Reading Report on PFPL

Theories and Formulations of Computation

Zhibo Chen
University of California, San Diego
zhc159@ucsd.edu

Abstract

In this report, I will summarize the main content of PFPL (Practical Foundations of Programming Languages) and the perspective that Professor Robert Harper has towards the computation and programming languages.

1. Introduction

In section 2, I will present how the logic system that the author has set up in the first part of his book. In section 3, I will present a system of specifying computation of terminating functions on natural numbers. I will prove the type safety of that system. In section 4, I will discuss a few applications of the theories and Harper's formulation.

2. Abstract Binding Tree

Abstract binding trees are the crucial notion in Harper's PFPL book. Each program is represented mathematically as an abstract binding tree.

Unlike the string-based formal systems that we see in proof theories and logic, where we first define vocabularies and grammar, we begin with a well-defined abstract binding tree, which has a desirable structure that make structural induction very easy to perform.

2.1 Definitions

2.1.1 Intuition

The syntax of abstract binding tree (or a program) is as follows:

$$expression := operator(bindings.expression, \dots)$$

where operator is a constant function that are defined by the syntax, and in the parenthesis is a list arguments to that function.

2.1.2 Formal Definitions

2.1.2.1 Sorts Sorts us the fundamental concept of distinguishing between different classes or categories of syntax trees. Sorts are not **types** in a conventional sense, since could be represented by abstract binding trees. For example, usually in programming language, there is the sort of `Exp` of expressions and the sort `Typ` of the types. We use s to refer to a sort and $\mathcal{S}$ to refer to the set of sorts under consideration. In this case, $\mathcal{S} = \{\texttt{Exp}, \texttt{Typ}\}$.

2.1.2.2 Variables Variables is a literal expression of a certain sort. We usually use x_1, x_2, ..., or x_n, or y, or z to refer to variables. We write $\mathcal{X}_s$ to indicate the set of variables of sort s.

2.1.2.3 Abstractor and Valence An abstractor has the form $x_1, \dots, x_k.a$ where $x_1, \dots, x_k$ are variables of sorts $s_1, \dots, s_k$ respectively and a is an abstract binding tree of sort s, we say that the abstractor $x_1, \dots, x_k.a$ has valence $v = s_1, \dots, s_k.s$, and that variables $x_1, \cdots, x_k$ are bound in abstract binding tree a.

2.1.2.4 Operators and Arities In an abstract syntax tree, if an operator accepts arguments of sort $s_1, s_2, \dots, s_n$ and the result of applying an operator to a list of arguments of correct sort is an abstract syntax tree of sort s, we say that the operator has arity $(s_1, s_2, \dots, s_n)s$.

In abstract binding tree, if an operator accepts arguments (which are abstractors) of valence $v_1, \dots, v_k$ and the result of applying an operator to a list of abstractors of correct valences is an abstract binding tree of sort s, we say that the operator has generalized arity $(v_1, \dots, v_n)s$, where v_i has the form $(s_1, \dots, s_k)s_i$.

2.1.2.5 Abstract Binding Trees Fix a set $\mathcal{S}$ of sorts, a family $\mathcal{O}$ of sets of operators indexed by their generalized arities, for a family of sets of variables $\mathcal{X} = \{X_s\}_{s \in \mathcal{S}}$, the family of abstract binding trees indexed by their sort $\mathcal{B}[\mathcal{X}] = \{\mathcal{B}[\mathcal{X}]_s\}_{s \in \mathcal{S}}$ has the following two kinds of elements,

(a) **Variables** if x is an variable of sort s, then x is an abstract syntax tree of sort s. That is, if $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.

(b) **Operators with Arguments** if o is an operator of generalized arity $(v_1, \ldots, v_k)s$ and $\vec{x}_1.a_1, \ldots, \vec{x}_k.a_k$ are abstractors of correct valence, then $o(\vec{x}_1.a_1; \ldots; \vec{x}_k.a_k)$ is an abstract syntax tree of sort s.

That is, (quoted from the book) "For each operator o of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)\, s$, if $a_1 \in \mathcal{B}\left[\mathcal{X}, \vec{x}_1\right]_{s_1}, \ldots,$ and $a_n \in \mathcal{B}\left[\mathcal{X}, \vec{x}_n\right] s_n$ then $o\left(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n\right) \in \mathcal{B}[\mathcal{X}]_s$"

This definition are not complete since abstract binding trees are intended to be "identified modulo alpha renaming", I would not go into the details of the definition, but the idea is that alpha-equivalent abstract binding trees are identical.

2.1.3 Structural Induction

Since abstract binding trees has only two kinds of members, we could use structural induction to prove that all ABTs of sort s has a property P if

(a) **Variables** all variables have the property P

(b) **Operators with Arguments** if all arguments have property P, then the operator has property P.

We could prove that **all** ABTs of sort s has property P by showing both (a) and (b) above.

I will leave out the technical details. Interested readers can refer to page 8 of the book.

2.2 Hypothetical Judgments and Rules Induction

Hypothetical judgments and rules induction are crucial to the design and implementation of programming languages. In fact, almost all properties of programming languages and type systems could be expressed by the rules in the form of hypothetical judgments and their properties could be proved using rules induction.

2.2.1 Definitions

2.2.1.1 Judgment A judgment is a statement that states the property of some object, usually, abstract binding tree. Sometimes, a judgment involves some other objects and is more similar to the concept of a relation. For instance, an expression has a certain form with respect to some type environment, a tree has a certain height, an expression takes certain number of steps to reduce to another expression. When another object is involved in expressing the property of the central object (usually an ABT), the judgment will be annotated with additional information to express that property. Below is a few examples of judgments.

