntax. The AST representation moves the programmer's focus from textual details to the structure of the program, thus emphasizing the logic and the computation behind the program. In addition, the AST structure is capable of abstracting the meaning of the programs of any paradigm and visualizing the meanings uniformly.

To ensure semantic safety of programs, the type system of Drag is able to infer and check types. Our design of Drag benefits from both statically and dynamically typed languages. The type system of Drag strictly enforces type rules by default. Drag allows programmers to turn off the type system on a module-by-module basis to achieve the flexibility of dynamic typing.

Drag is a compiler

The AST is a widely used intermediate representation in compilers. Because a complete Drag program is ready for the target-code generation, Drag's compiler skips the two most time-consuming parts of the compilation — lexical scanning and syntax parsing. This feature makes the compilation process light and fast, consequently making further analysis and optimization more efficient.

Drag is an IDE

Our implementation includes a graphic user interface built on Java Swing.

Because Drag's compiler skips scanning and parsing, the IDE of Drag is able to statically analyse programs with lower cost than the IDE of textual programming languages.

1.3 Research Contribution

Compared to classical textual programming languages, the two core advantages of Drag are:

- `AST-based syntax` provides consistency and intuitiveness to make programs readable
- `interactive programming style` provides the efficiency and power for program analysis and optimization.

The first main contribution of this thesis is to use the AST as a graphical and interactive programming language. We originally intended to build a graphical programming language like an electronic circuit. We later discovered that the AST structure enables an interactive programming style that avoids many troubles caused by the parser.

The second contribution of this thesis is to present the specification of Drag and its implementation. Our implementation is minimal but has all the vital parts, so that people can easily extend Drag to explore their ideas.

Chapter 2 Language Specification

> An elegant program is one that is readable the first time by a novice, not one that plays unexpected tricks.
>
> Raphael Finkel

2.1 Syntax and semantics

We designed the syntax of Drag to be easily parsed by both humans and machines.

Many classical programming languages tend to borrow the vocabulary and the grammars from natural languages. This creates two major problems in programming:

- `For humans` - Because the language designers blend in their personal understandings of natural languages, the interpretation of the same piece of program is experience-dependent.
  For example, the API `pop()` for a `LinkedList`: in JAVA the `pop()` removes the first element of the list instance and returns that element as the result of the function, but in the Standard Library of C++, the similar `pop_back()` and `pop_front()` operations only remove the first element and return **`void`**.

  Similar pitfalls exist because that different language designers understand the same words differently.

- `For machines` - Many programming languages unnecessarily imitate the grammars of the natural languages, especially English. This fact creates the difficulty that the machine needs to spend a considerable amount of time in scanning and parsing to understand a program. This unnecessary cost lowers the efficiency of the compilation and makes parsers difficult to write.

Some programming languages, for instance, the LISP family, build their syntax on top of the S-expression, which is essentially an Abstract Syntax Tree. Compared to the natural language imitators, the S-expression provides homoiconicity of programs and data, consistency of syntax and efficiency of parsing.

Learning from the LISP family, we designed the Node-Tree-Symbol model for Drag. The programmers directly build ASTs as the memory objects, not the S-expressions in the string form. Programs of the Node-Tree-Symbol model are as expressive and readable as S-expressions, and more efficient in parsing.

Node-Tree-Symbol

Every programming language has a set of strings as the reserved words. The programmers create the *meanings* of a new string by combining the existing strings using

the syntax rules. The reserved words have meanings by default; the compiler or the interpreter sorts out the meanings of created strings.

Similarly, the fundamental elements, like strings, manipulated in Drag are called **nodes**. **Symbols** stand for the meanings of nodes. A **tree**, the equivalent of a piece of program in a classical programming language, is either a node or a group of connected nodes.

A node implements a symbol

When writing a Drag program, the programmer creates a node by selecting a symbol. A symbol consists of the information of:

- `the name signature` - a list of strings that determines the names of the input and output **ports** of a node
- `the type signature` - a type variable that determines the types of the input and output ports of a node

A node contains a list of ports, where a port is an interface for node connections.

Our prototype includes a small set of primitive symbols on arithmetic operations, list operations, Boolean operations, the conditional expression operation and a few statement operations. We write a symbol as `SymbolName` - ⟨`NameSignature`⟩ ⟨`TypeSignature`⟩.

- `cons` - ⟨cons car cdr⟩ ⟨[a]←(a, [a])⟩
- `cdr` - ⟨cdr lst⟩ ⟨[a]←([a])⟩
- `car` - ⟨car lst⟩ ⟨a←([a])⟩
- `empty-list` - ⟨empty-list⟩ ⟨[a]⟩
- `null?` - ⟨null? lst⟩ ⟨bool←([a])⟩
- `add1` - ⟨add1 operand⟩ ⟨num←(num)⟩
- `sub1` - ⟨sub1 operand⟨ num←(num)⟩
- `*` - ⟨* operand1 operand2⟩ ⟨num←(num,num)⟩
- `zero?` - ⟨zero? operand⟩ ⟨bool←(num)⟩
- `less-than` - ⟨less-than operand1 operand2⟩ ⟨bool←(num, num)⟩
- `greater-than` - ⟨greater-than operand1 operand2⟩ ⟨bool←(num, num)⟩
- `=` - ⟨= operand1 operand2⟩ ⟨bool←(a, a)⟩
- `true` - ⟨true⟩ ⟨bool⟩
- `false` - ⟨false⟩ ⟨bool⟩

- `and` - ⟨and operand1 operand2⟩ ⟨bool←(bool, bool)⟩
- `or` - ⟨or operand1 operand2⟩ ⟨bool←(bool, bool)⟩
- `not` - ⟨not operand⟩ ⟨bool←(bool)⟩
- `cond-expr` - ⟨cond-expr cond consi alter⟩ ⟨a←(bool, a, a)⟩
- `assign` - ⟨assign target source⟩ ⟨stmt←(a,a)⟩
- `print` - ⟨print v⟩ ⟨stmt←(a)⟩
- `stmt-list` - ⟨stmt-list stmts⟩ ⟨stmt←([stmt])⟩

The first string in the name signature determines the name of the symbol; the rest of the strings in the name signature determine the names of the input parameters. The length of the name signature of a symbol corresponds to the number of elements in its type signature. The type system section presents more details about the type signature.

