

TYPED PREDICATE LOGIC

Logic in Computer Science

OUTLINE

1. [A Logical Framework](#)
2. [Formalisation](#)
3. [Typed Predicate Logic](#) (TPL)
4. [Type Theories](#)
5. [Models](#)
6. [Definitions in TPL](#)
7. [Generalised Computability](#)
8. [Polymorphism](#)
9. [The Type of Schema](#)
10. [Subtypes](#)
11. [A Theory of Algorithms](#)
12. [Recursive Definitions](#)
13. [Axiomatic Definitions](#)
14. [Correctness and Computability](#)
15. [Answers to Silly Questions](#)
16. [Answers to not so Silly ones](#)

A LOGICAL FRAMEWORK

The Impact of Computer Science

501 new logical systems and frameworks have entered the literature in the last 50 years.

1. New applications of existing logical systems.
2. New extensions of existing logical systems.
3. New Systems.

Existing Logical Systems

- Set Theory
- Constructive Type Theory and its extensions
- Higher Order Logic
- Second order Lambda calculus
- Classical and Intuitionistic logic
- Modal, Temporal and Deontic logic
- Free and Partial logic
- Many valued Logic
- Categorical logic...

Have all been used, extended and modified.

New Systems

- Sub structural logics
- Logics of state change. Dynamic Logic
- Logics of events and processes
- Logics of termination and correctness
- Constructive and classical theories of Operations and Types
- Calculus of Constructions
- Description Logic
- Domain Logic
- Temporal Logics
- [Logical Systems](#)

Applications

- The analysis of programs
- The articulation of notions of state and process
- The expression of termination
- The representation of time and events in computational systems
- Hypothetical reasoning
- Vagueness
- The formalisation of web ontology languages
- The formulation of legal norms in computational law
- Program specification
- The definition of polymorphism,.....

A Unified Approach

- We shall develop a framework in which many of these new computational systems may be expressed and studied in a single logical framework.
- A very flexible system which is not hamstrung by traditional syntax.
- Supports a mix and match policy – chose your logics and glue them together.
- Facilitates a mathematical exploration of theories and their relationships.

FORMALISATION

Theory From Practice

- Practice informs theory. It provides motivation and *informal* systems: the gist of the logicians mill.
- The primary task of the logician is to provide an accurate model of practice.
- This is mathematical modelling with logical systems.

Theoretical Cleansing

- However, many existing Computer Science formalisations suffer from *Over Zealous Pragmatism*: application is the only measure of success. This often results in a lack of attention to fundamental issues.
- Indeed, traditionally, logical systems have served as an aid to conceptual and mathematical analysis.
- They are intended to provide crisp and clean versions of the informal notions used in practice. For this they must be mathematically pleasing and tractable.
- For mathematical purposes, there has to be abstraction and cleansing.

Getting it Wrong

Feferman

Existing formalisations are not always

- Adequate: all concepts of the informal theory/system are covered.
- Faithful: does not go beyond the informal theory.

The representation of computational notions in standard systems of set and type theory are sometimes not even adequate let alone faithful.

Scott

Many existing formalisations suffer from *the hovercraft effect*—the use of high powered mathematics to skim over the surface of the problem without touching it.

- [Home](#)

TYPED PREDICATE LOGIC

Types

- At the basis of our approach is the notion of *type*.
- Type theory has its origins in the philosophical notions of *category* and *sort*.
- Frege and Russell introduced *type constructors* e.g. Simple and Ramified theories.
- Sorts and Relative Identity: Geach, Quine, Wiggins and the logician Martin-Löf.
- Type theory has become a mathematical discipline via mathematical logic (see Turner 2001) and more recently via
- Theoretical computer science: rich development of theories of types including dependent types, polymorphic types, generic, abstract data types, universal types etc.
- Type checking replaces and generalises the traditional role of grammar
- [Home](#)
- Different to *institutions*: model theoretic approach carried out inside category theory.

TPL

- Typed Predicate Logic (**TPL**) is a system of logic in which the notion of *type theory* plays a parallel role to that played by the notion of *first-order theory* in standard first-order predicate logic.
- First appears in my book ([Computable Models](#) Springer, 2009)
- Origins in Curry's Combinatorial Logic and CTT.

FOL AND TPL

- In First Order Logic (FOL) the notions of *operation*, *property*, *relation*, *object* are bare notions. They gain substance in the context of first order theories.
- In TPL, the notions of *operation*, *property*, *relation*, *object* and *type* are bare notions. They gain substance in the context of *Type theories*.