Judgment	Meaning
2 `even`	Number 2 is an even number
t `has height` h	Tree t has height h
$\Gamma \vdash e : \tau$	In the type environment Γ, expression e has type τ
$e \to^n e'$	Expression e reduces to expression e' in n steps

2.2.1.2 Inductive Definition and Rules An inductive definition defines property of certain abstract binding tree by giving a list of rules. For example, given operators, `empty` of arity `Exp`, and `node` of arity `(Exp, Exp) Exp`, the following definition defines the height of a tree.

$$\frac{}{\text{empty } \texttt{has height } 0} \tag{R1}$$

$$\frac{t_1 \texttt{ has height } n_1 \qquad t_2 \texttt{ has height } n_2}{\text{node}(t_1; t_2) \texttt{ has height } max(n_1, n_2) + 1} \tag{R2}$$

2.2.1.3 Derivation A derivation is a step-by-step proof of a judgment from known premises. For example, to derive that the height of $\text{node}(\text{empty}, \text{node}(\text{empty}, \text{empty}))$ is 2, we could construct the following derivation.

$$\dfrac{\dfrac{}{\text{empty } \texttt{has height } 0}\,\text{R1} \qquad \dfrac{\dfrac{}{\text{empty } \texttt{has height } 0}\,\text{R1} \qquad \dfrac{}{\text{empty } \texttt{has height } 0}\,\text{R1}}{\text{node(empty, empty) } \texttt{has height } 0}\,\text{R2}}{\text{node(empty, node(empty, empty)) } \texttt{has height } 0}\,\text{R2}$$

2.2.1.4 Relationship of Inductive Definition, Rules and Derivation On page 14, Harper writes:

> "A collection of rules is considered to define the strongest judgment form that is closed under, or respects, those rules. To be closed under the rules simply means that the rules are sufficient to show the validity of a judgment: J holds if there is a way to obtain it using the given rules. To be the strongest judgment form closed under the rules means that the rules are also necessary: J holds only if there is a way to obtain it by applying the rules. The sufficiency of the rules means that we may show that J holds by deriving it by composing rules. Their necessity means that we may reason about it using rule induction."

As I interpret it, the essential idea is that a set of rules is really a mathematical definition! The sufficiency condition states that, if we are able to derive that an ABT a has a certain property P, then we know that a has property P, The necessity condition state that, if we know that an ABT a has a certain property P, then there exists a derivation tree with a has P as its conclusion.

For example, rules R1 and R2 defines the height of a tree. That is, for tree $\text{node}(\text{empty}, \text{node}(\text{empty}, \text{empty}))$, if we present a derivation that its height is 2, then we know that the tree's height is 2. And if we know that the tree's height is is 2, we must be able to derive it using R1 and R2.

This property gives rise to the ability to do rule induction over inductive definitions.

For example, we could use rules induction to prove the following proposition:

Lemma. *If $node(t_1, t_2)$ has height n, then either t_1 has height $n-1$ or t_2 has height $n-1$.*

Proof. Assume that $node(t_1, t_2)$ has height n, then we either used R1 or R2 to derive this. Since the tree is of the form $node(-)$, R1 doesn't apply. So we must have derived it using

R2. Then, the inductive hypothesis tells the maximum of heights of t_1 and t_2 is $n-1$, we have either t_1 has height $n-1$ or t_2 has height $n-1$. □

2.2.1.5 Hypothetical Judgments Hypothetical judgments are judgments of the form $\Gamma \vdash J$, which asserts that we could derive J from propositions in Γ.

The judgment could appear in the premise and conclusion just as a regular judgment.

3. Characteristics of Programming Language, case study with System T

System T is a system with primitive recursion. All expressions are guaranteed to terminate. In explaining the system, I will explain the what Harper calls "statics", "dynamics", and type safety as a result of these notions.

Roughly statics take into consideration the type safety and well-formedness property. Dynamics is a specification of the evaluation, that is "how the program computes" and "what the program computes to". Type safety regards the relationship between the statics and dynamics. That is, a well typed program could evaluate and the evaluation step will produce a well typed program.

3.1 Definitions

3.1.1 Concrete and Abstract Syntax

We've introduced the abstract binding way as a way to express program. For example, we use $\mathrm{let}(a_1; x.a_2)$ to express a let binding, use $\mathrm{plus}(n_1; n_2)$ to express addition, and so on. We will call that syntax abstract syntax. Corresponding to each abstract syntax is a concrete syntax that is more human readable. For instance, we would write $\mathrm{let}\, x\, \mathrm{be}\, a_1\, \mathrm{in}\, a_2$ instead of $\mathrm{let}(a_1; x.a_2)$ and $n_1 + n_2$ instead of $\mathrm{plus}(n_1; n_2)$ for improved readability.

3.1.2 Syntax of T

There are two sorts of abstract binding trees, `Typ` τ for types and `Exp` e for expressions.

Abstract binding trees of sort τ has the following operators:

$\tau ::=$

Abstract Syntax	Concrete Syntax	Explanation
nat	$\mathbb{N}$	Natural numbers
$\mathrm{arr}(\tau_1; \tau_2)$	$\tau_1 \to \tau_2$	Functions

The syntax defines what it is to be a abstract binding tree of sort `Typ` or τ. Note here that arr has arity $(\mathtt{Typ}, \mathtt{Typ})\mathtt{Typ}$ For example, $\mathrm{arr}(\mathrm{arr}(\mathrm{nat}; \mathrm{nat}); \mathrm{nat})$ is while $\mathrm{nat}(\mathrm{arr}; \mathrm{nat})$ isn't. And the concrete syntax tells us that we could write $\mathrm{arr}(\mathrm{arr}(\mathrm{nat}; \mathrm{nat}); \mathrm{nat})$ conveniently as $(\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat}$.

Abstract binding trees of sort `Exp` e has the following members:

$e ::=$

Abstract Syntax	Concrete Syntax	Explanation
x	x	Variable
z	z	Zero
$\mathrm{s}(e)$	$\mathrm{s}(e)$	Successor
$\mathrm{rec}\{e_0; x, y.e_1\}(e)$	$\mathrm{rec}\, e\{\mathrm{z} \hookrightarrow e_0 \mid \mathrm{s}(x)\mathtt{with} y \hookrightarrow e_1\}$	Recursion
$\mathrm{lam}\{\tau\}(x.e)$	$\lambda(x:\tau)e$	Abstraction
$\mathrm{ap}(e_1; e_2)$	$e_1(e_2)$	Application

For the following discussion, I will not distinguish between the term "program", "expression", "abstract binding tree" and use them interchangeably, since in **T** , a program is an expression which is represented as an abstract binding tree.

3.1.3 Informal Semantics of T

Semantics concern the meaning of a program. System **T** could be used to define terminating function on natural numbers. We have two forms of numbers, z which represents the number 0, and $\mathrm{s}(e)$ represents the successor of the number represented by e. That is, $\mathrm{s}(\mathrm{z})$ represents the number 1, $\mathrm{s}(\mathrm{s}(\mathrm{z}))$ represents then number 2, and so on.

$\mathrm{s}(-)$ and z are called the introduction forms of the natural number. They are used to construct an element of type $\mathbb{N}$, whose meaning will become clear when we introduce the statics of **T** . Corresponding to the introduction forms there are elimination forms of $\mathbb{N}$, which specifies what can we do with a natural number. In HoTT (Homotopy Type Theory), and Martin-Löf's type theory, this is called the induction principle of $\mathbb{N}$, which specifies that in order to a property P of any natural number n, it is sufficient to prove the base case and the inductive steps.

