Propositional-Logic Typing

Developing a Logic System for Type Checking

Florian Schrögendorfer, florian.schroegendorfer@gmail.com

Propositional-Logic Typing

based on correspondence between types and propositions

uses propositional calculus for type checking programs

forthcoming development based on *intuitionistic type theory*

we start by defining a simple logic consisting of:

- a syntax definition for building well-formed formulas (wffs)
- a set of axioms and inference rules for proving wffs
- a notion of proof

A Simple Formal Language

defines sentences of the logic, called well-formed formulas (wffs)

Alphabet

propositional symbols: $P, Q, R, \ldots$

connectives: $\Rightarrow, \wedge$

auxiliary symbols: $\vdash, (,)$

Grammar

Sequent S	$::=$	$\Gamma \vdash \phi$	...these are the wffs
Context Γ	$::=$	$\phi_1, \phi_2, \ldots, \phi_n$	$n \geq 0$
Proposition ϕ	$::=$	$P \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \Rightarrow \phi_2)$	

read $\Gamma \vdash \phi$ as from assumption Γ, infer conclusion ϕ

Natural Deduction

sequents are the basic units of proof

deductive system given by:

- axioms for defining true propositions within the system
- inference rules to transform true propositions

therefore:

- each connective *op* defined by its set of inference rules
- propositions containing *op* built by introduction rules *op I*
- elimination rules *op E* remove connectives from propositions

Definition: Theorem

A proved sequent of the form $\vdash \phi$ is called a theorem in the logic.

Axiom and Inference Rules

Axiom

$$\Gamma \vdash \phi \qquad \text{if } \phi \in \Gamma \qquad \text{where } \Gamma = \phi_1, \phi_2, \ldots, \phi_n$$

Introduction Rules

$$\wedge I : \frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2} \qquad \Rightarrow I : \frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \Rightarrow \phi_2}$$

Elimination Rules

$$\wedge E_1 : \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_1} \qquad \wedge E_2 : \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_2}$$

$$\Rightarrow E : \frac{\Gamma \vdash \phi_1 \Rightarrow \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}$$

Sequent-Proof Trees

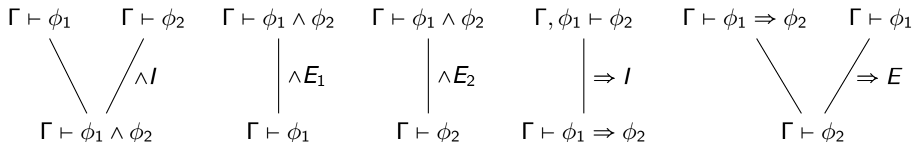
proofs interpreted by the notion of sequent-proof trees

Sequent-Proof Tree

A sequent-proof tree is a tree whose

- root is a sequent, $\Gamma \vdash \phi$
- leaves are axioms
- internal nodes are consequents of inference rules

following hypothetical trees can be built from the defined connectives



Sequent-Proof Trees - Example

$$(Q \wedge R) \Rightarrow S, P \wedge Q, R \vdash P \wedge Q$$

$\wedge E_2$

$$(Q \wedge R) \Rightarrow S, P \wedge Q, R \vdash R \qquad (Q \wedge R) \Rightarrow S, P \wedge Q, R \vdash Q$$

$\wedge I$

$$(Q \wedge R) \Rightarrow S, P \wedge Q, R \vdash Q \wedge R \qquad (Q \wedge R) \Rightarrow S, P \wedge Q, R \vdash (Q \wedge R) \Rightarrow S$$

$\Rightarrow E$

$$(Q \wedge R) \Rightarrow S, P \wedge Q, R \vdash S$$

$\Rightarrow I$

$$(Q \wedge R) \Rightarrow S, P \wedge Q \vdash R \Rightarrow S$$

Natural-Deduction Trees

Natural-Deduction Tree

A natural-deduction tree is a sequent-proof tree with context information omitted from its sequents.

