Date	Topic
Week 1	**How to design a programming language?**
06/24	Intro, Syntax I
06/25	Syntax II, Inference Rules
06/26	Operational Semantics
06/27	Section: Python Tutorial
Week 2	**What makes a programming language?**
07/01	Operational Semantics Practicum
07/02	Variables
07/03	Lambda Calculus
07/04	*No class (Independence Day)*
Week 3	**How to abstract *data*?**
07/08	Call-by
07/09	Rice's Theorem, Soundness & Completeness
07/10	Types
Week 4	**How to abstract *computation*?**
07/14	-
07/15	Quiz 1 (scheduled)
07/16	Type Soundness
07/17	Bidirectional Typing
Week 5	**How to change the world?**
07/22	Sum Types, Recursive Types
07/23	Polymorphism, Type Inference
07/24	*Quiz 2 (tentative)*
07/25	Section
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD

We've already overtaken the pace of all prior CS 162

Software correctness

- Problem: Rice's theorem. Any non-trivial property about a Turing machine is undecidable
- This means that we can never give an algorithm, that for all programs can decide if this program has an error on some inputs.
- What can we do?
- Give up?

This was the only slide on Rice's Theorem from last year's CS 162.

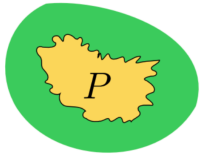
But we've actually proven this theorem!

Date	Topic
Week 1	**How to design a programming language?**
06/24	Intro, Syntax I
06/25	Syntax II, Inference Rules
06/26	Operational Semantics
06/27	Section: Python Tutorial
Week 2	**What makes a programming language?**
07/01	Operational Semantics Practicum
07/02	Variables
07/03	Lambda Calculus
07/04	*No class (Independence Day)*
Week 3	**How to abstract *data*?**
07/08	Call-by
07/09	Rice's Theorem, Soundness & Completeness
07/10	Types
Week 4	**How to abstract *computation*?**
07/14	-
07/15	Quiz 1 (scheduled)
07/16	Type Soundness
07/17	Bidirectional Typing
Week 5	**How to change the world?**
07/22	Sum Types, Recursive Types
07/23	Polymorphism, Type Inference
07/24	*Quiz 2 (tentative)*
07/25	Section
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD

We've already overtaken the pace of all prior CS 162

Soundness

- Specifically, we only care about abstract semantics that are sound
- Soundness means that for any program: If we evaluate it under *concrete* semantics (operational semantics) and our *abstract* semantics, the abstract value obtained overapproximates the concrete value.



This was the only slide on "soundness".

But we've saw a much more general definition + completeness + how they relate to Rice's Theorem

Date	Topic
Week 1	**How to design a programming language?**
06/24	Intro, Syntax I
06/25	Syntax II, Inference Rules
06/26	Operational Semantics
06/27	Section: Python Tutorial
Week 2	**What makes a programming language?**
07/01	Operational Semantics Practicum
07/02	Variables
07/03	Lambda Calculus
07/04	*No class (Independence Day)*
Week 3	**How to abstract *data*?**
07/08	Call-by
07/09	Rice's Theorem, Soundness & Completeness
07/10	Types
Week 4	**How to abstract *computation*?**
07/14	-
07/15	Quiz 1 (scheduled)
07/16	Type Soundness
07/17	Bidirectional Typing
Week 5	**How to change the world?**
07/22	Sum Types, Recursive Types
07/23	Polymorphism, Type Inference
07/24	*Quiz 2 (tentative)*
07/25	Section
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD

We've already overtaken the pace of all prior CS 162

Soundness

- The reason we only care about sound abstract semantics is the following:
- Theorem: If some abstract semantics are sound and an expression is of abstract value x, then its concrete value y is always part of the abstract value x.
- Why is this useful?

This was the only slide on type soundness.