Operator $\mathrm{rec}\{e_0; x, y.e_1\}$ of arity $(\mathtt{Exp})\mathtt{Exp}$ is the elimination form. Informally, any argument expression of type $\mathbb{N}$, it checks whether it is zero or not. If the argument is zero, the recursor will evaluate to e_0. If the argument is not zero, the argument must be of the form $\mathrm{s}(e')$, and then the recursor will evaluate to e_1, with x equal to e' and y equal to $\mathrm{rec}\{e_0; x, y.e_1\}(e')$, the result of applying the recursor of the predecessor of the original argument.

Once we understand the informal semantics, we could use system **T** to define a number of commonly seen mathematical expressions:

3.1.3.1 Double a number We could use **T** to define the function which doubles its argument.

$$\mathrm{double} = \lambda(n:\mathbb{N})\, \mathrm{rec}\, n\{\mathrm{z} \hookrightarrow \mathrm{z} \mid \mathrm{s}(x)\mathtt{with} y \hookrightarrow \mathrm{s}(\mathrm{s}(y))\}$$

This definition says, on any input argument n, invoke the recursor, if n is z, then return z since $2 \cdot 0 = 0$. If n is of form $\mathrm{s}(e')$, then return $\mathrm{double}(e') + 2$, which is $\mathrm{s}(\mathrm{s}(y))$.

3.1.3.2 Add two numbers We could use **T** to define the function which adds two numbers

$$\text{add} = \lambda(n_1 : \mathbb{N})\lambda(n_2 : \mathbb{N})\,\text{rec}\,n_1\{\text{z} \hookrightarrow n_2 \mid \text{s}(x)\texttt{with}y \hookrightarrow \text{s}(y)\}$$

This definition says, on any input argument n_1, n_2, we do a case analysis on n_1. If n_1 is zero, then $n_1 + n_2 = 0 + n_2 = n_2$, and we just return n_2. If n_1 is the form of $s(x)$, then $n_1 + n_2 = (x+1) + n_2 = (x + n_2) + 1$, and since $x + n_2$ is just the result of the recursive call $add(x)(n_2) = y$, we just return $s(y)$.

3.2 Statics

As quoted from page 33 of PFPL

> "Most programming languages exhibit a phase distinction between the static and dynamic phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be safe exactly when well-formed programs are well-behaved when executed."

In simple words, statics specify the typing rules for a programming language, and dynamics specify the execution and meaning (or semantics) of a language.

Typing is usually defined by the following judgment form.

Judgment	Meaning
$\Gamma \vdash e : \tau$	In the type environment Γ, expression e has type τ

In System **T** , the statics is defined by the following rules.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \text{(VAR-REFL)}$$

$$\frac{}{\Gamma \vdash \text{z} : \mathbb{N}} \quad \text{(NAT-INTRO-Z)}$$

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \text{s}(e) : \mathbb{N}} \quad \text{(NAT-INTRO-S)}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathbb{N}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \text{rec}\{e_0; x, y.e_1\}(e) : \tau} \quad \text{(NAT-ELIM)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1)e : \tau_1 \to \tau_2} \quad \text{(ARROW-INTRO)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2} \quad \text{(ARROW-ELIM)}$$

The symbol Γ, usually called type-checking environment, type environment, or simply environment, often is a sequence of type judgments, which assigns types to variables. Examples of Γ are

(i) $\varnothing$

(ii) $x : \mathbb{N}$

(iii) $y : \mathbb{N} \to \mathbb{N}$

(iv) $x : \mathbb{N}, y : \mathbb{N} \to \mathbb{N}$

(v) $x : \mathbb{N}, y : \mathbb{N} \to \mathbb{N}, x : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$

Rule VAR-REFL says that if in the environment variable x has type τ, then variable x has type τ.

Rule NAT-INTRO-Z states that operator z with no arguments is an expression of type $\mathbb{N}$.

Rule NAT-INTRO-S states that if expression e has type $\mathbb{N}$, then successor of e, $\text{s}(e)$ has type $\mathbb{N}$.

Rule NAT-ELIM states that if e is a natural number, and e_0 and e_1 with arguments applied has the same type, then the elimination form has that type.

Rule ARROW-INTRO states that if e has type τ_2 when the argument has type τ_1, then the lambda abstraction has type $\tau_1 \to \tau_2$.

Rule ARROW-ELIM states that an abstraction applied to correct arguments will yield expression of the correct type as result.

3.3 Dynamics

Dynamics concern how the program executes. Harper identified four forms of specifying dynamics. They are structural dynamics by defining how program transforms itself in each step, contextual dynamics where program's evaluation order is specified by presence of an evaluation context, equational dynamics where a equivalence relation is defined on the set of all programs, and evaluation dynamics where an evaluation relation between program terms and values are defined.

These formulation are almost equivalent to each other but for the purpose of subsequent type safety proof, I will present the structural dynamics of **T** .

Structural dynamics consist of two forms of judgments.

Judgment	Meaning
$e\,\text{val}$	Expression e is a value
$e \mapsto e'$	Expression e reduces to expression e' in n steps

A value is the final step in the evaluation process. For instance, $\text{s}(\text{s}(\text{z}))$, the number 2 is a value while a reducible expression, an expression that could be computed, e.g. $(\lambda(x : \mathbb{N})s(x))(\text{s}(\text{z}))$ is not a value. The reason is that the let expression and function binding could be further evaluated and produce $\text{s}(\text{s}(\text{z}))$ as a result.

3.3.0.1 Lazy vs. Eager semantics There are two flavors of evaluation strategies for System **T** . They differ in treating reducible expression that are subexpressions of the expression in question. For example, although both strategies agree that $(\lambda(x : \mathbb{N})\,\text{s}(x))(\text{s}(\text{z}))$ is not a value and could be evaluated to $\text{s}(\text{s}(\text{z}))$, they disagree on whether $\text{s}((\lambda(x : \mathbb{N})\,\text{s}(x))(\text{s}(\text{z})))$ is a value. The eager strategy will reduce that expression but the lazy strategy prefers not to reduce

that expression. For now I will use the lazy evaluation formulation.

3.3.0.2 Capture-free substitution Essential to the idea of evaluation is a substitution. We write the operation on the abstract binding tree "substitute the free occurrences of variable x in e by the expression e'" as $[e'/x]e$.