Judgements of TPL

T type

t:T

Φ prop

Φ (true)

⊕ For one of the 4

Contexts and Sequents

1. $c = x_1:T_1, \dots, x_n:T_n \dots$ *type contexts*
2. $\Gamma = x_1:T_1, \dots, \varphi_1, \dots, x_n:T_n, \dots, \varphi_m \dots$ *for general ones*
3. $\Gamma \vDash \Theta$ *Sequents* $c \vDash \Theta$

Rules for the Introduction of Constants, Relations, Functions and Types

$a:A$

the form of constant introduction

$t_1:T_1 \dots t_n:T_n$

$R(t_1, \dots, t_n) \text{ prop}$

the form of relation introduction

$t_1:T_1 \dots t_n:T_n$

$F(t_1, \dots, t_n) :T$

the form of function/operation introduction

$T_1 \text{ type} \dots T_n \text{ type}$

$O(T_1, \dots, T_n) \text{ type}$

the form of type introduction

Grammar Rules

Syntax given by rules. This permits the expression of systems that involve a rich notion of type including dependency and self application.

1. *Propositions* closed under the standard connectives

$$\frac{\Gamma \vDash \varphi \text{ prop} \quad \Gamma \vDash \psi \text{ prop}}{\Gamma \vDash \varphi \wedge \psi \text{ prop}}$$

3. And closed under typed Quantifiers

$$\frac{\Gamma, x:T \vDash \varphi \text{ prop}}{\Gamma \vDash \forall x:T.\varphi \text{ prop}} \qquad \frac{\Gamma, x:T \vDash \varphi \text{ prop}}{\Gamma \vDash \exists x:T.\varphi \text{ prop}}$$

* Note the *dependency* of the judgements

Equality

$$\frac{\Gamma \vDash a:T \quad \Gamma \vDash b:T}{\Gamma \vDash a=_\tau b \text{ prop}} \quad \frac{\Gamma \vDash a=_\tau b \quad \Gamma \vDash \phi[a]}{\Gamma \vDash \phi[b]}$$

$$\frac{\Gamma \vDash a:T}{\Gamma \vDash a=_\tau a}$$

Logical Rules

$$\Gamma \vDash \varphi \quad \Gamma \vDash \psi$$

$$\Gamma \vDash \varphi \wedge \psi$$
$$\Gamma \vDash \varphi \quad \Gamma \vDash \psi \text{ prop}$$

$$\Gamma \vDash \varphi \vee \psi$$
$$\Gamma, \varphi \vDash \psi$$

$$\Gamma \vDash \varphi \rightarrow \psi$$
$$\Gamma \vDash \varphi \quad \Gamma \vDash \varphi \rightarrow \psi$$

$$\Gamma \vDash \psi$$
$$\Gamma, x:T \vDash \psi$$

$$\Gamma \vDash \forall x:T. \psi$$
$$\Gamma \vDash \forall x:T. \psi \quad \Gamma \vDash t:T$$

$$\Gamma \vDash \psi[t/x]$$

COHERENCE

Theorem

1. *If $\Gamma \vDash t:T$ then $\Gamma \vDash T$ type*
2. *If $\Gamma \vDash \psi$ then $\Gamma \vDash \psi$ prop*
3. *If $\Gamma, x:T, \Delta \vDash \Theta$ then $\Gamma \vDash T$ type*
4. *If $\Gamma, \psi, \Delta \vDash \Theta$ then $\Gamma \vDash \psi$ prop*

Proof *By induction on the derivations: rule by rule argument.* •

- *Compare with the older Curry systems of logical anarchy.*
- [Home](#)

TYPE THEORIES

Theories in FOL

- Relations, functions and constants are governed by axiomatic conditions expressed in the language of FOL
- A single domain of discourse
- Examples include
 - Group theory
 - Ring theory
 - ZF Set theory
 -

Theories in TPL

- Generalisation of theories in FOL
- Axiomatic conditions governing operations, properties, relations, constants and their types.
- Best described as *theories of types* with their associated relations and operations etc.