But we've proven this theorem + saw how to break soundness

Date	Topic
Week 1	**How to design a programming language?**
06/24	Intro, Syntax I
06/25	Syntax II, Inference Rules
06/26	Operational Semantics
06/27	Section: Python Tutorial
Week 2	**What makes a programming language?**
07/01	Operational Semantics Practicum
07/02	Variables
07/03	Lambda Calculus
07/04	*No class (Independence Day)*
Week 3	**How to abstract *data*?**
07/08	Call-by
07/09	Rice's Theorem, Soundness & Completeness
07/10	Types
Week 4	**How to abstract *computation*?**
07/14	-
07/15	Quiz 1 (scheduled)
07/16	Type Soundness
07/17	Bidirectional Typing
Week 5	**How to change the world?**
07/22	Sum Types, Recursive Types
07/23	Polymorphism, Type Inference
07/24	*Quiz 2 (tentative)*
07/25	Section
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD

We've already overtaken the pace of all prior CS 162

Types in λ^+

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \textsc{T-Int} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{+, -, *\}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \textsc{T-Arith}$$

Type environment

$$\Gamma \vdash x : \mathsf{T}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad x : \mathsf{T}_1, \Gamma \vdash e_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mathsf{T}_2}\ \textsc{T-Let}$$

$$\frac{x : \mathsf{T}_1, \Gamma \vdash e : \mathsf{T}_2}{\Gamma \vdash (\mathsf{lambda}\ x : \mathsf{T}_1.\ e) : \mathsf{T}_1 \to \mathsf{T}_2}\ \textsc{T-Lambda}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_1}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2}\ \textsc{T-App}$$

This was one of the slides on typing rules.
But you also know how to design ~~them~~!
(operational semantics + abstraction function + lazy substitution)

- As an instructor, I'm extremely proud of what we've achieved in just 4 weeks.
- We've gone far beyond the standard PL theory course at UCSB.
- You're well-equipped to read PL papers and join research projects.

Next steps for us:

Week 5:

- 2 more lectures (HW 4 concepts)
- Quiz 2 (Thursday) covers up to today's lecture

Wee 6:

- Let's chill & reflect.
- Schedule a 1-1 meeting with me to go over your quiz corrections
- Finish your HW 5 + reflection essays x2.

Date	Topic
Week 1	**How to design a programming language?**
06/24	Intro, Syntax I
06/25	Syntax II, Inference Rules
06/26	Operational Semantics
06/27	Section: Python Tutorial
Week 2	**What makes a programming language?**
07/01	Operational Semantics Practicum
07/02	Variables
07/03	Lambda Calculus
07/04	*No class (Independence Day)*
Week 3	**How to abstract *data*?**
07/08	Call-by
07/09	Rice's Theorem, Soundness & Completeness
07/10	Types
Week 4	**How to abstract *computation*?**
07/14	-
07/15	Quiz 1 (scheduled)
07/16	Type Soundness
07/17	Bidirectional Typing
Week 5	**How to change the world?**
07/22	Sum Types, Recursive Types
07/23	Polymorphism, Type Inference
07/24	*Quiz 2 (tentative)*
07/25	Section
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD

for HW4

- As an instructor, I'm extremely proud of what we've achieved in just 4 weeks.
- We've gone far beyond the standard PL theory course at UCSB.
- You're well-equipped to read PL papers and join research projects.

Next steps for you to think about:

Contact professors for research opportunities

- Yu Feng (PL + security / blockchain / LLM)
- Ben Hardekopf (PL + architecture / systems)
- Tevfik Bultan (software engineering, LLM, PL)

Take these classes at UCSB

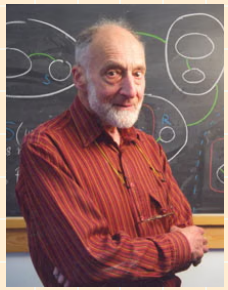
- CS 160 - Compilers (Ben/Tevfik/Chris)
- CS 260 - Static Analysis (Ben)
- CS 267 - Model Checking (Tevfik)
- CS 292C - Formal Verification (Yu)

Date	Topic
Week 1	**How to design a programming language?**
06/24	Intro, Syntax I
06/25	Syntax II, Inference Rules
06/26	Operational Semantics
06/27	Section: Python Tutorial
Week 2	**What makes a programming language?**
07/01	Operational Semantics Practicum
07/02	Variables
07/03	Lambda Calculus
07/04	*No class (Independence Day)*
Week 3	**How to abstract *data*?**
07/08	Call-by
07/09	Rice's Theorem, Soundness & Completeness
07/10	Types
Week 4	**How to abstract *computation*?**
07/14	-
07/15	Quiz 1 (scheduled)
07/16	Type Soundness
07/17	Bidirectional Typing
Week 5	**How to change the world?**
07/22	Sum Types, Recursive Types
07/23	Polymorphism, Type Inference
07/24	*Quiz 2 (tentative)*
07/25	Section
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD

Where we are:

- Static analysis: making sure software is bug-free before loading it onto an airplane.
- Type systems: sound but incomplete analysis
- Type soundness theorem:

$$\Gamma \vdash e : t \quad \rightarrow \quad e \Rightarrow v$$



In other words, "well-typed programs don't go wrong" (slogan).