For instance, the `cond-expr` symbol, which contains the name signature ⟨*cond-expr, cond, consi, alter*⟩ and the type signature ⟨*a←(bool, a, a)*⟩, can create a node named cond-expr (which is the first string in its name signature) that returns an *a* type and has three input ports named cond, consi and alter with the types *bool*, *a* and *a* correspondingly.

The output port does not necessarily mean the result value of a function. For an expression-typed node, which is like as a function, the output port is the result value. For a statement-typed node, for example, the assignment operation, the node does not return a value, the output port is used to organize the sequence of execution.

Connecting nodes builds a tree

The programmer may connect nodes when all the following conditions hold:

- The connection must connect an input and an output port.
- The output port and the input port of a connection must be on different nodes.
- A port may be involved in at most one connection.
- A connection must connect two ports with consistent types.

The above rules do not avoid the possibility of circular connections; the procedure of `condensing` removes all circular connections.

A group of connected nodes builds a tree, which is known as a function or a procedure in the classical programming languages. A tree has its own symbol table, which holds the mappings from identifier names to their values, which the lexical scoping section of this thesis addresses.

Condensing a tree creates a symbol

The programmer initiates condensing to create a user-defined symbol based on the current tree. Drag adds the created symbol to a proper **symbol table**, which is discussed in the lexical scoping section.

Condensing starts after the programmer selects a **root node** of the tree. If the output port of the root node involves a connection, Drag disconnects the connection. This action prevents circular connections. The output port of the root node is the output of the created symbol; the input symbols in the symbol table of the tree are the inputs of the created symbol.

A tree is **complete** when all its input ports are connected, in other words, the leaves of the tree are all identifier nodes that do not take input. Only a complete tree is valid to condense.

Here is an example — the less program in Figure 2.1, which is a complete tree with two input symbols, `var1` and `var2`, in its symbol table.

In detail, the output port of `less-than` connects to the `cond` port of the `cond-expr` node, the output port of the upper `var1` connects to the `consi` port of the `cond-expr` node, and the output port of the upper `var2` connects to the `alter` port of the `cond-expr` node. The output port of the lower `var1` connects to the `operand1` port of the `less-than` node, the output port of the lower `var2` connects to the `operand2` port of the `less-than` node.

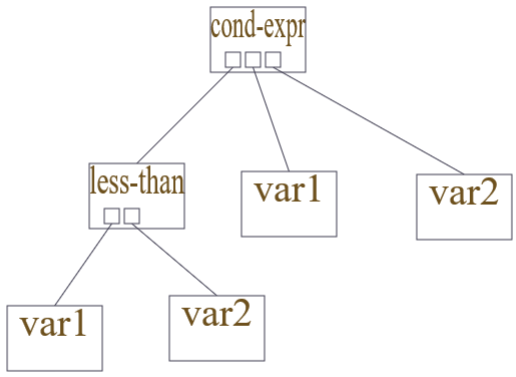

Figure 2.1: the less program

Condensing the tree results in the symbol: ⟨`name` var1 var2⟩ ⟨num←(num, num)⟩, where `name` is specified by the user.

Lexical Scope and Fix Point

Lexical scope, which is also known as static scope and deep binding, was first introduced by the Algol family [1]. The essential idea is to capture the environment of a method when the method is elaborated. A typical implementation is to build a closure that contains the method body and its environment, and to pass the closure when the method is passed.

In regards to scoping, Drag provides the following three operations:

- `add-new-binding` — to introduce a new binding in the current scope, where a binding maps from a variable name to an expression.

- `switch-to-scope` — to ask for a scope name, and then change the current scope to a received name. Changing the scope means to switch the focus to another method.

- `create-fixpoint` — to create a self-reference inside a scope.

In Drag, each tree contains a scope that contains the local mappings from the identifier names to their values. In addition, each tree contains the information of its non-local reference scope, the outer scope tree. Because each tree is bound to a unique scope, we use the term tree and scope interchangeably in this thesis. An identifier is **visible** to a tree if the identifier is in the tree or visible to the non-local reference scope of the tree. The programmers can use only visible identifiers when writing programs, so Drag disallows the use of free variables. The **fix point** operation is needed to implement recursion under the scope rules of Drag. While the programmer is introducing a method, the method is not visible to itself by default, because the identifier of the method is not in its own symbol table and as the method is not elaborated yet, the method is not visible to its non-local referencing environment. The fix point operation creates a link in the symbol table of current method pointing to the location in its non-local referencing environment. Thus, a method with a fix point can refer to itself.

Figure 2.2 shows a tree of a recursive function, where `fact` refers to the tree.

Functions as the first-class objects

A first-class object means that the object can be passed as a parameter to a function and returned as a value by a function [1]. In particular, a first class function can be passed to functions and be returned by functions.