Baby Numbers

\mathbf{N}_0 N type

\mathbf{N}_1 $0 : N$ \mathbf{N}_2 $\frac{a : N}{a^+ : N}$

\mathbf{N}_3 $\frac{\phi[0] \quad x : N, \phi[x] \vdash \phi[x^+]}{x : N \vdash \phi[x]}$

\mathbf{N}_4 $\frac{a : N}{a^+ \neq 0}$ \mathbf{N}_5 $\frac{a^+ =_N b^+}{a =_N b}$

Cartesian Products

T type S type

$T \otimes S$ type

$t:T$ $s:S$

$(t, s):T \otimes S$

$t:T \otimes S$

$!t:T$

$t:T$ $s:T$

$! (t, s) = t$

Typed Set Theory

$$\mathbf{S}_0 \quad \frac{T \text{ type}}{Set(T) \text{ type}}$$

$$\mathbf{S}_1 \quad \frac{T \text{ type}}{\emptyset_T : Set(T)}$$

$$\mathbf{S}_2 \quad \frac{a : T \quad b : Set(T)}{a \otimes_T b : Set(T)}$$

$$\mathbf{S}_3 \quad \frac{\phi[\emptyset] \quad \forall x : T \cdot \forall y : Set(T) \cdot \phi[y] \rightarrow \phi[x \otimes_T y]}{\forall x : Set(T) \cdot \phi[x]}$$

$$\mathbf{S}_4 \quad \frac{a : T \quad b : Set(T)}{a \otimes (a \otimes b) = a \otimes b}$$

$$\mathbf{S}_5 \quad \frac{a : T \quad b : T \quad c : Set(T)}{a \otimes (b \otimes c) = b \otimes (a \otimes c)}$$

$$\mathbf{S}_6 \quad \frac{a : T \quad b : Set(T)}{a \in_T b \text{ prop}}$$

$$\mathbf{S}_7 \quad \frac{a : T \quad b : Set(T)}{a \in a \otimes b}$$

$$\mathbf{S}_8 \quad \frac{a \in c \quad b : T \quad c : Set(T)}{a \in b \otimes c}$$

$$\mathbf{S}_9 \quad \frac{a \in \emptyset_T}{\Omega}$$

$$\mathbf{S}_{10} \quad \frac{a \in b \otimes c}{a = b \vee a \in c}$$

The Simple Typed Lambda Calculus

$$\frac{S \text{ type} \quad T \text{ type}}{\text{-----}} \\ T \Rightarrow S \text{ type}$$

$$\frac{x:T \vDash t:S}{\text{-----}} \quad \frac{f:T \Rightarrow S \quad t:T}{\text{-----}} \\ \lambda x:T.t : T \Rightarrow S \quad ft:S$$

$$\frac{x:T \vDash t:S \quad s:T}{\text{-----}} \\ (\lambda x:T.t)s = t[s/x]$$

Theory Notation

$$\text{Th}(B_1, \dots, B_n, O_1, \dots, O_m)$$

Where

B_1, \dots, B_n are basic types

O_1, \dots, O_m are type constructors.

Examples

1. $\text{CST} = \text{Th}(\mathbb{N}, \otimes, \text{Set})$

2. $\text{STLC} = \text{Th}(\mathbb{N}, \otimes, \Rightarrow)$

Grammar Independence

Definition

A Theory Th is Grammar Independent if

1. $\Gamma \vDash_{Th} \psi$ prop then $c_r \vDash_{TTh} \psi$ prop
2. $\Gamma \vDash_{Th} T$ type then $c_r \vDash_{TTh} T$ type
3. $\Gamma \vDash_{Th} t:T$ then $c_r \vDash_{TTh} t:T$

Where TTh is the theory with rules that have only grammatical premises and conclusions.

Theorem CST is a grammar independent theory

Theorem STLC is a grammar independent theory

Not always so.

1. Is every theory a conservative extension of a grammar independent theory?
2. Is the underlying type system decidable?

The Monadic π -Calculus in TPL

$p:\text{Process}$ $q:\text{Process}$

$p\parallel q: \text{Process}$

$p:\text{Process}$ $q:\text{Process}$

$p\parallel q \equiv q\parallel p$

Events

$$V_1 \quad \mathbf{E} \text{ type} \qquad V_2 \quad \frac{e_1 : \mathbf{E} \quad e_2 : \mathbf{E}}{e_1 \prec e_2 \text{ prop}}$$

$$V_3 \quad \frac{e_1 : \mathbf{E} \quad e_2 : \mathbf{E}}{e_1 \circ e_2 \text{ prop}} \qquad V_4 \quad \frac{e_1 \prec e_2}{\neg(e_2 \prec e_1)}$$

$$V_5 \quad \frac{e_1 \prec e_2 \quad e_2 \prec e_3}{e_1 \prec e_3} \qquad V_6 \quad \frac{e_1 \circ e_2}{e_2 \circ e_1}$$

$$V_7 \quad e_1 \circ e_1 \qquad V_8 \quad \frac{e_1 \prec e_2}{\neg(e_1 \circ e_2)}$$

$$V_9 \quad \frac{e_1 \prec e_2 \quad e_2 \circ e_3 \quad e_3 \prec e_4}{e_1 \prec e_4}$$