- Problem: $\Gamma \vdash e : t$ is not algorithmic, i.e., we have to guess.
- Solution: decompose it into:

Type synthesis: $\boxed{\Gamma \vdash e \downarrow t}$ **Type checking:** $\boxed{\Gamma \vdash e \uparrow t}$

This is called bidirectional typing

By the way,

- all of this is "not that hard"
- none of it will matter

Elon Musk
@elonmusk

Done right, a compiler should be able to figure out type automatically. It's not that hard.

Not that it will matter much in the AI future.

1:31 PM · Jan 6, 2024 · **595.7K** Views

235 358 1.3K 151

How to design a bidirectional type system?

$$\frac{}{\Gamma, x : t \vdash x : t}\ \text{T-Id} \qquad \frac{\Gamma \vdash x : t \quad (x \neq y)}{\Gamma, y : t \vdash x : t}\ \text{T-Weaken}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Nat}}\ \text{T-Nat} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Nat} \quad \Gamma \vdash e_2 : \mathsf{Nat} \quad (\oplus \in \{+, -, *\})}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Nat}}\ \text{T-Arith} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Nat} \quad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 : \mathsf{Bool}}\ \text{T-Eq}$$

$$\frac{}{\Gamma \vdash b : \mathsf{Bool}}\ \text{T-Bool} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad (t_2 = t_3)}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : t_2}\ \text{T-If}$$

$$\text{(No rule for variables)} \qquad \frac{(\Delta(X) = t) \quad \Gamma \vdash e : t}{\Gamma \vdash X : t}\ \text{T-Abbrev} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2}\ \text{T-Let}$$

$$\frac{\Gamma, x : t_i \vdash e : t_o}{\Gamma \vdash \lambda x.\ e : t_i \to t_o}\ \text{T-Lambda} \qquad \frac{\Gamma \vdash e_1 : t_i \to t_o \quad \Gamma \vdash e_2 : t_i}{\Gamma \vdash e_1\ e_2 : t_o}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1, t_2)}\ \text{T-Pack} \qquad \frac{\Gamma \vdash e_1 : (t_{11}, t_{12}) \quad \Gamma, x_1 : t_{11}, x_2 : t_{12} \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ (x_1, x_2) = e_1\ \mathsf{in}\ e_2 : t}\ \text{T-Unpack}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e : t) : t}\ \text{T-Ann}$$

Easy! Let's visit all 8.5 billion alternative universes and see which one works.

Solution 2: talk to a logician and steal their work

PL designer

Work of mathematicians & logicians

The Pfenning Recipe for designing bidirectional type systems

AKA "the one outcome in which we defeat Thanos"

Intercalation Calculus for Intuitionistic Propositional Logic

by

Saverio Cittadini

May 1992

Report CMU-PHIL-29

Tridirectional Typechecking

Jana Dunfield
jd169@queensu.ca

Frank Pfenning
fp@cs.cmu.edu

Carnegie Mellon University
Pittsburgh, PA

ABSTRACT

In prior work we introduced a pure type assignment system that encompasses a rich set of property types, including intersections, unions, and universally and existentially quantified dependent types. This system was shown sound with respect to a call-by-value operational semantics with effects, yet is inherently undecidable.

In this paper we provide a decidable formulation for this system based on bidirectional checking, combining type synthesis and analysis following logical principles. The presence of unions and existential quantification requires the additional ability to visit subterms in evaluation position before the context in which they occur, leading to a *tridirectional* type system. While soundness with respect to the type assignment system is immediate, completeness requires the novel concept of *contextual type annotations*, introducing a notion from the study of principal typings into the source program.

Categories and Subject Descriptors: F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*Type structure*; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms: Languages, Theory

Keywords: Type refinements, intersection types, union types, dependent types

compilation), it has been less successful in making the expressive type systems directly available to the programmer. One reason for this is the difficulty of finding the right balance between the brevity of the additional required type declarations and the feasibility of the typechecking problem. Another is the difficulty of giving precise and useful feedback to the programmer on ill-typed programs.