Drag allows all nodes to be the input of other nodes. In particular, Drag uses the **textttwrap** operation to make function passing explicit.

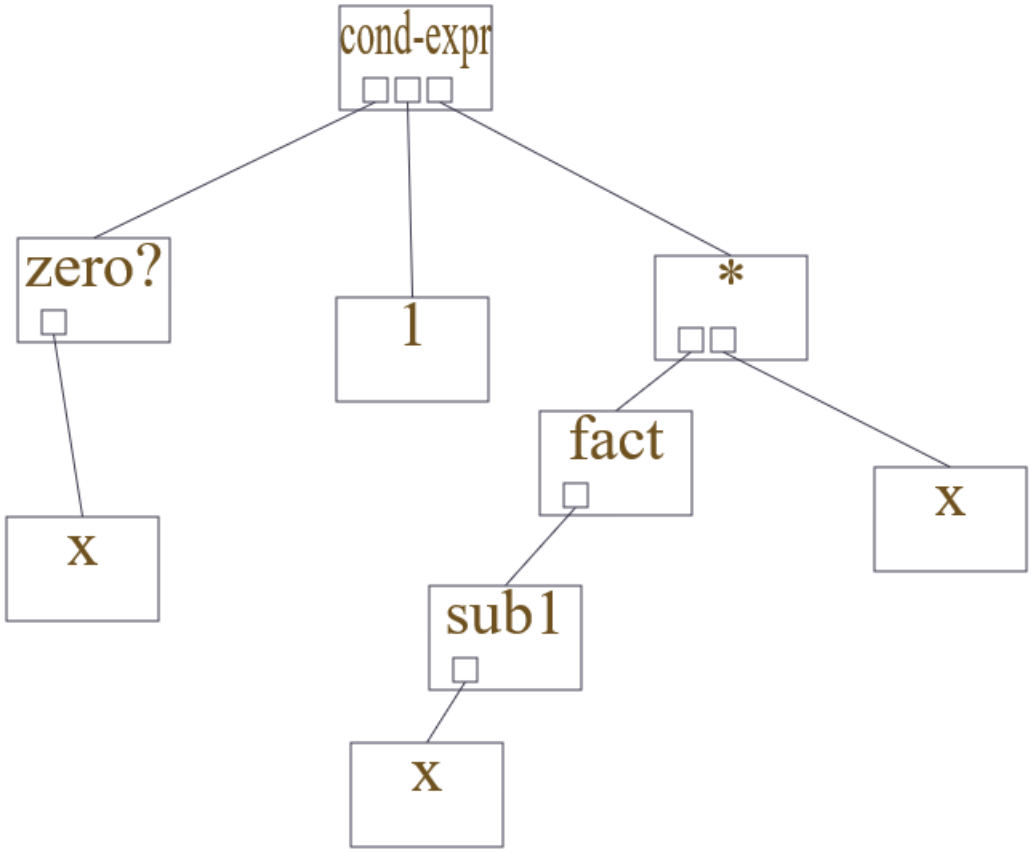

Figure 2.2: factorial, a sample recursive function

The wrap operation distinguishes operators and operands. A node is an operator by default. An operator means that the node takes some inputs and generates an output. The programmer may wrap a node to close all its input ports, so that the node now takes no input and generates *itself* as its output.

For instance, in Figure 2.3, the default `add1` node, a function that expects a *num* input and outputs a *num* value, is an operator; in Figure 2.4, the wrapped `add1` node, an identifier that takes no input and outputs a function value, whose type is ⟨`num` ← `(num)`⟩, is an operand.

Figure 2.3: the default add1

Figure 2.4: the wrapped add1

2.2 Type system

The concept of type system has evolved with the development of the concept of data type in programming languages. Early assembler languages view type as a property of a datum rather instead of a property of a datum container — the fundamental difference was only between addresses and data. Pascal and later languages started to shift the idea of type from what makes sense to machines to what makes sense to programmers [1].

Recently, the term **type system** refers to a broader field of logic, mathematics and philosophy. More than the application to programming languages, some researchers focus on the connection between various lambda-calculi and logic, via the *Curry-Howard correspondence* [4].

In Drag, we implement a simple type system that focuses on type checking and type inference. Programmers do not need to decribe the type signature explicitly — Drag is able to reason out the type of every node (within Drag's current type system). In addition, Drag maintains the tables that contain the mappings from symbols to their types.

Type signature

Here we use `t` to represent a type variable; `box` to represent a dynamic type; `expr` for an expression; `num` for a number type; `str` for a string type; `bool` for a Boolean type; ← for a function, with its output on the left-hand-side and its inputs on the right-hand-side; `[t]` for a list contains the elements of the type `t` homogeneously; and `()` for a tuple; we use English letters, from `a` to `z`, for the expression types that have not been explicitly "narrowed" yet.

Later parts of this sections talk about the details on the function type, the list type and the dynamic type.

⟨*t*⟩ ::= `box`
  | `stmt`
  | ⟨*expr*⟩

⟨*expr*⟩ ::= `num`
  | `str`
  | `bool`
  | ⟨*t*⟩ `<-` ⟨*tuple*⟩
  | `[` ⟨*t*⟩ `]`
  | ⟨*type-id*⟩

⟨*tuple*⟩ ::= `(` ⟨*tuple-inner*⟩ `)`

⟨*tuple-inner*⟩ ::= ⟨*t*⟩ `,` ⟨*tuple-inner*⟩
  | NULL

⟨*type-id*⟩ ::= `a`
  | `b`
  | ...
  | `z`

The construction of a type variable

In order to let programmers write programs interactively, the type inference system must be able to specialize and generalize types. For example, if the programmer disconnects two nodes, then Drag must restore the state before the connections. A more complex situation is that the types of the two nodes may be affected by some other connections.