CSP: A Theory of Events and Processes

$e:\text{Event}$ $p:\text{Process}$

----- prefix

$e \rightarrow p: \text{Process}$

$p:\text{Process}$ $q:\text{Process}$

----- deterministic Choice

$p \square q: \text{Process}$

$p:\text{Process}$ $q:\text{Process}$

----- nondeterministic choice

$p \sqcap q: \text{Process}$

$p:\text{Process}$ $q:\text{Process}$

----- Interleaving

$p !! q: \text{Process}$

Cardelli's ζ -Calculus

$$m_1:M \dots m_n:M$$

$$[l_1=m_1, \dots, l_n=m_n] : \text{Object}$$
 $o:O$ $m:M$ $e:V$

 $o[l \Leftarrow m] : O$ $\zeta x \cdot e : M$

Constructive Type Theory

T type S type

$T \oplus S$ type

$a:T$ $b:T$

$I[T,a,b]$ type

$x:T \vDash S$ type

$\Sigma x:T.S$ type

$x:T \vDash S$ type

$\Pi x:T.S$ type

Axiomatic Approach

- Many of the current theories of CS can be captured in this way i.e., with the appropriate theory of types.
- Direct axiomatisation rather than representation in another theory.
- TPL allows a mix and match policy – chose your type constructors and glue them together.
- Type theories may be garnered from programming languages or TCS or..
- Next 700 specification languages: explore combinations- more later

Relationships

Language Extension: *all legitimate expressions of one language are legitimate expressions of the next.*

Conservative: By adding new things and their properties, one cannot deduce anything new about old ones.

Eliminability: Anything said about new things can be reduced to something said about old ones.

Grammatical Extension

Most of the notions of FOL have to be newly defined for TPL.

Let T_1 and T_2 be two theories. T_2 is a Grammatical Extension of T_1 iff

- I. *If $T_1 \models \varphi$ prop then $T_2 \models \varphi$ prop*
- II. *If $T_1 \models T$ type then $T_2 \models T$ type*
- III. *If $T_1 \models t:T$ type then $T_2 \models t:T$*

Conservative Extension

Has to be redefined for TPL.

Let T_2 be a grammatical extension of T_1 . It is a Conservative Extension iff

If $T_1 \models \varphi$ prop then

$T_2 \models \varphi$ implies $T_1 \models \varphi$

Elimination

Let T_2 be a grammatical extension of T_1 . It is a Elimination Extension iff

*If $T_2 \models \varphi$ prop then for some ψ
we have*

$T_1 \models \psi$ prop and $T_2 \models \varphi \leftrightarrow \psi$

Translations

These, and other more complex relationships between theories, are often established by the construction of a translations

$$*:T_2 \Rightarrow T_1$$

* is Sound iff

- I. *If $T_1 \models \varphi$ prop then $T_2 \models \varphi^*$ prop*
- II. *If $T_1 \models T$ type then $T_2 \models T^*$ type*
- III. *If $T_1 \models t:T$ type then $T_2 \models t^*:T^*$*
- IV. *If $T_2 \models \varphi$ then $T_1 \models \varphi^*$*

[Home](#)

Models

Set Theoretic Models

- Simple types are modelled as sets.
- Relations and functions are set theoretic relations and functions over those sets.
- Soundness and Completeness - standard arguments for general case.
- Some complications with dependency

Category Theoretic Models

- Use the type-category relationship.
- E.g. Typed Lambda calculus and Cartesian closed categories
- Each type theory requires a different theory of categories.

Recursive Models

- Terms modelled as numbers
- Equality to be decidable
- The sigma relations, operations and types are modelled as codes of RE sets/relations
- All our theories have a recursive model: a test for the computational legitimacy of a theory. Why?
- Proof theoretically very weak.
- For complexity reasons one might wish to design theories that are interpretable inside primitive recursive or even polynomial arithmetic. One then has to pay attention to induction principles. Proofs are hard ; lots of techniques required. See Feferman 95 and Turner 96.
- [Home](#)

DEFINITIONS IN TPL

Definitions

Definitions introduce new named

1. Relations
2. Operations
3. Objects
4. Types

Definitions

Within TPL such definitions take the following form.