They are defined inductively as follows:

(i) if e is a variable x', then $[e'/x]x'$ is e' if $x = x'$ and $[e'/x]x'$ remains x' if $x \neq x'$.

(ii) if e is an operator $o(\vec{x}_1.e_1, \ldots, \vec{x}_n.e_n)$, then $[e'/x]e$ is $o([e'/x]\vec{x}_1.e_1, \ldots, [e'/x]\vec{x}_n.e_n)$,

where

$[e'/x]\vec{x}_i.e_i$ is $\vec{x}_i.[e'/x]e_i$ if $x \notin \vec{x}_i$, and

$[e'/x]\vec{x}_i.e_i$ remains $\vec{x}_i.e_i$ if $x \in \vec{x}_i$.

To avoid capture of free variable in e' by the abstraction $\vec{x}_i$, we require that no free variable of e', $FV(e')$ occurs in $\vec{x}_i$. That is, we requires $\vec{x}_i \notin e'$. We may implement automatic renaming of variables should a naming conflict occurs.

Note that we write $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$ to mean the simultaneous substitution of variables. The expression is equivalent to $[e_1/x_1] \ldots [e_n/x_n]e$

I will now present the rules of evaluation in lazy semantics.

$$\frac{}{\mathrm{z\ val}} \quad \text{(Z-VAL)}$$

$$\frac{}{\mathrm{s}(e)\ \mathrm{val}} \quad \text{(S-VAL)}$$

$$\frac{}{\lambda(x:\tau)e\ \mathrm{val}} \quad \text{(LAM-VAL)}$$

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \quad \text{(APP-FUNC)}$$

$$\frac{}{(\lambda(x:\tau)e)(e_2) \mapsto [e_2/x]e} \quad \text{(APP-BETA)}$$

$$\frac{e \mapsto e'}{\mathrm{rec}\{e_0; x, y.e_1\}(e) \mapsto \mathrm{rec}\{e_0; x, y.e_1\}(e')} \quad \text{(REC-ARG)}$$

$$\frac{}{\mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{z}) \mapsto e_0} \quad \text{(REC-Z)}$$

$$\frac{}{\mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{s}(e)) \mapsto [e, \mathrm{rec}\{e_0; x, y.e_1\}(e)/x, y]e_1} \quad \text{(REC-S)}$$

The rules Z-VAL, S-VAL, and LAM-VAL tell us that zero, successor of an expression and lambda abstraction are values.

The rules APP-FUNC and APP-BETA tell us that to evaluate an application, we first evaluate the expression in the function position. Once evaluated, the function position should be a lambda abstraction (if the program is well typed, but we will prove this later) then the result is just to substitute the argument into the body of the function. Notice, under strict semantics we would also need to make sure the argument is fully evaluated, but in the lazy semantics, we can just substitute the argument in.

The rules REC-ARG, REC-Z and REC-S tells us how to evaluate an recursor. Namely, first evaluate the argument, which is a natural number. If the argument is zero, then we take a step and transition to e_0. If the argument is the successor of another argument, we evaluate e_1 with x being the predecessor, and y being the result of recursive call on the predecessor, which is represented as $\mathrm{rec}\{e_0; x, y.e_1\}(e)$

3.4 Example of Evaluation

Once we have the rules, we could see how functions are evaluated. Recall the add function we've defined previously

$$\mathrm{add} = \lambda(n_1 : \mathbb{N})\lambda(n_2 : \mathbb{N})\,\mathrm{rec}\, n_1\{\mathrm{z} \hookrightarrow n_2 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\}$$

If we were to compute $2 + 3$, that is, we would do the following: (We will abbreviate $\mathrm{s}(\mathrm{s}(\mathrm{z})))$ as 3 and so on as needed).

$$\begin{aligned}
&\mathrm{add}(3)(2)\\
&= (\lambda(n_1 : \mathbb{N})\lambda(n_2 : \mathbb{N})\\
&\qquad \mathrm{rec}\, n_1\{\mathrm{z} \hookrightarrow n_2 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\})(\mathrm{s}(\mathrm{s}(\mathrm{z})))(3)\\
&\mapsto (\lambda(n_2 : \mathbb{N})\\
&\qquad \mathrm{rec}(\mathrm{s}(\mathrm{s}(\mathrm{z}))\{\mathrm{z} \hookrightarrow n_2 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\})(3)\\
&\qquad\qquad \text{(by APP-BETA)}\\
&\mapsto \mathrm{rec}(\mathrm{s}(\mathrm{s}(\mathrm{z}))\{\mathrm{z} \hookrightarrow 3 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\}\\
&\qquad\qquad \text{(by APP-BETA)}\\
&\mapsto s(\mathrm{rec}(\mathrm{s}(\mathrm{z}))\{\mathrm{z} \hookrightarrow 3 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\})\\
&\qquad\qquad \text{(by REC-S)}
\end{aligned}$$

We are now done under lazy semantics we've introduced. However, we could further evaluate the expression if we were using a strict semantics.

$$\begin{aligned}
&s(\mathrm{rec}(\mathrm{s}(\mathrm{z}))\{\mathrm{z} \hookrightarrow 3 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\}) \quad \text{(by REC-S)}\\
&\mapsto s(s(\mathrm{rec}(\mathrm{z})\{\mathrm{z} \hookrightarrow 3 \mid \mathrm{s}(x)\texttt{with}y \hookrightarrow \mathrm{s}(y)\}))\\
&\qquad\qquad \text{(by REC-S)}\\
&\mapsto s(s(3)) \qquad\qquad \text{(by REC-Z)}\\
&= 5
\end{aligned}$$

In this case, all rules used have no hypothesis.

3.5 Structural Properties of Type System

Statics of the System **T** enjoys the substitution weakening structural properties. Substitution is the well-definedness property of the type system and is essential to ensure the

type safety during execution. Weakening is crucial for the derivation tree especially accessing variables that are not in the immediate abstraction. We could prove both property by rule induction.

Note: There is one thing that I was confused about the proof of the weakening lemma. Namely, how we interpret and handle contexts that are extended by a variable. say VAR-REFL, it is trivial to show that $\Gamma, x : \tau, e : \tau' \vdash e : \tau'$. But I think the weakening lemma actually wants us to prove $\Gamma, e : \tau', x : \tau \vdash e : \tau'$. I don't see how we can prove that, without the help of permutation lemma, which I don't see how we can prove either, at least in Harper's formulation. While Pierce in his Types and Programming languages avoid such problem by defining VAR-REFL rule's premise as $x \in dom(\Gamma)$, Harper's notion is more obscure since he did not mention and whether the sequence is ordered or unordered. I think Harper might have the same idea to treat the context as a set rather than a sequence in mind, since he insisted that the comma means "extending the context with a fresh variable".Treating the context as a set would make the following permutation lemma trivial. I guess that is the reason why Harper never stated the permutation lemma in his book, but instead used the automatic alpha renaming of abstract binding trees to avoid the problem of repeated variable names in the context.