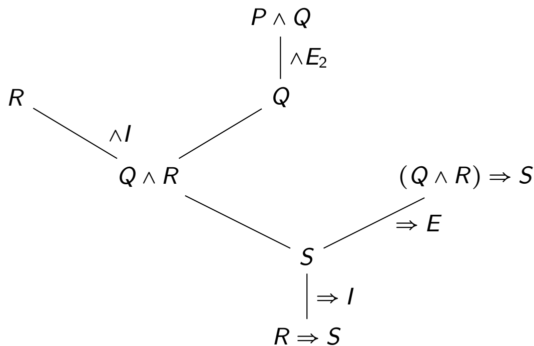
variant of a proof tree convenient for proving a sequent

simpler representation of a proof situation (easy to generate)

see assumptions as propositions whose proof is forthcoming

- can therefore be pasted together, root ϕ_1 to leaf ϕ_2 iff $\phi_1 = \phi_2$
- must be verified as well-formed (by attaching context information)

Natural-Deduction Trees - Example



Linearizing Natural-Deduction Trees

Linearization of Natural-Deduction Tree

The linearized trees produced by inference rules then are as follows:

- Let t_1 and t_2 be linearized trees proving ϕ_1, resp. ϕ_2
- x the hypothetical tree for a local assumption.

$$\begin{aligned}
\wedge I &:\Leftrightarrow (\wedge I \;\; t_1 \;\; t_2) \\
\wedge E_1 &:\Leftrightarrow (\wedge E_1 \;\; t_1) \\
\wedge E_2 &:\Leftrightarrow (\wedge E_2 \;\; t_2) \\
\Rightarrow I &:\Leftrightarrow (\Rightarrow I \;\; (x \in \phi_1) \;\; t_2) \\
\Rightarrow E &:\Leftrightarrow (\Rightarrow E \;\; t_1 \;\; t_2)
\end{aligned}$$

Linearization of the previous Tree

$$(\Rightarrow I \;\; (x \in R)\, (\Rightarrow E \;\; ((Q \wedge R) \Rightarrow S)\, (\wedge I \;\; (\wedge E_2 \;\; (P \wedge Q))\; x)))$$

Heyting Interpretation

Heyting Interpretation

Interpret natural-deduction trees as expressions in lambda calculus.

- proof of $\phi_1 \wedge \phi_2$ is a pair of a proofs for ϕ_1 resp. ϕ_2
- proof of $\phi_1 \Rightarrow \phi_2$ is a function mapping proofs of ϕ_1 into proofs of ϕ_2

transformation to lambda calculus by purely syntactic reformatting:

- $(\wedge I \;\; t_1 \;\; t_2) \mapsto (t_1, t_2)$ ...an ordered pair
- $(\wedge E_1 \;\; t) \mapsto \mathit{fst}\ t$ ...an indexing operation on a pair
- $(\wedge E_2 \;\; t) \mapsto \mathit{snd}\ t$ ...an indexing operation on a pair
- $(\Rightarrow I \;\; (x \in \phi)\ e) \mapsto \lambda x \in \phi.e$ ...a lambda abstraction
- $(\Rightarrow E \;\; (x \in \phi)\ e) \mapsto t_1 \cdot t_2$ ...the application of lambda abstraction

Lambda Expression corresponding to the previous Tree

$$\lambda\, x \in R.\, ((Q \wedge R) \Rightarrow S) \cdot (\mathit{snd}(P \wedge Q), x)$$

Viewing Programs as Proofs

propositional logic vs. lambda calculus

language of natural-deduction trees $\Rightarrow$ typed lambda calculus

- logical connective $\Rightarrow$ corresponds to function-type constructor $\rightarrow$
- logical connective $\wedge$ corresponds to product-type constructor $\times$

Curry-Howard Isomorphism

A proof of $\Gamma \vdash \phi$ is a program of type ϕ within type assignment Γ.

Justifies use of following synonyms:

- proposition $\Leftrightarrow$ type
- proof $\Leftrightarrow$ program
- natural-deduction tree $\Leftrightarrow$ expression

Typing Rules

Judgement

A judgement is a sequent of the form $\Gamma \vdash p \in \phi$, where Γ is a list of items $x_i \in \phi$, such that no identifier x_i appears twice in Γ.

Read the judgment $\Gamma \vdash p \in \phi$ as:

- within context Γ, p is a proof of ϕ, or
- within type assignment Γ, p is a program of type ϕ

identifiers x_i represent hypothetical proof trees for local assumptions

correspondence to typing judgments in the simply typed lambda calculus

viewed as typing rules, the inference rules built from judgements attach context and typing information to natural-deduction trees

Rewritten Inference Rules

Axiom

$$\Gamma \vdash x \in \phi \qquad \text{if } (x \in \phi) \in \Gamma$$

Introduction Rules

$$\wedge I : \frac{\Gamma \vdash t_1 \in \phi_1 \quad \Gamma \vdash t_2 \in \phi_2}{\Gamma \vdash (t_1, t_2) \in \phi_1 \wedge \phi_2} \qquad \Rightarrow I : \frac{\Gamma, x \in \phi_1 \vdash t \in \phi_2}{\Gamma \vdash \lambda x \in \phi_1 . t \in \phi_1 \Rightarrow \phi_2}$$