Where

$$x_1:T_1, \dots, x_n:T_n \models \phi[x_1, \dots, x_n] \text{ prop}$$

We may (conservatively) introduce a new relation

$$R \triangleq [x_1:T_1, \dots, x_n:T_n . \phi[x_1, \dots, x_n]]$$

Schema Notation

R

$x_1:T_1, \dots, x_n:T_n$

$\phi[x_1, \dots, x_n]$

Axiom Schema

- This is governed by

$$\forall x_1:T_1, \dots, x_n:T_n. R(x_1, \dots, x_n) \leftrightarrow \phi[x_1, \dots, x_n]$$

- *The addition of such a relation results in new theory $TPL_{R..}$*
- Applies to any theory of types.

Set Union

Set Union

$x:\text{Set}(N), y:\text{Set}(N), z:\text{Set}(N)$

$\forall w:N. w \in z \leftrightarrow w \in x \vee w \in y$

Finite Groups

Group

$$G : \text{Set}(U)$$

$$* : G \times G \Rightarrow G$$

$$e \in G$$

$$\forall g \in G \cdot \forall f \in G \cdot \forall h \in G \cdot g * (f * h) = (g * f) * h$$

$$\forall g \in G \cdot e * g = g$$

$$\forall f \in G \cdot \exists g \in G \cdot g * f = e$$

Topological Space in CST

Topological Space

$$F: \text{Set}(\text{Set}(U))$$

$$\emptyset \in F$$

$$U \in F$$

$$\forall G : \text{Set}(\text{Set}(U)) . G \subseteq F \Rightarrow \cup G \in F$$

$$\forall f, g : \text{Set}(U) . f, g \in F \Rightarrow f \cap g \in F$$

Functions and Definite Descriptions

1. If $\forall x:T. \exists! y:S. \phi[x,y]$

Then we may conservatively introduce a new function symbol that satisfies

$$\forall x:T. \forall y:S. F(x)=y \Leftrightarrow \phi[x,y]$$

2. If $\exists! x:T. \beta[x]$

Then we may introduce an object symbol

$$\mu x. \beta[x]$$

Such that $\beta[\mu x. \beta[x]] \wedge \forall x:T. \beta[x] \Rightarrow (x = \mu x. \beta[x])$

Admissible Specifications

In the classical theory of stipulative definitions, there are coherence/consistency constraints. In simplified form:

Conservative: For any prop φ of TPL

$TPL_R \models \varphi$ implies in $TPL \models \varphi$

Elimination: For any prop φ of TPL_R there is a prop ψ of TPL such that $TPL_R \models \varphi \leftrightarrow \psi$

[Home](#)

GENERALISED COMPUTABILITY

Church-Turing

- Applies to numbers but there have been many attempts to generalise to more general mathematical systems.
- For numbers, Turing computable relations are characterised as those relations definable by disjunction, conjunction, existential quantification and bounded quantification – the Σ propositions

Sigma Propositions

- ϕ is Σ iff it is constructed from atomic propositions via \wedge , \vee and \exists .
- ϕ is Δ iff ϕ is Σ and $\neg\phi$ is provable equivalent to a Σ proposition
- Applies to any theory of types but in some theories there maybe additional logical connectives and quantifiers.
- Completeness requires *self reference*

Generalised Computability

- This provides a generalised computability theory that extends Church-Turing computability to arbitrary type theories cast within TPL.
- Justification is that the notions of Σ and Δ in Peano arithmetic are logical characterisations of CT computability and in arbitrary type theory these notions get mapped to their arithmetic cousins.

Research topic: extend the work in recursion theory to arbitrary type theories. Explore the issue of Turing completeness. Here is a start

Recursive and RE Relations

Definition

$$R \Leftrightarrow [x_1:T_1, \dots, x_n:T_n . \phi[x_1, \dots, x_n]]$$

is Recursively Enumerable iff ϕ is Σ . R
is Recursive if ϕ is Δ .

POLYMORPHISM

Types of Polymorphism

- In specification and programming languages many different forms of polymorphism can be found.
- *Implicit*
- *Explicit*
- *Subtype*

Implicit Polymorphism

- *Implicit*: The MIRANDA programming language
- Here programs have no type information attached to them.
- Types are computed via type-inference engines that try and find the most general type.