In prior work [9] we developed a system of pure type assignment designed for call-by-value languages with effects and proved progress and type preservation. The intended atomic program properties are data structure refinements [11, 10, 28], but our approach does not depend essentially on this choice. Atomic properties can be combined into more complex ones through intersections, unions, and universal and existential quantification over index domains. As a pure type assignment system, where terms do not contain any types at all, it is inherently undecidable [4].

In this paper we develop an annotation discipline and typechecking algorithm for our earlier type assignment system. The major contribution is the type system itself which contains several novel ideas, including an extension of the paradigm of bidirectional typechecking to union and existential types, leading to the *tridirectional system*. While type soundness follows immediately by erasure of annotations, completeness requires that we insert *contextual typing annotations* reminiscent of principal typings [13, 25]. Decidability is not obvious; we prove it by showing that a slightly altered *left*

Make vs Use

$t \in type$	$::=$	Nat	natural number type
	$\mid$	Bool	boolean type
	$\mid$	$t_1 \to t_2$	function type
	$\mid$	(t_1, t_2)	pair type

Step 1: For every type,

- identify its introduction form - the language construct that "makes" / "constructs" something of that type.
- identify its elimination form - the language construct that "uses" / "consumes" something of that type

How to make a Bool? ⇝ $(n_1 = n_2)$
Boolean constants

How to use a Bool?
if b th... else...

How to make a Nat?
0, 1, 2, 5, ...

How to use a Nat?
+, *, -, ==,

How to make a T1 -> T2?
λx. e

How to use a T1 -> T2?
application $(e_1\ e_2)$

How to make a (T1, T2)? (e_1, e_2)

How to use a (T1, T2)?
let $(x_1, x_2) = e_1$ in e_2

Step 2: For every typing rule, classify it as

1. an introduction rule for a type (where the intro form of the type appears, usually appear below the line)
2. an elimination rule for a type (the elim form of the type appears, usually below the line)
3. + others

Exercise: carry out this step for every rule in Lamp's type system.

$$\frac{}{\Gamma, x : t \vdash x : t}\ \text{T-Id} \qquad \frac{\Gamma \vdash x : t \quad (x \neq y)}{\Gamma, y : t \vdash x : t}\ \text{T-Weaken}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Nat}}\ \text{T-Nat} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Nat} \quad \Gamma \vdash e_2 : \mathsf{Nat} \quad (\oplus \in \{+, -, *\})}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Nat}}\ \text{T-Arith} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Nat} \quad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 : \mathsf{Bool}}\ \text{T-Eq}$$

$$\frac{}{\Gamma \vdash b : \mathsf{Bool}}\ \text{T-Bool} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad (t_2 = t_3)}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : t_2}\ \text{T-If}$$

(No rule for variables) $\qquad \frac{(\Delta(X) = t) \quad \Gamma \vdash e : t}{\Gamma \vdash X : t}\ \text{T-Abbrev} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2}\ \text{T-Let}$

$$\frac{\Gamma, x : t_i \vdash e : t_o}{\Gamma \vdash \lambda x.\ e : t_i \to t_o}\ \text{T-Lambda} \qquad \frac{\Gamma \vdash e_1 : t_i \to t_o \quad \Gamma \vdash e_2 : t_i}{\Gamma \vdash e_1\ e_2 : t_o}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1, t_2)}\ \text{T-Pack} \qquad \frac{\Gamma \vdash e_1 : (t_{11}, t_{12}) \quad \Gamma, x_1 : t_{11}, x_2 : t_{12} \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ (x_1, x_2) = e_1\ \mathsf{in}\ e_2 : t}\ \text{T-Unpack}$$

Step 3: For every intro / elim rule, identify its principal judgment = the judgment where the type being introduced / eliminated appears

Step 4: Assign up/down arrows using the rules:

- principal judgment of intro rule = check ($e \uparrow t$)
- principla judgment of elim rule = synth ($e \downarrow t$)

Step 5: Based on assigned arrows, propogate information "naturally"

Step 6:

- Remaining unresolved choices = check
- Variables = synth
- Add a rule to allow synth / check to "meet in the middle": if I'm checking e has type t1, and I synthesized t2 instead, then this is fine as long as t1 = t2

$$\frac{\Gamma \vdash e \downarrow t_2 \qquad (t_1 = t_2)}{\Gamma \vdash e \uparrow t_1} \downarrow\uparrow$$

$$\frac{}{\Gamma, x : t \vdash x \downarrow t}\ \text{T-Id} \qquad \frac{\Gamma \vdash x \downarrow t \quad (x \neq y)}{\Gamma, y : t \vdash x \downarrow t}\ \text{T-Weaken}$$

$$\frac{}{\Gamma \vdash n \uparrow \mathsf{Nat}}\ \text{T-Nat} \qquad \frac{\Gamma \vdash e_1 \downarrow \mathsf{Nat} \quad \Gamma \vdash e_2 \downarrow \mathsf{Nat} \quad (\oplus \in \{+, -, *\})}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Nat}}\ \text{T-Arith} \qquad \frac{\Gamma \vdash e_1 \downarrow \mathsf{Nat} \quad \Gamma \vdash e_2 \downarrow \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 : \mathsf{Bool}}\ \text{T-Eq}$$

NatElim NatElim

$$\frac{}{\Gamma \vdash b \uparrow \mathsf{Bool}}\ \text{T-Bool} \qquad \frac{\Gamma \vdash e_1 \downarrow \mathsf{Bool} \quad \Gamma \vdash e_2 \uparrow t_2 \quad \Gamma \vdash e_3 \uparrow t_3 \quad (t_2 = t_3)}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \uparrow t_2}\ \text{T-If}$$

(No rule for variables) $\qquad \frac{(\Delta(X) = t) \quad \Gamma \vdash e : t}{\Gamma \vdash X : t}\ \text{T-Abbrev} \qquad \frac{\Gamma \vdash e_1 \downarrow t_1 \quad \Gamma, x : t_1 \vdash e_2 \uparrow t_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \uparrow t_2}\ \text{T-Let}$

$$\frac{\Gamma, x : t_i \vdash e \uparrow t_o}{\Gamma \vdash \lambda x.\ e \uparrow t_i \to t_o}\ \text{T-Lambda} \qquad \frac{\Gamma \vdash e_1 \downarrow t_i \to t_o \quad \Gamma \vdash e_2 \downarrow t_i}{\Gamma \vdash e_1\ e_2 \downarrow t_o}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 \uparrow t_1 \quad \Gamma \vdash e_2 \uparrow t_2}{\Gamma \vdash (e_1, e_2) \uparrow (t_1, t_2)}\ \text{T-Pack} \qquad \frac{\Gamma \vdash e_1 \downarrow (t_{11}, t_{12}) \quad \Gamma, x_1 : t_{11}, x_2 : t_{12} \vdash e_2 \uparrow t}{\Gamma \vdash \mathsf{let}\ (x_1, x_2) = e_1\ \mathsf{in}\ e_2 \uparrow t}\ \text{T-Unpack}$$

$x : \mathsf{Bool}, y : \mathsf{Nat}, z : \mathsf{Nat} \vdash \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z+1 \uparrow \mathsf{Nat}$

$x : \mathsf{Bool}, y : \mathsf{Nat} \vdash \lambda z.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z+1 \uparrow \mathsf{Nat} \to \mathsf{Nat}$

$x : \mathsf{Bool} \vdash \lambda y.\ \lambda z.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z+1 \uparrow \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$

$\vdash$ `\x. \y. \z. if x then y else z + 1` $\uparrow \mathsf{Bool} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$

$$\frac{}{\Gamma, x : t \vdash x : t}\ \text{T-Id} \qquad \frac{\Gamma \vdash x : t \quad (x \neq y)}{\Gamma, y : t \vdash x : t}\ \text{T-Weaken}$$