Thus in addition to unification, which is the typical specialization algorithm for type inference, we slightly complicate the structure of a type variable so that Drag can efficiently generalize the resulting types if necessary: A type variable has the following members:

- `type_was` — what the type was when the node was first created
- `unified_types` — a list that points to the other type variables that is unified with this type variable
- `sig` — a list that points to the related type variables of this type variable. The signature of a function type contains the input types and output type; the

signature of a list type contains the element type; the signature of a type other than function or list is an empty list.

- `boxed?` — a bool that indicates if this type is dynamic

Type rules

We describe type rules in the language of propositional logic. For each inference rule, we write the goal below the line, and the premises above the line. We use the symbol Γ for the general environment of types; Γ_1,Γ_2 for the result of combining two environments; *something*:τ for *something* has the type τ; *cause* $\vdash$ *consi* for *cause* entails *consi*.

The first three rules are for user-defined functions; the later rules are for primitive functions.

rule name	**formula**
abstract	$\dfrac{\Gamma, (x : \alpha) \vdash body : \beta}{\Gamma \vdash (\lambda\ (x)\ body) : \beta \leftarrow (\alpha)}$
apply	$\dfrac{\Gamma \vdash rator : \beta \leftarrow (\alpha) \quad \Gamma \vdash rand : [\alpha]}{\Gamma \vdash (rator\ rand) : \beta}$
var	$\dfrac{\Gamma(var) = \alpha}{\Gamma \vdash var : \alpha}$
num (Axiom)	$\dfrac{}{\Gamma \vdash n : num}$
str (Axiom)	$\dfrac{}{\Gamma \vdash s : str}$

cons	$$\frac{\Gamma \vdash car : \alpha \quad \Gamma \vdash cdr : [\alpha]}{\Gamma \vdash (cons\ car\ cdr) : [\alpha] \leftarrow (\alpha, [\alpha])}$$
car	$$\frac{\Gamma \vdash lst : [\alpha]}{\Gamma \vdash (car\ lst) : \alpha \leftarrow ([\alpha])}$$
cdr	$$\frac{\Gamma \vdash lst : [\alpha]}{\Gamma \vdash (cdr\ lst) : [\alpha] \leftarrow ([\alpha])}\ \text{cdr}$$
empty-list	$$\frac{}{\Gamma \vdash (empty\text{-}list) : [\alpha]}$$
null?	$$\frac{\Gamma \vdash lst : [\alpha]}{\Gamma \vdash (null?\ lst) : bool \leftarrow ([\alpha])}$$
add1	$$\frac{\Gamma \vdash operand : num}{\Gamma \vdash (add1\ operand) : num \leftarrow (num)}$$
$*$	$$\frac{\Gamma \vdash operand1 : num \quad \Gamma \vdash operand2 : num}{\Gamma \vdash (*\ operand1\ operand2) : num \leftarrow (num, num)}$$
less-than	$$\frac{\Gamma \vdash operand1 : num \quad \Gamma \vdash operand2 : num}{\Gamma \vdash (less\text{-}than\ operand1\ operand2) : bool \leftarrow (num, num)}$$
greater-than	$$\frac{\Gamma \vdash operand1 : num \quad \Gamma \vdash operand2 : num}{\Gamma \vdash (greater\text{-}than\ operand1\ operand2) : bool \leftarrow (num, num)}$$

=	$\dfrac{\Gamma \vdash operand1 : num \quad \Gamma \vdash operand2 : num}{\Gamma \vdash (= \ operand1 \ operand2) : bool \leftarrow (num, num)}$
true (Axiom)	$\dfrac{}{\Gamma \vdash true : bool}$
false (Axiom)	$\dfrac{}{\Gamma \vdash false : bool}$
and	$\dfrac{\Gamma \vdash operand1 : bool \quad \Gamma \vdash operand2 : bool}{\Gamma \vdash (and \ operand1 \ operand2) : bool \leftarrow (bool, bool)}$
or	$\dfrac{\Gamma \vdash operand1 : bool \quad \Gamma \vdash operand2 : bool}{\Gamma \vdash (or \ operand1 \ operand2) : bool \leftarrow (bool, bool)}$
not	$\dfrac{\Gamma \vdash operand : bool}{\Gamma \vdash (not \ operand) : bool \leftarrow (bool)}$
cond-expr	$\dfrac{\Gamma \vdash cond : bool \quad \Gamma \vdash consi : \alpha \quad \Gamma \vdash alter : \alpha}{\Gamma \vdash (cond\text{-}expr \ cond \ consi \ alter) : \alpha \leftarrow (bool, \alpha, \alpha)}$
assign	$\dfrac{\Gamma \vdash target : \alpha \quad \Gamma \vdash source : \alpha}{\Gamma \vdash (assign \ target \ source) : stmt \leftarrow (\alpha, \alpha)}$
print	$\dfrac{\Gamma \vdash v : \alpha}{\Gamma \vdash (print \ v) : stmt \leftarrow (\alpha)}$

Type checking and type inference

Drag provides an immediate error message when a programmer violates a type rule. For each operation that modifies the trees, such as creation, connection and removal, Drag checks type consistency and updates the involved type variables by applying the type rules. The programmer may attempt an operation that violates the type rules, like connecting a `bool` expression to the parameter position of the `add1` operation. This operation can not be completed and results in an error message. We expect this curing-the-future-illness approach to be more effective than displaying error messages during compile time and run-time.