Elimination Rules

$$\wedge E_1 : \frac{\Gamma \vdash t \in \phi_1 \wedge \phi_2}{\Gamma \vdash \mathit{fst}\ t \in \phi_1} \qquad \wedge E_2 : \frac{\Gamma \vdash t \in \phi_1 \wedge \phi_2}{\Gamma \vdash \mathit{snd}\ t \in \phi_2}$$

$$\Rightarrow E : \frac{\Gamma \vdash t_1 \in \phi_1 \Rightarrow \phi_2 \quad \Gamma \vdash t_2 \in \phi_1}{\Gamma \vdash t_1 \cdot t_2 \in \phi_2}$$

Judgment-Proof Tree - Example

$$a_1 \in (Q \wedge R) \Rightarrow S, a_2 \in P \wedge Q, x \in R \vdash a_2 \in P \wedge Q$$

$\wedge E_2$

$$a_1 \in (Q \wedge R) \Rightarrow S, a_2 \in P \wedge Q, x \in R \vdash x \in R \qquad a_1 \in (Q \wedge R) \Rightarrow S, a_2 \in P \wedge Q, x \in R \vdash snd\ a_2 \in Q$$

$\wedge I$

$$a_1 \in (Q \wedge R) \Rightarrow S, a_2 \in P \wedge Q, x \in R \vdash (snd\ a_2, x) \in Q \wedge R \qquad \vdash a_1 \in Q \wedge R \Rightarrow S_{\Box}$$

$\Rightarrow E$

$$a_1 \in (Q \wedge R) \Rightarrow S, a_2 \in P \wedge Q, x \in R \vdash a_1 \cdot (snd\ a_2, x) \in S$$

$\Rightarrow I$

$$a_1 \in (Q \wedge R) \Rightarrow S, a_2 \in P \wedge Q \vdash \lambda x \in R.a_1 \cdot (snd\ a_2, x) \in R \Rightarrow S$$

Introducing Primitive Types

previous chapters suggested that basic programming languages contain:

- a core set of primitive types
- extended by lambda abstraction, records etc.

in our core logic, data types are represented by propositions

axioms and inference rules define the data types

introducing primitive types *bool* and *nat* by defining their set of rules

see expr. like $3 + 2$ as natural-deduction trees proving proposition *nat*

Introducing Primitive Types (contd.)

Boolean Primitives

$$\mathit{bool}\ I_1 : \Gamma \vdash \mathit{false} \in \mathit{bool} \qquad \mathit{bool}\ I_2 : \Gamma \vdash \mathit{true} \in \mathit{bool}$$

$$\mathit{bool}\ E : \frac{\Gamma \vdash t_1 \in \mathit{bool} \quad \Gamma \vdash t_2 \in \phi \quad \Gamma \vdash t_3 \in \phi}{\Gamma \vdash \ \mathrm{if}\ t_1\ t_2\ t_3 \in \phi}$$

Natural Numbers

$$\mathit{nat}\ I_n : \Gamma \vdash n \in \mathit{nat}, \ \ \text{for}\ \ n \geq 0$$

$$\mathit{nat}\ E_1 : \frac{\Gamma \vdash n_1 \in \mathit{nat} \quad \Gamma \vdash n_2 \in \mathit{nat}}{\Gamma \vdash n_1 + n_2 \in \mathit{nat}}$$

$$\mathit{nat}\ E_2 : \frac{\Gamma \vdash n_1 \in \mathit{nat} \quad \Gamma \vdash n_2 \in \mathit{nat}}{\Gamma \vdash n_1 = n_2 \in \mathit{bool}}$$

Propositions vs. Types

An Example of the Extended Language

$$\vdash \lambda x \in nat.x + 1 \in nat \Rightarrow nat$$

Relation between the Concepts

thus, following terms can be seen as equivalent:

- constructing a function of type $nat \rightarrow nat$
- proving the proposition $nat \Rightarrow nat$

however, there are infinitely many ways of proving a theorem

analogously, there exist infinitely many programs of type $nat \rightarrow nat$

under this insight, decidability becomes an issue

Decidability

1. For a natural-deduction tree and a context what proposition does the tree proof?
    - yes: the algorithm checks if the natural-deduction tree is well-formed and calculates the proposition using defined set inference rules
    - in a programming language such a tree checker is referred to as type checker
2. Is a given proposition ϕ a theorem?
    - yes: a trivial algorithm would compute ϕ's truth table and check if is a tautology
    - addressed by work on automated theorem proving

Computation Rules

we saw that programs are evaluated by computation (rewriting) rules

computation rules in our setting are proof simplification rules

Proof Simplification Rule

A tree containing the application of an introduction rule $op\,I$, followed by an application of an elimination rule $op\,E$ can be simplified without affecting the sequent the tree proves, assuming an unchanged context.