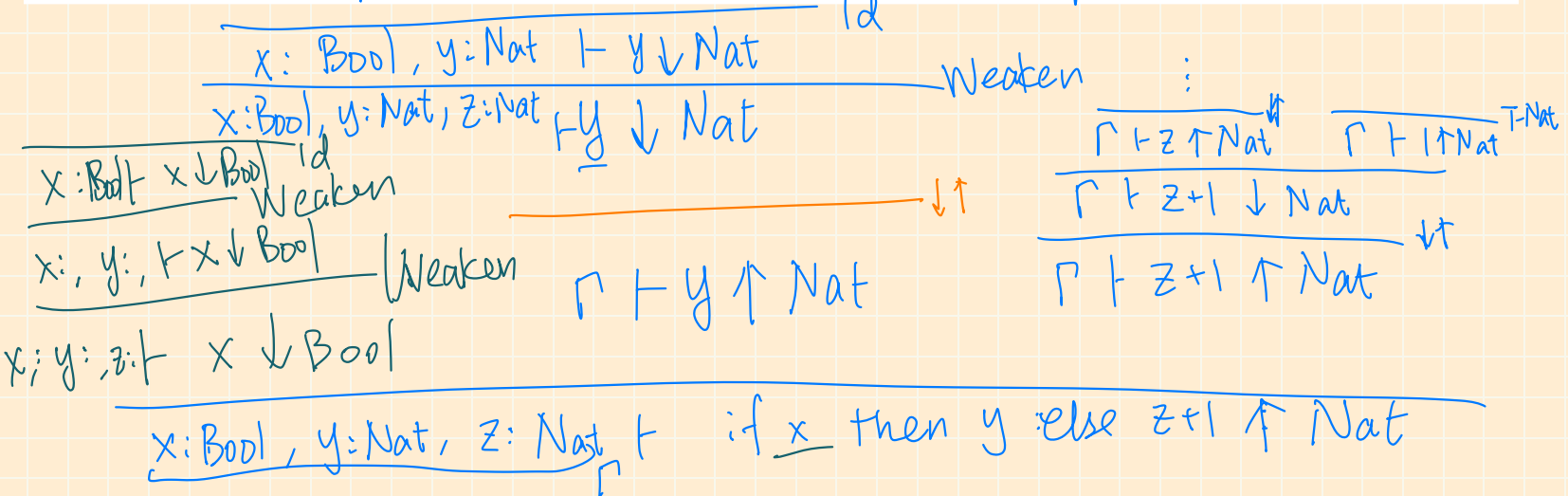
$$\frac{}{\Gamma \vdash n : \mathsf{Nat}}\ \text{T-Nat} \qquad \frac{\Gamma \vdash e_1 \uparrow \mathsf{Nat} \quad \Gamma \vdash e_2 \uparrow \mathsf{Nat} \quad (\oplus \in \{+, -, *\})}{\Gamma \vdash e_1 \oplus e_2 \downarrow \mathsf{Nat}}\ \text{T-Arith} \qquad \frac{\Gamma \vdash e_1 \uparrow \mathsf{Nat} \quad \Gamma \vdash e_2 \uparrow \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 \downarrow \mathsf{Bool}}\ \text{T-Eq}$$

$$\frac{}{\Gamma \vdash b \uparrow \mathsf{Bool}}\ \text{T-Bool} \qquad \frac{\Gamma \vdash e_1 \downarrow \mathsf{Bool} \quad \Gamma \vdash e_2 \uparrow t_2 \quad \Gamma \vdash e_3 \uparrow t_2}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \uparrow t_2}\ \text{T-If}$$

(No rule for variables) $\qquad \frac{(\Delta(X) = t) \quad \Gamma \vdash e : t}{\Gamma \vdash X : t}\ \text{T-Abbrev} \qquad \frac{\Gamma \vdash e_1 \downarrow t_1 \quad \Gamma, x : t_1 \vdash e_2 \uparrow t_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \uparrow t_2}\ \text{T-Let}$

$$\frac{\Gamma, x : t_i \vdash e \uparrow t_o}{\Gamma \vdash \lambda x.\ e \uparrow t_i \to t_o}\ \text{T-Lambda} \qquad \frac{\Gamma \vdash e_1 \downarrow t_i \to t_o \quad \Gamma \vdash e_2 \uparrow t_i}{\Gamma \vdash e_1\ e_2 \downarrow t_o}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 \uparrow t_1 \quad \Gamma \vdash e_2 \uparrow t_2}{\Gamma \vdash (e_1, e_2) \uparrow (t_1, t_2)}\ \text{T-Pack} \qquad \frac{\Gamma \vdash e_1 \downarrow (t_{11}, t_{12}) \quad \Gamma, x_1 : t_{11}, x_2 : t_{12} \vdash e_2 \uparrow t}{\Gamma \vdash \mathsf{let}\ (x_1, x_2) = e_1\ \mathsf{in}\ e_2 \uparrow t}\ \text{T-Unpack}$$



Let's add type annotations "(e: t)" = programmer explicitly tells us that the type of e should be t.

Examples:

- (1+2 : Nat)
- (\x. x : Nat -> Nat)
- (\x. x : Bool -> Bool)