The boxed type

Gradual typing mixes static checking and dynamic checking by assigning a **dynamic tag** to a type variable and later making implicit conversions[5].

Via the boxed type, Drag incorporates gradual typing. Here we consider the code-writing time, when type checking and type inference happens, as *static*. We consider the run-time as *dynamic*. By definition, a boxed type in Drag is consistent with all static typing rules and does not participate in type inference.

For example, if function `f` has the type ⟨bool←box⟩ and function `g` has the type ⟨bool←a⟩, the type of `g` after unifying with `f` is ⟨bool←a⟩.

Drag dynamically casts the boxed types to their underlying types and throws run-time errors if the resulting types violate the rules.

Chapter 3 The Drag compiler

A classical compiler, at its simplest, consists of four parts: the lexical scanner, the syntax parser, the semantic analyzer, and the code generator. In Drag, because the source code is the AST, the syntax scanner and the semantic parser are not necessary.

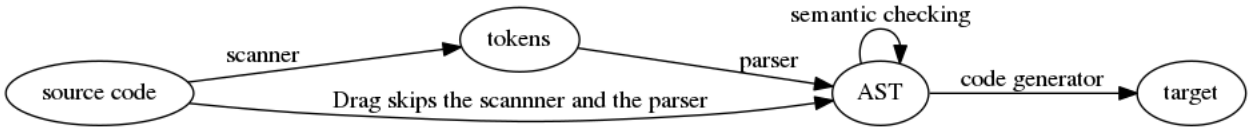

Figure 3.1: classical compiler vs Drag's compiler

To illustrate the distribution of the compilation time, we have tested three C programs under the clang-3.5 compiler and plotted the result in Figure 3.2. In Figure

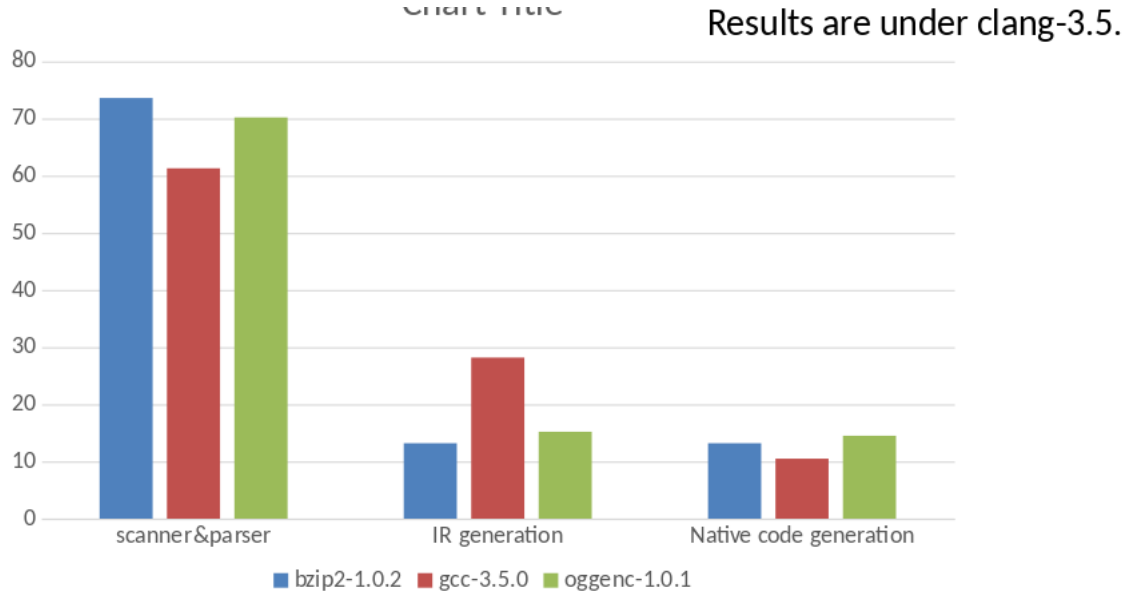

Figure 3.2: An example of the distribution of the compilation time

3.2, the summation of the bars of the same color is 100 percent. The figure indicates that compiler all three programs spends more than 60 percent of the time in the scanning and the parsing. Although this sample is small, both for compilers and programs, a reasonable conjecture is that the scanner and the parser consume most of the compilation time, which Drag avoids.

In addition, because Drag obtains type information at the code-writing time, it saves time during type checking, which contributes to a portion of the IR generation measure of Figure 3.2.

The current implementation of Drag targets on LLVM. The following parts of this chapter include (1) a brief introduction to LLVM and (2) the LLVM code generator of Drag.

3.1 LLVM

LLVM is an open-source library for building compilers. It provides facilities for building both front-ends (from high-level languages to the LLVM instruction set) and back-ends (from the LLVM instruction set to machine code). We choose LLVM because (1) the low level virtual machine provides a balance between run-time efficiency and cross-platform availability (2) there are many tools based on LLVM for program analysis and optimization (3) the modular design of LLVM divides the compiler into many passes, thus providing the simplicity of building optimizers of a compiler, and (4) LLVM provides informative error messages.

The instructions set

The core of LLVM is its instruction set. Some features of the LLVM instruction set include:

- The instructions are register-based, but without machine-specific constraints like the number of physical registers, existence of a pipeline and traps.
- The memory includes a stack, a heap and the global area.
- Values are transferred between the memory and the registers by `load` and `store` operations; objects are allocated on the stack and heap by `alloca` and `malloc` operations, respectively.
- Instructions use three-address code.
- The registers are in static single-assignment-form — a register can be assigned once at most; each instruction implicitly generates a register to hold the result value.
- Strict type rules are enforced on the registers.