Subtype Polymorphism

- Is a form of type polymorphism in which program constructs written to operate on elements of the super type can also operate on elements of the subtype.
- If S is a subtype of T , the sub typing relation is often written $S <: T$, to mean that any term of type S can be *safely used in a context where* a term of type T is expected.

$$\frac{S <: T \quad \phi[T] \text{ prop}}{\phi[S] \text{ prop}}$$

Explicit Polymorphism

- Second Order Lambda Calculus
- Abstraction over types.
- $\Pi X.T$
- $\lambda X.t$

Explicit Polymorphism in TPL

Add a universal type U

$$\frac{T \text{ type}}{\text{-----}} \quad \frac{T:U}{\text{-----}}$$

$T:U$ $T \text{ type}$

*Much depends on what class of types is allowed.
Arithmetic model: the set of RE sets is RE.*

Set Union

Set Union

$x:\text{Set}(N), y:\text{Set}(N), z:\text{Set}(N)$

$\forall w:N. w \in z \leftrightarrow w \in x \vee w \in y$

Polymorphic Set Union

Set Union

$u:\mathbf{U},$
 $x:\text{Set}(u), y:\text{Set}(u), z:\text{Set}(u)$

$$\forall w:u. w \in z \leftrightarrow w \in x \vee w \in y$$

THE TYPE OF SCHEMA

The Type of Schema

- So far Definitions are part of the metatheory
- One may include them as a new type
- We shall call Σ definitions *schema*

The Theory $\mathbf{SC}(\Sigma)$

$$\mathbf{Sc}_0 \frac{T \text{ type}}{\mathbf{S}(T) \text{ type}}$$

$$\mathbf{Sc}_1 \frac{x : T \vdash \phi \text{ prop}}{[x : T \mid \phi] : \mathbf{S}(T)}$$

$$\mathbf{Sc}_2 \frac{s : \mathbf{S}(T) \quad t : T}{s(t) \text{ prop}}$$

$$\mathbf{Sc}_3 \frac{x : T \vdash \phi \text{ prop} \quad t : T \quad \phi[t/x]}{[x : T \mid \phi](t)}$$

$$\mathbf{Sc}_4 \frac{x : T \vdash \phi \text{ prop} \quad t : T \quad [x : T \mid \phi](t)}{\phi[t/x]}$$

Operations on Schema

Example 1

$$\begin{array}{c} \cup_U \\ \hline f : \mathbf{S}(U), g : \mathbf{S}(U), h : \mathbf{S}(U) \\ \hline h = [x : U \mid fx \vee gx] \end{array}$$

Example 2 (Generalized Union)

$$\begin{array}{c} \cup_U \\ \hline f : \mathbf{S}(\mathbf{S}(U)), g : \mathbf{S}(U) \\ \hline g = [x : T \mid \exists z : \mathbf{S}(U) \cdot f(z) \wedge z(x)] \end{array}$$

Example 3 (Schema Product)

$$\begin{array}{c} \otimes \\ \hline u : \mathbf{U}, v : \mathbf{U} \\ f : \mathbf{S}(u), g : \mathbf{S}(v), h : \mathbf{S}(u \otimes v) \\ \hline h = [x : u \otimes v \mid f(x_1) \wedge g(x_2)] \end{array}$$

SUBTYPES

Why Subtypes

- Expressive power
- Allows more fine grained types: more total functions
- Pre-conditions in definitions

Subtypes

$$\text{Sep}_0 \quad \frac{x : T \vdash \phi \text{ prop}}{\{x : T \mid \phi\} \text{ type}}$$

$$\text{Sep}_1 \quad \frac{x : T \vdash \phi \text{ prop} \quad a : T \quad \phi[a/x]}{a : \{x : T \mid \phi\}}$$

$$\text{Sep}_2 \quad \frac{a : \{x : T \mid \phi\}}{a : T}$$

$$\text{Sep}_3 \quad \frac{a : \{x : T \mid \phi\}}{\phi[a/x]}$$

Example

Example 1 *In the theory $\text{Th}(\mathbf{N}, \text{DP}, \text{Set}, \text{Sep})$, the **type of maps**, a subtype of the type of set-theoretic relations, is defined as*

$$A \rightarrow_m B \triangleq \{z : \text{Set}(A \otimes B) \cdot \forall x \in z \cdot \forall y \in z \cdot x_1 = y_1 \rightarrow x_2 = y_2\}$$

*Similarly, the type of **injective maps** may be defined as a subtype of the type of maps.*

$$A \twoheadrightarrow_m B \triangleq \{z : A \rightarrow_m B \cdot \forall x \in z \cdot \forall y \in z \cdot x_2 = y_2 \rightarrow x_1 = y_1\}$$

Example

Example 2 *Suppose that*

$$y : A, z : B \vdash \psi[y, z] \text{ prop}$$

*We specify a **Collect** operator as follows.*

Collect

$$x? : \text{Set}(\{y : A \cdot \exists z : B \cdot \psi[y, z]\}), w! : \text{Set}(B)$$

$$\forall y \in x \cdot \exists z \in w \cdot \psi[y, z]$$

Functions with Subtypes

Example 1 (Strong Collection) *Suppose that*

$$y : A, z : B \vdash \psi[y, z] \text{ prop}$$

We may then specify

Strongcollect

$$x? : \text{Set}(\{y : A \cdot \exists z : B \cdot \psi[y, z]\}), w! : \text{Set}(B)$$

$$\forall u \in x \cdot \exists v \in w \cdot \psi[u, v] \wedge \forall z \in w \cdot \exists y \in x \cdot \psi[y, z]$$

We can now strengthen the last result.