Example: Proof Tree Simplification

$$\dfrac{\dfrac{\overset{e_1}{\phi_1} \quad \overset{e_2}{\phi_2}}{\phi_1 \wedge \phi_2}\,\wedge I}{\phi_1}\,\wedge E_1 \quad \Rightarrow \quad \overset{e_1}{\phi_1} \quad \Rightarrow$$

$$(\wedge E_1\ (\wedge I\ \ e_1\ e_2)) \to e_1$$

$$\Downarrow$$

$$fst(e_1, e_2) \to e_1$$

Computation Rules (Contd.)

simplification rules interpreted as computation rules in lambda calculus

- as expected under the Heyting interpretation

Computation Rules in our current Logic

$\wedge$: $fst(t_1, t_2) \Rightarrow t_1$ — as the indexing rule for pairs
$snd(t_1, t_2) \Rightarrow t_2$

$\Rightarrow$: $(\lambda\, x \in \phi_1.\ t_1) \cdot t_2 \Rightarrow [t_2/x]t_1$ — as the β-reduction

bool : if *true* $t_1\ t_2 \Rightarrow t_1$
if *false* $t_1\ t_2 \Rightarrow t_2$

nat : $n_1 + n_2 \Rightarrow n_3$
$n = n \Rightarrow true$
$n_1 = n_2 \Rightarrow false$ — where n_1 and n_2 are different numerals.

Extending the Logic

a standard propositional logic also contains disjunction and negation

- introduction of the disjunction $\vee$ follows the usual pattern
- negation is introduced as a logic constant: the nullary connective $\perp$

Extended Grammar

$$\phi \quad ::= \quad \cdots \mid (\phi_1 \vee \phi_2) \mid \perp$$

Inference Rules

$$\vee I_1 : \frac{\Gamma \vdash \phi_1}{\Gamma \vdash \phi_1 \vee \phi_2} \qquad \vee I_2 : \frac{\Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \vee \phi_2}$$

$$\vee E : \frac{\Gamma \vdash \phi_1 \vee \phi_2 \quad \Gamma, \phi_1 \vdash \phi_3 \quad \Gamma, \phi_2 \vdash \phi_3}{\Gamma \vdash \phi_3} \qquad \perp E : \frac{\Gamma \vdash \perp}{\Gamma \vdash \phi}$$

Extending the Logic (Contd.)

Heyting Interpretation and Linearization

Heyting interpretation of disjunction and falsehood:

- proof of $\phi_1 \vee \phi_2$ is a proof of ϕ_1, labeled *inl* or a proof of ϕ_2 labeled by *inr*
- proofs of $\perp$ do not exist

Lambda expressions of linearized trees for the new connectives:

- $(\vee I_1^{\phi_1 \vee \phi_2}\ \ t) \mapsto inl_{\phi_1 \vee \phi_2} t$
- $(\vee I_2^{\phi_1 \vee \phi_2}\ \ t) \mapsto inr_{\phi_1 \vee \phi_2} t$
- $(\vee E\ t_1\ (x \in \phi_1)\ t_2\ (y \in \phi_2)\ t_3) \mapsto cases\ t_1\ of\ isl(x \in \phi_1).\ t_2\ [\!]\ isr(y \in \phi_2).\ t_3$
- $(\perp E^{\phi}\ \ t) \mapsto abort_{\phi}\ t$

Computation Rules

$$\vee : \quad \begin{array}{l} cases\ (inl\ t_1)\ of\ isl\ (x \in \phi_1).\ t_2\ [\!]\ isr\ (y \in \phi_2).\ t_3 \Rightarrow [t_1/x]t_2 \\ cases\ (inr\ t_1)\ of\ isl\ (x \in \phi_1).\ t_2\ [\!]\ isr\ (y \in \phi_2).\ t_3 \Rightarrow [t_1/y]t_3 \end{array}$$

Negation and Falsehood

Negation is introduced in terms of falsehood.

read $\neg\phi$ as an abbreviation for $\phi \Rightarrow \perp$

we're only concerned with consistent intuitionistic logics

omitting an introduction rule for falsehood keeps logic consistent

Consistent Propositional Logic

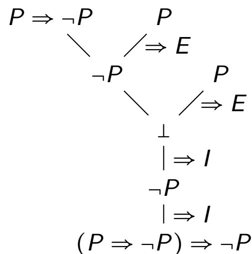
A logic is *consistent* if it is impossible to proof $\vdash \perp$, or equivalently $\vdash \phi$ as well as $\vdash \neg\phi$ for any proposition ϕ.