Registers

Registers store the result of instructions and organize the sequence of instructions. In the current implementation, a global counter keeps track of the number of registers; Drag records each register's name and type. The register name is a number that is taken from the state of the global counter. The register types in our current implementation include `i32`, `i8*`, and function pointers. Here `i32` stands for a 32-bit register, which is often an integer; `i8*` stands for pointer of an 8-bit register, which is used for a void pointer; a function pointer is in the form of `out(in)*`. For example, `i32(i32)*` points to a function that takes a 32-bit integer and generates

a 32-bit integer. The LLVM instruction set distinguishes local and global registers by appending `%` or `@` to the front of the register name, respectively.

3.2 Code generator

Because the LLVM instruction set is purely imperative, the functional programming parts of Drag need some extra work to generate, compared to the imperative programming part. The extra work includes creating the lists and representing higher-order functions.

List and higher-order functions

To implement a homogeneous and polymorphic list, we use **void pointers**.

First we define the **pair** structure. Each pair has a **head** and a **tail**, where the head is a void pointer to a value and the tail is a pointer to another pair.

```
%struct.pair = type { i8*, %struct.pair* }
```

To build a list, given a head and a tail, Drag converts the head and the tail to void pointers, records the types of the head and the tail, and then generates a pair pointer that constructs the list using a library function that is defined as following. To establish a firm relationship between the primitive Drag function, `cons`, and the LLVM code it generates, we choose cons for the name of our library function as well. We also choose the same names for car and cdr.

```
define %struct.pair* @cons(i8* %car, i8* %cdr) #0 {
  %1 = alloca i8*, align 8
  %2 = alloca i8*, align 8
  %new_pair = alloca %struct.pair*, align 8
  store i8* %car, i8** %1, align 8
  store i8* %cdr, i8** %2, align 8
  %3 = call noalias i8* @malloc(i64 24) #2
  %4 = bitcast i8* %3 to %struct.pair*
  store %struct.pair* %4, %struct.pair** %new_pair, align 8
  %5 = load i8** %1, align 8
  %6 = load %struct.pair** %new_pair, align 8
  %7 = getelementptr inbounds %struct.pair* %6, i32 0, i32 0
  store i8* %5, i8** %7, align 8
  %8 = load i8** %2, align 8
  %9 = load %struct.pair** %new_pair, align 8
  %10 = getelementptr inbounds %struct.pair* %9, i32 0, i32 1
  store i8* %8, i8** %10, align 8
  %11 = load %struct.pair** %new_pair, align 8
  ret %struct.pair* %11
}
```

To fetch from a list, including `car`, which fetches the **head**, and `cdr`, which fetches the **tail**, Drag gets the void pointer of the **head** or the pair pointer of the **tail** then converts the type of the pointer to the recorded type.

The library function `car` is defined as following.

```
define i8* @car(%struct.pair* %aPair) #0 {
  %1 = alloca %struct.pair*, align 8
  store %struct.pair* %aPair, %struct.pair** %1, align 8
  %2 = load %struct.pair** %1, align 8
  %3 = getelementptr inbounds %struct.pair* %2, i32 0, i32 0
  %4 = load i8** %3, align 8
  ret i8* %4
}
```

The library function `cdr` is defined as following.

```
define i8* @cdr(%struct.pair* %aPair) #0 {
  %1 = alloca %struct.pair*, align 8
  store %struct.pair* %aPair, %struct.pair** %1, align 8
  %2 = load %struct.pair** %1, align 8
  %3 = getelementptr inbounds %struct.pair* %2, i32 0, i32 1
  %4 = load i8** %3, align 8
  ret i8* %4
}
```

LLVM provides function pointers. We use function pointers to implement higher order functions.

Code generation

The code generator starts at the root node of the tree, recursively visits the children of the node, aborts if it finds a unconnected node, and generates a string of LLVM instructions as it proceeds. Each node has two states that affect the result of the code generation: `userDefinedP` and `wrappedP`.

userDefinedP

Drag has a set of primitive symbols, which contain the methods for code generation. The nodes of such symbols directly translate into strings. If the code generator reaches a node of a non-primitive symbol, it recurses on the definition of that symbol from the symbol table. The generated string from the symbol is appended to the front of the string generated from the current tree.

wrappedP

As discussed in the language specification chapter, a wrapped node acts as an identifier. The generated string of a wrapped node is the identifier name of the node; the generated string of an unwrapped node follows the above-mentioned pattern — the code generator recursively visits the children of the node.

Chapter 4 IDE

4.1 User interface

We wrote the graphic user interface of Drag in Java Swing. Here we present the draft and the actual view of the user interface in Figures 4.1 and 4.2 respectively.