So for the reset of this report, I will treat the type judging context Γ as a set so the following permutation lemma, which treat the Γ as a sequence trivial.

Lemma. *(Permutation) In* **T** *, if $\Gamma \vdash e : \tau$, and Δ is a permutation of Γ, then $\Delta \vdash e : \tau$.*

Proof. Trivial if Γ is a set. □

For the following proofs of structural properties, when I write $\Gamma, x : \tau$, I will assume that $x \notin dom(\Gamma)$.

Lemma. *(Weakening) In* **T** *, if $\Gamma \vdash e : \tau$, then $\Gamma, x : \tau' \vdash e : \tau$, assuming $x \notin dom(\Gamma)$.*

Proof. By induction on the derivation of the antecedent, $\Gamma \vdash e' : \tau'$.

(i) VAR-REFL
In this case, $e = x'$ has type τ, where

$$\Gamma, x : \tau', x' : \tau \vdash x' : \tau$$

We have

$$\Gamma, x' : \tau, x : \tau' \vdash e : \tau$$

(ii) NAT-INTRO-Z
In this case, $e = \mathrm{z}$ has type $\mathbb{N}$, we have

$$\Gamma, x : \tau' \vdash \mathrm{z} : \mathbb{N}$$

so

$$\Gamma, x : \tau' \vdash e : \mathbb{N}$$

(iii) NAT-INTRO-S
In this case, $e = \mathrm{s}(e')$ has type $\mathbb{N}$, where

$$\Gamma \vdash e' : \mathbb{N}$$

By inductive hypothesis

$$\Gamma, x : \tau' \vdash e' : \mathbb{N}$$

By NAT-INTRO-S

$$\Gamma, x : \tau' \vdash s(e') : \mathbb{N}$$

Since $e = \mathrm{s}(e')$, we get

$$\Gamma, x : \tau' \vdash e : \mathbb{N}$$

(iv) NAT-ELIM
In this case, $e = \mathrm{rec}\{e'; x', y.e''\}(e''')$ has type τ, where

$$\Gamma \vdash e''' : \mathbb{N} \quad \text{(a)}$$

$$\Gamma \vdash e' : \tau \quad \text{(b)}$$

$$\Gamma, x' : \mathbb{N}, y : \tau \vdash e''' : \tau \quad \text{(c)}$$

By the inductive hypothesis, we get

$$\Gamma, x : \tau' \vdash e''' : \mathbb{N} \quad \text{(a)}$$

$$\Gamma, x : \tau' \vdash e' : \tau \quad \text{(b)}$$

$$\Gamma, x : \tau', x' : \mathbb{N}, y : \tau \vdash e''' : \tau \quad \text{(c)}$$

By NAT-ELIM of (a)(b)(c) above,

$$\Gamma, x : \tau' \vdash \mathrm{rec}\{e'; x', y.e''\}(e''') : \tau$$

Since $e = \mathrm{rec}\{e'; x', y.e''\}(e''')$ we get

$$\Gamma, x : \tau' \vdash e : \tau$$

(v) ARROW-INTRO
In this case $e = \lambda(x' : \tau_1)e'$ has type $\tau_1 \to \tau_2$, where

$$\Gamma, x' : \tau_1 \vdash e' : \tau_2$$

By inductive hypothesis, we get

$$\Gamma, x : \tau', x' : \tau_1 \vdash e' : \tau_2$$

By ARROW-INTRO.

$$\Gamma, x : \tau' \vdash \lambda(x' : \tau_1)e' : \tau_1 \to \tau_2$$

Since $e = \lambda(x' : \tau_1)e'$, we get

$$\Gamma, x : \tau' \vdash e : \tau_1 \to \tau_2$$

(vi) ARROW-ELIM
In this case $e = e'(e'')$ has type τ, where

$$\Gamma \vdash e' : \tau_1 \to \tau$$

and

$$\Gamma \vdash e'' : \tau_1$$

By inductive hypothesis,

$$\Gamma, x : \tau' \vdash e' : \tau_1 \to \tau$$

and

$$\Gamma, x : \tau' \vdash e'' : \tau_1$$

By ARROW-ELIM, we get

$$\Gamma, x : \tau' \vdash e'(e'') : \tau$$

Since $e = e'(e'')$, we have

$$\Gamma, x : \tau' \vdash e : \tau$$

□

Lemma. *(Substitution) If* $\Gamma \vdash e_2 : \tau'$*, and* $\Gamma, x : \tau' \vdash e_1 : \tau$*, then* $\Gamma \vdash [e_2/x]e_1 : \tau$.

Proof. By induction on the derivation of $\Gamma, x : \tau' \vdash e_1 : \tau$.

(i) VAR-REFL
In this case, e_1 is a variable with type τ. There are only two cases. Case 1, $e_1 = x$, then $\tau = \tau'$, then $[e_2/x]e_1 = e_2$ has type τ. Case 2, $e_1 \neq x$, then $[e_2/x]e_1 = e_1$ has type τ.

(ii) NAT-INTRO-Z
In this case, $e_1 = \mathrm{z}$ has type $\mathbb{N}$, the type remains $\mathbb{N}$.

