Interpreting IfArith

CIS352 — Spring 2021

Kris Micinski

Bug in P2 public testcases (TC)

- Very sorry about that—bonus portion I didn't assign was buggy (caused tests to be buggy)
- No changes needed to starter code—check out p2-fix from autograde
  - P2 is blackholed, you can't submit—but you can copy-paste your solution to p2-fix
- Deadline extended to Oct 15 (Saturday) since it was my fault

Today, we’re going to start building our **own** languages

We’re going to do this by writing **interpreters**

To build a programming language, we need two things:

- A **syntax** for the language (and the ability to **parse** it)
- A **semantics** for the language. Typically either an **interpreter** or a **compiler**

For this class, all of our programs are going to be written as Racket datums

We specify syntax via a predicate that uses pattern matching

This means we can just write programs in our language just by building data in Racket

Here is the first language we will define:

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

"Any integer is a program in our language."

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

“If e0 is an expression in our language, and e1 is an expression in our language, `(plus ,e0 ,e1) is, too.”

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

Here are some example expressions:

```
‘(plus 1 (div 2 3))
'(if 0 (plus 1 2) (div 2 2))
'(if 0 (plus 1 (div 2 3)) (if 1 (plus 2 3) 0))
```

IMPORTANT NOTE

We are defining a **new language** by **using** Racket. But our language is **not** Racket. In Racket, booleans are #t and #f. In **our** language, we will use 0 to represent false and non-0 to represent true (as in C).

Again, because this is confusing

When writing interpreters, always be careful to mentally separate the **language you are defining** and the language you are using to build the interpreter (Racket).

This can become confusing as the languages we build will "look like" Racket. Try to be mindful.

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

The “result” of programs will be a Racket integer:

```
(define value? integer?)
```

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

The “result” of programs will be a Racket integer:

```
(define value? integer?)

(define/contract (evaluate e)
  (-> expr? value?)
  ‘todo)
```

What should the following return…?
Remember, this is our own **new language we are defining, not necessarily Racket**

```
(evaluate '(plus 1 2))
=> 3
(evaluate '(if 0 (plus 1 2) (div 2 2)))
=> ‘todo
(evaluate '(if 1 (div 4 3) (plus 1 -1)))
=> ‘todo
```

What should the following return…?
Remember, this is our own **new language we are defining, not necessarily Racket**

```
(evaluate '(plus 1 2))
=> 3
(evaluate '(if 0 (plus 1 2) (div 2 2)))
=> 1
(evaluate '(if 1 (div 4 3) (plus 1 -1)))
=> 4/3
```

Now, let's build **evaluate** ourselves

In this lecture, we built a **metacircular** interpreter

Important Definition

A metacircular interpreter is an interpreter which uses features of a "host" language to define the semantics of a "target" language

Which features of Racket did we use to define our language…?

Important Definition

A metacircular interpreter is an interpreter which uses features of a “host” language to define the semantics of a “target” language

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    …
```

Notice how we **inherit** the definition of + from Racket

John Reynolds introduced metacircular interpreters in 1978. One key idea: metacircular interpreters inherit properties of their host language!

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Reynolds/Definitional

72-1

1

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call-by-value versus call-by-name) in the defined language

INTRODUCTION

An important and frequently used method of defining a programming language is to give an interpreter for the language which is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have

Note: our interpreter is **direct-style**, it is **not** tail recursive

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    …
```

This means we are relying on Racket's **stack** as well

We will later see how to eliminate the need for this

Natural Deduction for IfArith

CIS352 — Fall 2022

Kris Micinski

In this lecture, we'll introduce **natural deduction**

Natural deduction is a mathematical formalism that helps ground the ideas in metacircular interpreters

Natural deduction first used in mathematical logic, to specify **proofs** using inductive data

We will use natural deduction as a framework for specifying semantics of various languages throughout the course

Introduction Rules **Elimination Rules**

$$\dfrac{\dfrac{}{\stackrel{N}{\vdash} A}u \atop {\vdots \atop \stackrel{N}{\vdash} B}}{\stackrel{N}{\vdash} A \supset B}\supset\mathrm{I}^u \qquad \dfrac{\stackrel{N}{\vdash} A \supset B \qquad \stackrel{N}{\vdash} A}{\stackrel{N}{\vdash} B}\supset\mathrm{E}$$

$$\dfrac{\dfrac{}{\stackrel{N}{\vdash} A}u \atop {\vdots \atop \stackrel{N}{\vdash} p}}{\stackrel{N}{\vdash} \neg A}\neg\mathrm{I}^{p,u} \qquad \dfrac{\stackrel{N}{\vdash} \neg A \qquad \stackrel{N}{\vdash} A}{\stackrel{N}{\vdash} C}\neg\mathrm{E}$$

$$\dfrac{\stackrel{N}{\vdash} [a/x]A}{\quad}\forall\mathrm{I}^a \qquad \dfrac{\stackrel{N}{\vdash} \forall x.\, A}{\quad}\forall\mathrm{E}$$

When we specify the semantics of a language using natural deduction, we give its semantics via a set of **inference rules**

Rules read: if the thing on the top is true, then the thing on the **bottom** is also true.

This rule says: "if c is an integer (mathematically: c ∈ ℚ), then c evaluates to c."