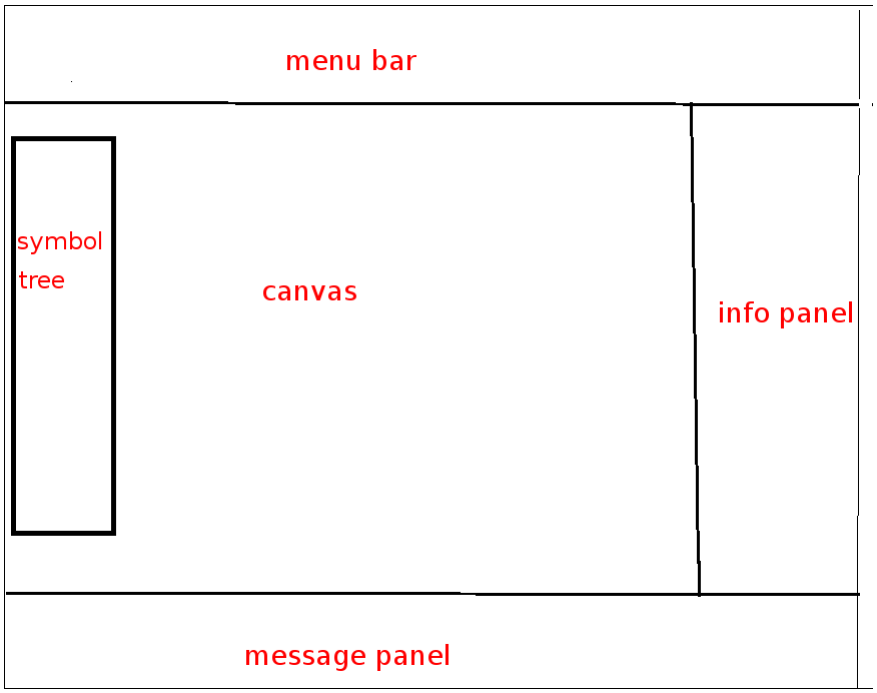

Figure 4.1: the draft of the user interface

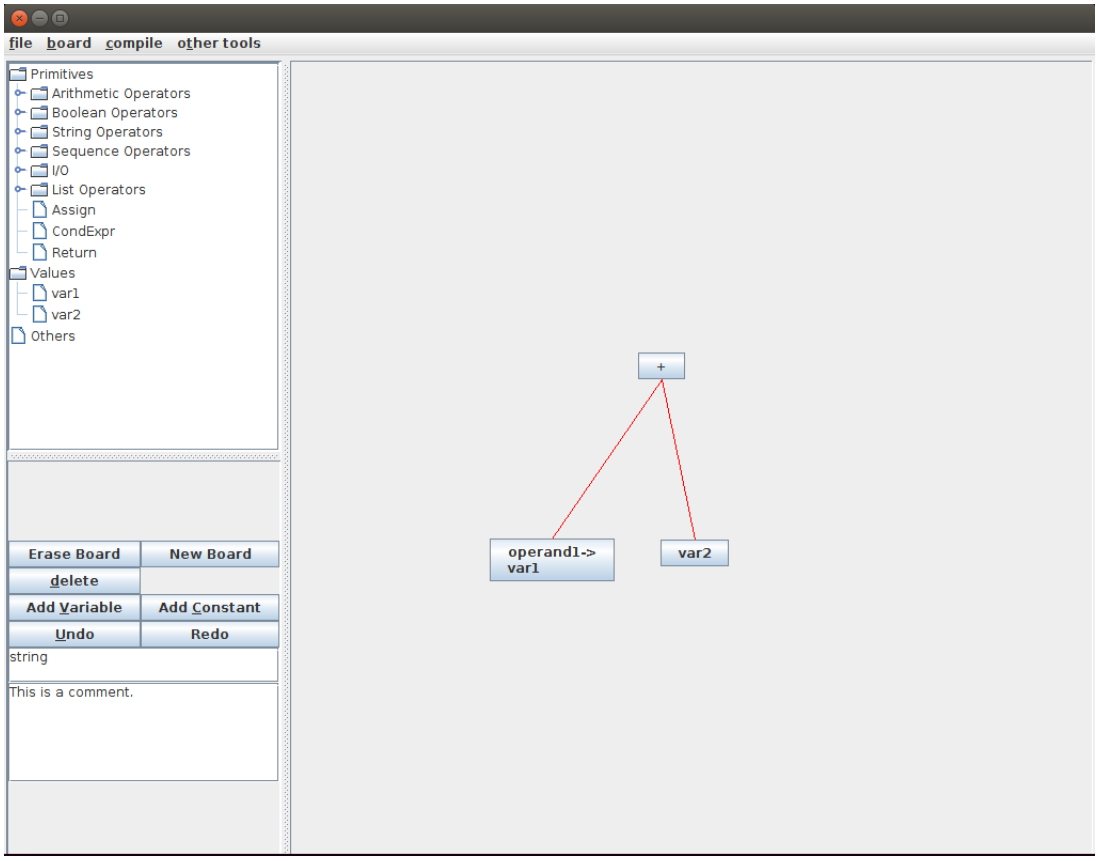

Figure 4.2: the view of the user interface

Java Swing

Java Swing is built on the Java Abstract Window Toolkit (AWT). Swing components are platform-independent and support a pluggable look and feel. The component architecture is called the Model-Delegate model, simplified from the Model-View-Controller model.

The Java AWT provides an interface between a Java program and the local machine. Each graphical API in AWT corresponds to a peer API in the operating system. Java AWT only provides a limited number of graphical APIs. Based on AWT, Java Swing extends the library. Consequently, the run-time efficiency of a Swing GUI is slightly lower than an AWT GUI, which is the cost of its powerful and ample APIs.

Components and containers

A Swing container is a component that holds a group of components. The containment hierarchy starts at a single top-level container. The abstract class JComponent is the superclass of all components, except for the four top-level container components: JFrame, JApplet, JWindow and JDialog. The JComponent class inherits the Component class in the AWT package and provides the painting infrastructure and methods to handle keyboard presses. The `add()` API lets the Swing programmer to add another JComponent into the hierarchy.

Frames and panels

The JFrame class provides a top-level titled window with a few basic facilities to maximize, minimize and close. The JPanel class provides a window to hold other components.