Corollary 2 *Strong collection is total and functional.*

Systems with Subtypes

- Membership in types depends upon the truth of propositions – SEP₁
- Subtypes destroy grammar independence
- However:

The Addition of Subtypes is Conservative (very often)

Each type proposition and T is translated (Via *)
Types break into 2 parts

- i. A type T_+
- ii. A predicate T_- over T_+

Such that

Theorem: *If $\Gamma \models t:T$ then $\Gamma^* \models t^*:T^*$ and $T_-(t^*)$*

The actual construction depends upon the type constructors of the theory – i.e. It has to be established for each theory – can we do better?

A THEORY OF ALGORTHIMS

Programs and their Types

$l:L$ $n:N$

$l:=n: \text{Program}$

$B: \text{Bool}$ $P: \text{Program}$ $Q: \text{Program}$

$\text{if } B \text{ then } P \text{ else } Q: \text{Program}$

$P: \text{Program}$ $Q: \text{Program}$

$P;Q: \text{Program}$

$B: \text{Bool}$ $P: \text{Program}$

$\text{While } B \text{ do } P: \text{Program}$

Definitional Equality

$$\frac{P=P' \quad Q=Q'}{P;Q = P';Q'}$$

$$\frac{P=P' \quad B=B'}{\text{While } B \text{ do } P = \text{While } B' \text{ do } P'}$$

- Etc.
- *What is syntax and what is semantics* has to do with equality

States

$s:\text{State}$ $n:N$ $l:L$

$\text{Update}(s, l, n):\text{State}$

$s:\text{State}$ $l:L$

$\text{Lookup}(s, l):N$

$s:\text{State}$ $n:N$ $l:L$

$\text{Lookup}(\text{Update}(s, l, n), l) =_N n$

$s:\text{State}$ $n:N$ $l':L$ $l \neq_N l'$

$\text{Lookup}(\text{Update}(s, l, n), l') =_N \text{Lookup}(s, l')$

Evaluation

$p:\text{Program}$ $s:\text{State}$ $s':\text{State}$

$\langle p, s \rangle \Downarrow s' \text{ Prop}$

Evaluating p in state s terminates in s' .

Assignment

$$\langle E, s \rangle \Downarrow v$$

$$\langle x:=E, s \rangle \Downarrow \mathbf{Update}(s, x, v)$$

The meaning of assignment is determined by the operations of the machine.

Sequencing and Conditionals

$$\frac{\langle P, s \rangle \Downarrow s' \quad \langle Q, s' \rangle \Downarrow s''}{\langle P; Q, s \rangle \Downarrow s''}$$

$$\frac{\langle B, s \rangle \Downarrow \mathbf{true} \quad \langle P, s \rangle \Downarrow s'}{\langle \mathbf{If } B \mathbf{ do } P \mathbf{ else } Q, s \rangle \Downarrow s'}$$

$$\frac{\langle B, s \rangle \Downarrow \mathbf{false} \quad \langle Q, s \rangle \Downarrow s''}{\langle \mathbf{If } B \mathbf{ do } P \mathbf{ else } Q, s \rangle \Downarrow s''}$$

While

$\langle B, s \rangle \Downarrow \text{true}$ $\langle P, s \rangle \Downarrow s'$ $\langle \text{while } B \text{ do } P, s' \rangle \Downarrow s''$

$\langle \text{while } B \text{ do } P, s \rangle \Downarrow s''$

$\langle B, s \rangle \Downarrow \text{false}$

$\langle \text{while } B \text{ do } P, s \rangle \Downarrow s$

A Theory of Algorithms

- We can now use the machinery of TPL to explore this theory.
- Definitions of termination
- Equality

Termination

Term

p:Program

s:State

$\exists s':State. \langle p, s \rangle \Downarrow s'$

Partial Equality

\approx

p:Program

q:Program

$\forall s_1:State \cdot \forall s_2:State \cdot \langle p, s_1 \rangle \Downarrow s_2 \leftrightarrow \langle q, s_1 \rangle \Downarrow s_2$