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c}$$

Note: the notation e ⇓ v is read "e evaluates to v."

Some rules will have more than one **antecedent** (thing on the top).

You read these: "if the first thing, and second thing, and … are **all** true, then the thing on the bottom is true."

$$\textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

"If $e_0 \Downarrow n_0$, and $e_1 \Downarrow n_1$, and $n' = n_0 + n1$, **then** I can say $(\text{plus } e_0\ e_1) \Downarrow n'$."

$$\textbf{Plus}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\mathsf{div}\ e_0\ e_1) \Downarrow n'}$$

The natural deduction rule for **div** is similar

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\mathsf{div}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0} : \frac{e \Downarrow 0}{(\mathsf{not}\ e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1} : \frac{e \Downarrow n \quad n \neq 0}{(\mathsf{not}\ e) \Downarrow 0}$$

We have **two** rules for not

Natural Deduction Rules for IfArith

$$\textbf{Const}: \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\mathsf{div}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0}: \frac{e \Downarrow 0}{(\mathsf{not}\ e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1}: \frac{e \Downarrow n \quad n \neq 0}{(\mathsf{not}\ e) \Downarrow 0}$$

$$\textbf{If}_\mathbf{T}: \frac{e_0 \Downarrow 0 \quad e_1 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'} \qquad \textbf{If}_\mathbf{F}: \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'}$$

Question: Now that we have the rules, what can we do with them?

Answer: Use them to **formally prove** that some program calculates some result

Let’s say I want to prove that the following program evaluates to 4:

```
(if (plus 1 -1) 3 4)
```

What rule could go here..?

$$\frac{???}{(\text{if } (\text{plus } 1 \;\; -1) \; 3 \; 4) \Downarrow 4}$$

$$\mathbf{If_T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'} \quad \mathbf{If_F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{???}{\big(\mathsf{if}\ (\mathsf{plus}\ 1\ \ -1)\ 3\ 4\big) \Downarrow 4}$$

$$\mathbf{If_T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'} \quad \mathbf{If_F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{???}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

To apply a natural-deduction rule,
we must perform **unification**

There can be no variables in the resulting unification!

$$\mathbf{If_F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{(\text{plus } 1\ \ -1) \Downarrow 0 \qquad\qquad 4 \Downarrow 4}{\left(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

We perform unification:

e_0: (plus 1 -1), e_1: 3

e_2: 4, n’: 4

Not done yet, now we have to prove **these** things

$$\frac{(\text{plus } 1\ \ -1) \Downarrow 0 \qquad 4 \Downarrow 4}{\left(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

Why can we say $4 \Downarrow 4$? Because of the **Const** rule

$$\frac{(\text{plus } 1 \ -1) \Downarrow 0 \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\text{if } (\text{plus } 1 \ -1) \ 3 \ 4\right) \Downarrow 4}$$

We're not done yet, because **plus** requires an antecedent:

$$\textbf{Plus}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\frac{(\text{plus } 1\ \ -1) \Downarrow 0 \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

But we're **still** not done, because we need to finish these three

$$\dfrac{\dfrac{1 \Downarrow 1 \quad -1 \Downarrow -1 \quad 1 + -1 = 0}{(\mathsf{plus}\ 1\ -1) \Downarrow 0} \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{(\mathsf{if}\ (\mathsf{plus}\ 1\ -1)\ 3\ 4) \Downarrow 4}$$

Things that are simply true from algebra require no antecedents, we take them as "axioms."

$$\dfrac{\dfrac{\dfrac{1 \in \mathbb{Q}}{1 \Downarrow 1} \quad \dfrac{-1 \in \mathbb{Q}}{-1 \Downarrow -1} \quad \dfrac{}{1 + -1 = 0}}{(\mathsf{plus}\ 1\ -1) \Downarrow 0} \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

This is a complete proof that the program computes 4

$$\dfrac{\dfrac{\dfrac{1 \in \mathbb{Q}}{1 \Downarrow 1} \quad \dfrac{-1 \in \mathbb{Q}}{-1 \Downarrow -1} \quad \dfrac{}{1 + -1 = 0}}{(\mathsf{plus}\ 1\ -1) \Downarrow 0} \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ -1)\ 3\ 4\right) \Downarrow 4}$$

Question: could you write this proof..? What would happen if you tried…?

$$\frac{???}{\left(\text{if }(\text{plus } 1\ \ -1)\ 3\ 4 \Downarrow 3\right)}$$

$$\textbf{If}_\textbf{T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\textsf{if } e_0\ e_1\ e_2) \Downarrow n'} \quad \textbf{If}_\textbf{F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\textsf{if } e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{:(}{\left(\textsf{if } (\textsf{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 3}$$

Answer: you **can't** write this proof, because IfT will only let you evaluate e1 when e0 is non-0!

$$\frac{???}{\text{(plus (plus 0 1) 2)} \Downarrow 3} \qquad \frac{???}{\text{(if 1 (div 1 1) 2)} \Downarrow 1}$$

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0/n_1}{(\text{div } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0} : \frac{e \Downarrow 0}{(\text{not } e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1} : \frac{e \Downarrow n \quad n \neq 0}{(\text{not } e) \Downarrow 0}$$

$$\textbf{If}_\mathbf{T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'} \quad \textbf{If}_\mathbf{F} : \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$