Some components

JLabel displays text. JTextField and JTextArea display editable text. JTextField is single-lined and JTextArea is multi-lined. JButton is a clickable component. It reacts to keyboard and mouse events. JTree is displays a tree structure.

Events

A Java Swing GUI is event-driven. Events enable interaction between the user and the GUI. Java Swing handles events in an ActionListener model. The Swing programmer binds a JComponent to an event-action by calling `addActionListener()`.

Layout managers

LayoutManager is an interface in the Java AWT package, with several implementations. A LayoutManager is used to locate and tailor the components in a container

component, and a LayoutManager may overwrite the position and size of the components. The Swing programmer binds components to a LayoutManager by calling `setLayout()`.

Canvas

The canvas is where the Drag builds programs. Drag displays a node as a block of white background with the node name and the node ports shown. A connection between nodes is displayed as a white line.

Drag tries to display the most frequently used information during programming in the canvas — the visible symbols and the structure of the focused function. This information displays at the center of the screen, so that the users can get information almost without any cost. The other panels display less frequent information, such as type signatures and editing history, so the user is able to get this information at a relatively low cost — a shift of a glimpse. Information that is not likely to be useful, like the implementation details of some other functions, needs more effort to retrieve — typing in the name and clicking the `retrieve` button.

The user can move a node by click-and-drag with the left mouse button. The user can activate the context menu by right clicking on a node. The context menu provides several operations: delete the node, make the node a root node, wrap the node, add input to the symbol of the node and box the node. The user can disconnect two connected nodes by right clicking the line between the nodes.

The symbol tree display shows the tree containing the visible symbols, where a symbol is the child node of its nonlocal reference environment. Left clicking on a symbol that has children expands/folds the branch. Left clicking on a symbol that does not have a child creates a node of that symbol. Right clicking on a symbol actives a context menu that provides the operations of creating a fixed point of the symbol and switching to the scope of the symbol.

We implement the canvas by extending the `JPanel` class of Java Swing and the symbol tree by extending the `Tree` class of Java Swing. We implement the mouse reaction by extending the `MouseInputAdapter` and `MouseMotionListener` classes of Java AWT.

Menu bar

The menu bar contains a textbox, a button to reset the user interface, a button to retrieve a target symbol, a button to create a variable, a button to create a constant, a button to run the program, and a button to generate target code.

The textbox is for general input, for example, a variable name. When the user clicks on the `retrieveSymbol` button, Drag starts to search for the symbol whose name matches the string in the textbox and switches to the scope of the symbol if found. When the user clicks on the `newVar` button, Drag tries to create a new variable whose name is the string in the textbox in the current scope and then adds the created symbol to the symbol tree if successful.

Info panel

The info bar contains a textbox for the type signature. The type signature, the string in the textbox, updates when the user places the mouse over different nodes or symbols on the canvas.

Message panel

The message panel contains feedback, such as switching scopes, important type signatures, computation results and error messages. The background color of the message panel changes when the feedback is nontrivial. We categorize the nontrivial feedbacks into two groups, negative and positive. The background color turns to red for negative feedback, including a connection that violates the type consistency, searching for a symbol that does not exist and creating a duplicated symbol; the background turns to green for positive feedback, including the computation result or the generated code.

Chapter 5 Conclusion and further work

5.1 Conclusion

This thesis introduces the Drag programming language, which allows the programmer to build abstract syntax trees graphically and interactively. The purpose of Drag is to make programming simple and accessible.

We developed the **node-symbol-tree** model for the syntax and semantics of Drag. Drag tries to express information uniformly to gain consistency and readability. Drag is a multi-paradigm programming language, and our current implementation focuses on functional programming. Drag integrates both static type checking and dynamic type checking. The interactive style moves static type checking from compile time to writing-code time, which makes debugging easier. The type system of Drag is able to infer types.

Drag has a lighter and more efficient compiler than the classical programming languages.

The graphical user interface of Drag builds on Java Swing. The idea of the user interface design is to help the user focus on important elements during programming and neglect the less important elements.

5.2 Direction of future work

Type system

We would like to introduce a few more powerful features to the type system of Drag. We have implemented some ideas of recursive types and polymorphism but have not formalized the ideas yet. We are thinking about adding the feature of dependent type so that we can extend Drag to the realm of theorem proving.

User interface

We notice the low efficiency of editing in Drag. Compared to the classical programming languages, Drag does not make a good use of the keyboard, which provides editing efficiency for professionals. Adding keyboard short cuts is one option. We would probably learn from the popular editors, like Vim, and integrate their shorcuts and editing styles into Drag.

Support of proof systems

The Curry-Howard correspondence reveals that there is a strong connection between computer programs and mathematical proofs [4]. We are considering enriching Drag for representing a the mathematical proof system. We believe the graphical and interactive style of Drag can make proof theory more accessible to programmers.

Bibliography

[1] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] Charles N. Fischer, Ronald K. Cytron, and Richard J. LeBlanc. *Crafting A Compiler*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, April 1960.

[4] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[5] Jeremy Siek and Walid Taha. Gradual typing for objects. pages 2–27, 2007.

Vita

Weixi Ma received the degrees of Bachelor of Electrical Engineering from University of Kentucky and China University of Mining and Technology in May 2013.
Attended the Master program of Computer Science at University of Kentucky in August 2013.
Attended the PhD program of Computer Science at Indiana University in August 2015.
Served as research assistant at University of Kentucky from January 2012 to May 2013.
Served as research assistant at University of Kentucky from May 2014 to May 2015.
Served as associate instructor at Indiana University from August 2015 to present.
This thesis was defended in April 2015 at the University of Kentucky.