(iii) NAT-INTRO-S
In this case, $e_1 = \mathrm{s}(e')$ has type $\mathbb{N}$, where

$$\Gamma, x : \tau' \vdash e' : \mathbb{N}$$

By the inductive hypothesis,

$$\Gamma \vdash [e_2/x]e' : \mathbb{N}$$

By NAT-INTRO-S,

$$\Gamma \vdash s([e_2/x]e') : \mathbb{N}$$

By rules of substitution,

$$\Gamma \vdash [e_2/x]s(e') : \mathbb{N}$$

Since $e_1 = s(e')$, we get

$$\Gamma \vdash [e_2/x]e_1 : \mathbb{N}$$

(iv) NAT-ELIM
In this case, $e_1 = \mathrm{rec}\{e'; x', y.e''\}(e''')$ has type τ, where

$$\Gamma, x : \tau' \vdash e''' : \mathbb{N} \quad \text{(a)}$$

$$\Gamma, x : \tau' \vdash e' : \tau \quad \text{(b)}$$

$$\Gamma, x : \tau', x' : \mathbb{N}, y : \tau \vdash e''' : \tau \quad \text{(c)}$$

By the inductive hypothesis, we get

$$\Gamma \vdash [e_2/x]e''' : \mathbb{N} \quad \text{(a)}$$

$$\Gamma \vdash [e_2/x]e' : \tau \quad \text{(b)}$$

$$\Gamma, x' : \mathbb{N}, y : \tau \vdash [e_2/x]e''' : \tau \quad \text{(c)}$$

By NAT-ELIM of (a)(b)(c) above,

$$\Gamma \vdash \mathrm{rec}\{[e_2/x]e'; x', y.[e_2/x]e''\}([e_2/x]e''') : \tau$$

By rules of substitution

$$\Gamma \vdash [e_2/x]\,\mathrm{rec}\{e'; x', y.e''\}(e''') : \tau$$

Since $e_1 = \mathrm{rec}\{e'; x', y.e''\}(e''')$ we get

$$\Gamma \vdash [e_2/x]e_1 : \tau$$

(v) ARROW-INTRO
In this case $e_1 = \lambda(x' : \tau_1)e'$ has type $\tau_1 \to \tau_2$, where

$$\Gamma, x : \tau', x' : \tau_1 \vdash e' : \tau_2$$

By inductive hypothesis,

$$\Gamma, x' : \tau_1 \vdash [e_2/x]e' : \tau_2$$

By ARROW-INTRO.

$$\Gamma \vdash \lambda(x' : \tau_1)[e_2/x]e' : \tau_1 \to \tau_2$$

By the rules of substitution,

$$\Gamma \vdash [e_2/x](\lambda(x' : \tau_1)e') : \tau_1 \to \tau_2$$

Since $e_1 = \lambda(x' : \tau_1)e'$, we get

$$\Gamma \vdash [e_2/x]e_1 : \tau_1 \to \tau_2$$

(vi) ARROW-ELIM
In this case $e_1 = e'(e'')$ has type τ, where

$$\Gamma, x : \tau' \vdash e' : \tau_1 \to \tau$$

and

$$\Gamma, x : \tau' \vdash e'' : \tau_1$$

By inductive hypothesis,

$$\Gamma \vdash [e_2/x]e' : \tau_1 \to \tau$$

and

$$\Gamma \vdash [e_2/x]e'' : \tau_1$$

By ARROW-ELIM, we get

$$\Gamma \vdash ([e_2/x]e')([e_2/x]e'') : \tau$$

By rules of substitution, we get

$$\Gamma \vdash [e_2/x](e'(e'')) : \tau$$

Since $e_1 = e'(e'')$, we have

$$\Gamma \vdash [e_2/x]e_1 : \tau$$

□

3.5.1 Discussion

Notice that the above proof of weakening and substitution is structurally similar, we could just say that "they can be proved by induction on the structure of type derivation".

Notice the above proofs are also mechanical, it only involves string substitution and invoking induction hypothesis. This characteristic gives rise the to the easy mechanical checking of the proofs by a proof checker such as Coq or Agda. Proofs in such proof assistant are written using an expressive language and such proof tools could check whether the proof is correct.

3.6 Safety

The type of safety of any programming system states that if I have a program that's well typed, then the evaluation will not get stuck. This is expressed by two lemmas, preservation and progress. Preservation states that evaluation does not change the type of the program, and progress states that if a term is well typed, then we could evaluate it.

Lemma. *(Canonical Form of $\mathbb{N}$) In* **T** *, if $\cdot \vdash e : \mathbb{N}$ and e is a value, then either e is z, or e is of the form $\mathrm{s}(e')$, where $\cdot \vdash e' : \mathbb{N}$.*

Proof. By induction on the derivation of $\cdot \vdash e : \mathbb{N}$.

(i) VAR-REFL
This rule does not apply since the type environment should be empty.

(ii) NAT-INTRO-Z
e is z in this case.

(iii) NAT-INTRO-S
e is $\mathrm{s}(e')$ and $\cdot \vdash e' : \mathbb{N}$ by induction hypothesis.

(iv) NAT-ELIM
e is $\mathrm{rec}(-)$ can never be a value, the claim holds vacuously.

(v) ARROW-INTRO
e is not of type $\mathbb{N}$, so this rule does not apply.

(vi) ARROW-ELIM
e is $\mathrm{ap}(-;-)$ can never be a value, the claim holds vacuously.

□

Lemma. *(Canonical Form of $\to$) In* **T** *, if $\cdot \vdash e : \tau_1 \to \tau_2$ and e is a value, then either e is of the form $\lambda(x : \tau_1)e'$.*

Proof. By induction on the derivation of $\cdot \vdash e : \mathbb{N}$.

(i) VAR-REFL
This rule does not apply since the type environment should be empty.

(ii) NAT-INTRO-Z
e is not of type $\tau_1 \to \tau_2$, so this rule does not apply.

(iii) NAT-INTRO-S
e is not of type $\tau_1 \to \tau_2$, so this rule does not apply.

(iv) NAT-ELIM
e is $\mathrm{rec}(-)$ can never be a value, the claim holds vacuously.

(v) ARROW-INTRO
e is of the form $\lambda(x : \tau_1)e'$.

(vi) ARROW-ELIM
e is $\mathrm{ap}(-;-)$ can never be a value, the claim holds vacuously.

□

Lemma. *(Preservation) if $e \mapsto e'$ and $\cdot \vdash e : \tau$, then $\cdot \vdash e' : \tau$.*

Proof. By induction on the derivation of $e \mapsto e'$.