Consistent Context

A context Γ is consistent if it is impossible to proof $\Gamma \vdash \perp$.

therefore negation in our system is reduced to falsehood

Negation and Falsehood - Examples



Reasoning about Negation (using rules for $\Rightarrow I$ and $\Rightarrow E$)

observe: applying $\Rightarrow I$ on a proposition $\neg\phi$ derives following rule

$$\frac{\Gamma, \phi \vdash \bot}{\Gamma \vdash \phi \Rightarrow \bot}$$

corresponds to a weak version of proof by contradiction under context Γ

Summary: Programming-Languages as Logic Systems

recall that the terms program and proof are synonymous

structuring constructs defined by rules for logical connectives

values created by introduction rules, operators by elimination rules

computation rules define operators semantics on values of a type

core data types are the primitive propositions

previous examples showed development of basic functional language

but also imperative languages may be defined this way

The Core Imperative Language

previously defined rules for core data types ($bool$, nat) still hold

introduction of connectives follows typing rules of the core language

Judgement Rules for Storing Numerals

$$natloc\ I : \Gamma \vdash n \geq 0 \in natloc \qquad store\ I_1 : \Gamma \vdash nil \in store$$

$$store\ I_2 : \frac{\Gamma \vdash l \in natloc \quad \Gamma \vdash n \in nat \quad \Gamma \vdash s \in store}{\Gamma \vdash update\ \ l\ n\ s \in store}$$

$$store\ E : \frac{\Gamma \vdash l \in natloc \quad \Gamma \vdash s \in store}{\Gamma \vdash lookup\ \ l\ s \in nat}$$

Computation Rules for Storing Numerals

$$\begin{aligned} store: \quad & lookup\ \ l\ nil \Rightarrow 0 \\ & lookup\ \ l(update\ \ l\ n\ s) \Rightarrow n \\ & lookup\ \ l(update\ \ m\ n\ s) \Rightarrow lookup\ \ l\ s \qquad \text{if } l \neq m \end{aligned}$$

The Core Imperative Language (Contd.)

Rules for Expressions of Type *natexp*

$$natexp\ I_1 : \frac{\Gamma \vdash N \in nat}{\Gamma \vdash N \in natexp} \qquad natexp\ I_2 : \frac{\Gamma \vdash E_1 \in natexp \quad \Gamma \vdash E_2 \in natexp}{\Gamma \vdash E_1 + E_2 \in natexp}$$

$$natexp\ E : \frac{\Gamma \vdash E \in natexp \quad \Gamma \vdash s \in store}{\Gamma \vdash (E\ s) \in nat}$$

Rules for Commands of Type *comm*

$$comm\ I_1 : \frac{\Gamma \vdash L \in natloc \quad \Gamma \vdash E \in natexp}{\Gamma \vdash L := E \in comm}$$

$$comm\ I_2 : \frac{\Gamma \vdash C_1 \in comm \quad \Gamma \vdash C_2 \in comm}{\Gamma \vdash C_1; C_2 \in comm}$$

$$comm\ E : \frac{\Gamma \vdash C \in comm \quad \Gamma \vdash s \in store}{\Gamma \vdash (C\ s) \in store}$$

The Core Imperative Language (Contd.)

computation rules are built from the rewriting rules of lambda calculus

usual pattern of operators on canonical exp. created by elimination rule

Computation Rules

$$
\begin{aligned}
\textit{comm}: \quad & (L := E\ s) \Rightarrow \textit{update}\ L (E\ s) s \\
& (C_1; C_2) \Rightarrow (C_1 (C_2\ s)) \\
& (\text{if } E \text{ then } C_1 \text{ else } C_2 \text{ fi } s) \Rightarrow \textit{if}\ (E\ s)\ (C_1\ s)\ (C_2\ s) \\
& (\text{while } E \text{ do } C \text{ od } s) \Rightarrow \textit{if}\ (E\ s)(\text{while } E \text{ do } C \text{ od } (C\ s))\ s \\
\textit{natexp}: \quad & (N\ s) \Rightarrow N \\
& (@L\ s) \Rightarrow \textit{lookup}\ \ L\ s \\
& (E_1 + E_2) \Rightarrow (E_1\ s) + (E_2\ s) \\
& (\neg E\ s) \Rightarrow \textit{if}\ \ (E\ s) \textit{false true}
\end{aligned}
$$

Le Fin

Thanks for your attention!