Equivalence

i.e., we cannot tell them apart in terms of their extensional behaviour. This is an equivalence relation. Moreover,

- (1) If **true** then P else $Q \simeq P$
- (2) If **false** then P else $Q \simeq Q$
- (3) **while** B **do** $P \simeq$ **If** B **then** $(P; \text{while } B \text{ do } P)$
else skip

Dynamic Logic

Every program defines a modal operator

$$\Box_p \Leftrightarrow [s:\text{States}, s':\text{States} \mid \langle P, s \rangle \Downarrow s']$$

1-step reduction is an alternative

[Home](#)

RECURSIVE DEFINITIONS

The Form of Recursive Definitions

- In the theory **SC**, assume that

$$x : T, f : \mathbf{S}(T) \vdash \phi[f, x] \text{ prop} \quad \mathbf{R}_0$$

where, when f occurs in ϕ , it occurs as a predicate.

- A recursive schema specification has the following form:

$$R \triangleq [x : T \mid \phi[R, x]] \quad \mathbf{Rec}$$

where $\phi[R, x]$ is obtained by replacing every occurrence of f by R .

- This is taken to introduce a new relation symbol that satisfies the following versions of **R₁**, **R₂**, and **R₃**.

$$x : T \vdash R(x) \text{ prop} \quad \mathbf{R}_1$$

$$\forall x : T \cdot \phi[R, x] \rightarrow R(x) \quad \mathbf{R}_2$$

$$\frac{x : T \vdash \theta[x] \text{ prop} \quad \forall x : T \cdot \phi[\theta, x] \rightarrow \theta[x]}{\forall x : T \cdot R(x) \rightarrow \theta[x]} \quad \mathbf{R}_3$$

Recursion operator taken from Gödel's Functionals of finite type

R

$u : \text{type}, x : u, f : S(\mathbf{N} \otimes u \otimes u), y : \mathbf{N}, z : u$

$$y = 0 \wedge z = x$$

\vee

$$y \neq 0 \wedge \exists w : u \cdot R(u, x, f, \text{pred}(y), w) \wedge f(\text{pred}(y), w, z)$$

Reduction

- In the theory with states and recursive definitions
- One can define \Downarrow

Which Theories Support Recursion?

- PA
- Typed Finite Set Theory
- May take them as primitive
- With a universe we can have recursive type definitions.

[Home](#)

AXIOMATIC DEFINITIONS

Group

$$G : \text{Set}(U)$$

$$* : G \times G \Rightarrow G$$

$$e \in G$$

$$\forall g \in G \cdot \forall f \in G \cdot \forall h \in G \cdot g * (f * h) = (g * f) * h$$

$$\forall g \in G \cdot e * g = g$$

$$\forall f \in G \cdot \exists g \in G \cdot g * f = e$$

Monoid

$G: \text{Set}(U)$

$+: \text{Set}(U) \otimes \text{Set}(U) \Rightarrow \text{Set}(U)$

$I: \text{Set}(U)$

$$\forall x:U. I+x=x+I=x$$

$$\forall x,y,z:U. (x+y)+z=x+(y+z)$$

Operations on Definitions

- Ring = Abelian Group Join Monoid

CORRECTNESS AND COMPUTABILITY

Correctness

R is correct relative to S (written $R \prec S$)

iff

I. $\text{Dom}(R) \subseteq \text{Dom}(S)$

II. $\underline{S}(x, y) \rightarrow R(x, y)$

Where $\underline{S} = S \downarrow \text{Dom}(R)$

- Where S is functional II reduces to: $R(x, S(x))$.
It states that S correctly implements R
- R provides the correctness criteria for S

Implementable in Theory

- S is Weakly Implementable iff there exists a Σ -definition R such that $S < R$.
- Halting Problem is not weakly Implementable.

Halting Problem

Halt

p:Program

s:State

b:Bool

$\exists s':State. \langle p, s \rangle \Downarrow s' \rightarrow b=true$

\vee

$b=false$

Incompleteness and Halting

- Can one construct a program that meets this specification?
- Suppose so: then we can show that the halting problem is decidable
- From which we can deduce incompleteness.

Answers to Silly Questions

Why Theory?

Someone who had begun to learn geometry with Euclid, when he had learnt the first theorem, asked Euclid

"What shall I get by learning these things?" Euclid called his slave and said

"Give him threepence since he must make gain out of what he learns"

Is this an appropriate response when applied to the role of logic in computer science?

Answers to Not so Silly Ones