(i) APP-FUNC
APP-FUNC states that

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)}$$

We know that

$$\cdot \vdash e_1(e_2) : \tau$$

The only typing rules that apply in this case is APP-ELIM, so we get

$$\cdot \vdash e_1 : \tau_1 \to \tau$$

and

$$\cdot \vdash e_2 : \tau_1$$

Since $e_1 \mapsto e_1'$, and $\cdot \vdash e_1 : \tau_1 \to \tau$, by induction hypothesis,

$$\cdot \vdash e_1' : \tau_1 \to \tau$$

Since $\cdot \vdash e_2 : \tau_1$, by APP-ELIM we get

$$\cdot \vdash e_1'(e_2) : \tau$$

(ii) APP-BETA
APP-BETA states that

$$\frac{}{(\lambda(x:\tau_1)e)(e_2) \mapsto [e_2/x]e}$$

We know that

$$\cdot \vdash (\lambda(x:\tau_1)e)(e_2) : \tau \tag{1}$$

The only typing rules that was applied to get (1) is APP-ELIM, so we get

$$\cdot \vdash (\lambda(x:\tau_1)e) : \tau_1 \to \tau \tag{2}$$

and

$$\cdot \vdash e_2 : \tau_1 \tag{3}$$

The only typing rules that was applied to get (2) is APP-INTRO, so we get

$$\cdot, x:\tau_1 \vdash e : \tau \tag{4}$$

By applying substitution lemma on (4)(3), we get

$$\cdot \vdash [e_2/x]e : \tau$$

(iii) REC-ARG
REC-ARG states that

$$\frac{e \mapsto e'}{\mathrm{rec}\{e_0; x, y.e_1\}(e) \mapsto \mathrm{rec}\{e_0; x, y.e_1\}(e')}$$

We know that

$$\cdot \vdash \mathrm{rec}\{e_0; x, y.e_1\}(e) : \tau \tag{1}$$

The only typing rules that was applied to get (1) is NAT-ELIM, so we get

$$\cdot \vdash e : \mathbb{N} \tag{2}$$

$$\cdot \vdash e_0 : \tau \tag{3}$$

and

$$\cdot, x:\mathbb{N}, y:\tau \vdash e_1 : \tau \tag{4}$$

Since $e \mapsto e'$, by applying inductive hypothesis with (2), we get

$$\cdot \vdash e' : \mathbb{N} \tag{5}$$

Apply NAT-ELIM to (3)(4)(5), we get

$$\cdot \vdash \mathrm{rec}\{e_0; x, y.e_1\}(e') : \tau$$

(iv) REC-Z
REC-Z states that

$$\frac{}{\mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{z}) \mapsto e_0}$$

We know that

$$\cdot \vdash \mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{z}) : \tau$$

The only typing rules that was applied to get (1) is NAT-ELIM, so we get

$$\cdot \vdash e_0 : \tau$$

(v) REC-S
REC-S states that

$$\frac{}{\mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{s}(e)) \mapsto [e, \mathrm{rec}\{e_0; x, y.e_1\}(e)/x, y]e_1}$$

We know that

$$\cdot \vdash \mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{s}(e)) : \tau \tag{1}$$

The only typing rules that was applied to get (1) is NAT-ELIM, so we get

$$\cdot \vdash \mathrm{s}(e) : \mathbb{N} \tag{2}$$

$$\cdot \vdash e_0 : \tau \tag{3}$$

and

$$\cdot, x:\mathbb{N}, y:\tau \vdash e_1 : \tau \tag{4}$$

The only typing rules that was applied to get (2) is NAT-INTRO-S, so we get

$$\cdot \vdash e : \mathbb{N} \tag{5}$$

Apply NAT-ELIM to (3)(4)(5), we get

$$\cdot \vdash \mathrm{rec}\{e_0; x, y.e_1\}(e) : \tau \tag{6}$$

By applying substitution lemma twice on (6)(5)(4), we get

$$\cdot \vdash [e, \mathrm{rec}\{e_0; x, y.e_1\}(e)/x, y]e_1 : \tau$$

□

Lemma. *(Progress) If* $\cdot \vdash e : \tau$*, then either* $e\,\mathrm{val}$ *or there exists* e' *such that* $e \mapsto e'$*.*

Proof. By induction on the derivation of $\cdot \vdash e : \tau$.

(i) VAR-REFL
VAR-REFL states that

$$\frac{}{\Gamma, x:\tau \vdash x : \tau}$$

The context is not empty. So this rule does not apply.

(ii) NAT-INTRO-Z
NAT-INTRO-Z states that

$$\frac{}{\Gamma \vdash \mathrm{z} : \mathbb{N}}$$

By rule Z-VAL,

$$\mathrm{z}\,\mathrm{val}$$

(iii) NAT-INTRO-S
NAT-INTRO-S states that

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathrm{s}(e) : \mathbb{N}}$$

By rule, S-VAL,

$$\mathrm{s}(e)\,\mathrm{val}$$

(iv) NAT-ELIM
NAT-ELIM states that

$$\frac{\Gamma \vdash e : \mathbb{N} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathbb{N}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathrm{rec}\{e_0; x, y.e_1\}(e) : \tau}$$

In this case $\Gamma = \cdot$, we want to show either $\mathrm{rec}\{e_0; x, y.e_1\}(e)$ is a value or there exists some expression that $\mathrm{rec}\{e_0; x, y.e_1\}(e)$ transitions to.
Since we have $\Gamma \vdash e : \mathbb{N}$ as the premise of the rule, we have

$$\cdot \vdash e : \mathbb{N}$$

By inductive hypothesis, either e val or there exists e' such that $e \mapsto e'$. We consider these two cases one by one.

(a) There exists e' such that $e \mapsto e'$.
Since $e \mapsto e'$, by REC-ARG,

$$\mathrm{rec}\{e_0; x, y.e_1\}(e) \mapsto \mathrm{rec}\{e_0; x, y.e_1\}(e')$$

(b) e val.
Since

$$\cdot \vdash e : \mathbb{N}$$

, by the Canonical Form of $\mathbb{N}$ lemma, we have either e is z or e is of the form $\mathrm{s}(e')$.
If e is z, then REC-Z applies and

$$\mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{z}) \mapsto e_0$$

If e is of the form $\mathrm{s}(e')$, then REC-S applies and

$$\mathrm{rec}\{e_0; x, y.e_1\}(\mathrm{s}(e')) \mapsto [e', \mathrm{rec}\{e_0; x, y.e_1\}(e')/x, y]e_1$$

(v) ARROW-INTRO
ARROW-INTRO states that

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1)e : \tau_1 \to \tau_2}$$

By LAM-VAL

$$\lambda(x : \tau_1)e \text{ val}$$

(vi) ARROW-ELIM The ARROW-ELIM states that

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}$$

In this case $\Gamma = \cdot$, so we have

$$\cdot \vdash e_1 : \tau_1 \to \tau_2$$

and

$$\cdot \vdash e_2 : \tau_1$$

By inductive hypothesis, either e_1 val or there exists e_1' such that $e_1 \mapsto e_1'$.

If $e_1 \mapsto e_1'$, then by APP-FUNC,

$$e_1(e_2) \mapsto e_1'(e_2)$$

If e_1 val:
Since $\cdot \vdash e_1 : \tau_1 \to \tau_2$, by the Canonical Form of $\to$, e_1 is of the form $\lambda(x : \tau_1)e'$.
Then, by APP-BETA

$$(\lambda(x : \tau_1)e')(e_2) \mapsto [e_2/x]e'$$

□

So far, through the preservation and progress theorem, we've proved that the type system of **T** is safe.

4. Summary and Future Works

The System **T** is a simple arithmetic evaluation system that can only compute terminating functions on natural number, but the techniques used in the language and formalization could be adapted to other programming systems with more complex architecture and more advanced language features.

Due to space and time constraints, I am not able to write on all the systems and prove their properties, I will list here a few systems that I find interesting and the essential ideas in constructing those systems. In all these systems, the preservation and progress theorems could be proved, and such proofs ensure correct formulations of the computational properties of those systems.

4.1 The fixed point operator

To construct a Turing-complete language, we would need the ability to compute functions that has infinite loop, namely, the ability to do recursion. System **PCF** extends the syntax of **T** with a fixed point operator, $\mathrm{fix}\{\tau\}(x.e)$, which has the following typing rule.

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathrm{fix}\{\tau\}(x.e) : \tau}$$

and the following computation rule

$$\frac{}{\mathrm{fix}\{\tau\}(x.e) \mapsto [\mathrm{fix}\{\tau\}(x.e)/x]e}$$

The fixed-point expression unfolds itself indefinitely, thus creating capabilities to define non-terminating functions.

4.2 Sum types and product types

Sum types are types of the form $A + B$ which has either expression of type A or expressions of type B as its inhabitant. We use injection operation to construct a sum type. Given x if type A or y of type B, $inl(x)$ or $inr(y)$ would be of type $A+B$. To destruct a sum type, we would do a case operation on the value of the type, and select different branches based on the type of the value in it.

Product types are types of the form $A \times B$. A value of this type is a pair that contains both a value of type A and a value

of type B. Given $x : A$ and $y : B$, the pair (x, y) would have type $A \times B$. To destruct a pair, we could do a projection, so $(x, y) \cdot l = x : A$ and $(x, y) \cdot r = y : B$.

Sum types and product types are called algebraic data types, and they are a good model for a lot of data structures. For instance, Haskell has algebraic data types as its main data structure.

The nullary sum type is $void$ or $\mathbf{0}$ and has no inhabitant. The nullary product type is $unit$ or $\mathbf{1}$ and has only one inhabitant $\langle\rangle$. The natural numbers could be modeled as $\mathbf{1} + \mathbf{1}$, where the left injection represents zero and the right injection represents the successor.

4.3 System F of polymorphic types

System **F** is a good system which has parametric polymorphism. This system performs substitution operation on abstract binding tree used for types. In the syntax for the sort of types in **T** , besides $\mathbb{N}$ and $\rightarrow$, we have two more thing, a variable t and a universal generalization operator $\forall(t.\tau)$.

The syntax of expression is also extended by two new forms of operators, a type generalization $\Lambda(t.e)$ and a type application $\mathrm{App}\{\tau\}(e)$, informally written as $e[\tau]$. Besides the usual context Γ that maps variables to types, we have a type context Δ that holds what type variables are in context.

The associated typing rule is that

$$\frac{\Delta, t \ \texttt{type} \ \ \Gamma \vdash e : \tau}{\Delta\Gamma \vdash \Lambda(t.e) : \forall(t.\tau)}$$

$$\frac{\Delta\Gamma \vdash e : \forall(t.\tau') \qquad \Delta \vdash \tau \ \texttt{type}}{\Delta\Gamma \vdash \mathrm{App}\{\tau\}(e) : [\tau/x]\tau'}$$

The computations rules are

$$\frac{}{\mathrm{App}\{\tau\}(\Lambda(t.e)) \mapsto [\tau/t]e}$$

Notice that the computation rule states that type abstraction can only be destructed through type application. Such correspondence is seen in a lot of formulations of computation that introduces new types. Progress theorems ensure that the computation won't get stuck.

4.4 Generic Programming

Contrary to the generic programming as used in Java, generic programming used in PFPL is used to represent generic data structures and a systematic way of mapping between structurally similar objects with different types.

The system is implemented via a $\mathrm{map}\{t.\tau\}$ operator of arity $(\texttt{Exp}, \texttt{Exp})\texttt{Exp}$,where τ represents the structure of the argument that is polymorphic in t. The the first argument represents a mapping in t, and the second argument is some instantiation of $t.\tau$. Roughly, the typing rule states that

$$\frac{\Gamma, x : \rho \vdash e' : \rho' \qquad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathrm{map}\{t.\tau\}(x.e')(e) : [\rho'/t]\tau}$$

For example, if τ is the form $A \times B$, thus the argument structure must be a pair, the computation rules will map function be applied to each of its components, without knowing what's inside each component as they are structurally irrelevant.

4.5 Inductive and Coinductive Types

The inductive types are specified by the type operator $\mu(t.\tau)$ which corresponds to the least fixed point solution of some type equation, while the coinductive types are specified by the type operator $\nu(t.\tau)$, which corresponds to the greatest fixed point solution of some type equation.

I am still learning Harper's formulation of these concepts.

4.6 Untyped Lambda Calculus

Perhaps the untyped lambda calculus is one reason why Harper doesn't want to give call his "statics" "type system", since it is still possible to give a statics formulation of the classical untyped calculus – all terms have a single type. And the statics states the well-formedness of expressions. The statics ensure that no free variables occur in a closed program.

4.7 Modal Distinction (Mutable States)

Harper introduced the system "Modernized Algol" as a formulation of a language that has modal distinction. While Haskell uses the notion of IO monad for the same concept, what underlies the idea of an IO monad is another sort of syntax, called "command", or m.

The most significant members of this set are $ret(e)$, which encapsulates a pure expression into command, $bnd(e; x.m)$ which chains together two commands and $dcl(e; a.m)$ which opens a new scope. Expressions are extended with operator $cmd(m)$ which encapsulates a command. The evaluation rules ensure that the variable allocated are deallocated when the current evaluation finishes, by evaluates the body of declaration $a.m$ in place instead of substitution.

5. Concluding Remarks

In this report, I've presented the abstract syntax tree formulation. Then I've presented a case study with System **T** of various language properties such as type safety. Finally, I've discussed a few applications of Harper's system. The methodology introduced by Harper could be applied to formulate the computational properties of a lot of paradigm cases in language design and system implementation.

Acknowledgments

I would like to thank Professor Sicun Gao (Sean Gao) for providing me this valuable opportunity for completing this report. I have learned a lot in the process of writing this report.