Automated Equational Reasoning in Nondeterministic λ-Calculi Modulo Theories $\mathcal{H}^*$

by
Fritz H. Obermeyer

2009:05:01

Department of Mathematics
Mellon College of Science
Carnegie Mellon University
Pittsburgh, PA

Submitted for partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Abstract

In this thesis I study four extensions of untyped λ-calculi all under the maximally coarse semantics of the theory $\mathcal{H}^*$ (observable equality), and implement a system for reasoning about and storing abstract knowledge expressible in languages with these extensions. The extensions are:

(1) a semilattice operation $\mathbf{J}$, the join w.r.t the Scott ordering;
(2) a random mixture $\mathbf{R}$ for stochastic λ-calculus;
(3) a computational comonad $\langle \mathsf{code}, \mathsf{apply}, \mathsf{eval}, \mathsf{quote}, \{-\}\rangle$ for Gödel codes modulo provable equality; and
(4) a Π^1_1-complete oracle $\mathbf{O}$.

I develop three languages from combinations of these extensions. The syntax of these languages is always simple: each is a finitely generated combinatory algebra. The semantics of these languages are various fragments of Dana Scott's D_∞ models. Although the languages use ideas from the theory of computer programming languages, they have no operational semantics and do not describe programs.

The first language, SKJ , extends combinatory algebra with a join operation, with respect to the information ordering on terms. I show that an interpretation of types-as-closures reflects from D_∞ down to this definable fragment. The main theorem for SKJ is that simple types are definable as closures. The resulting type theory is very rich (ω-order polymorphism, dependent types, a type-of-types, power-types, quotient types) but logically unsound (every type is inhabited). However I demonstrate that this type system provides an expressive interpretation of type-as-symmetry and join-as-ambiguity.

The second language, SKRJ , extends combinatory algebra with a random bit, then with join. I show that SKRJ provides semantics for a monadic λ-calculus for convex sets of probability distributions (CSPDs). This follows from our choice of join distributing over randomness – the opposite of what one would expect from the usual interpretation of join-as-concurrency. I conjecture that a weakened definable-types theorem also holds in SKRJ , and provide some evidence to this effect. I also show that, in case simple types are definable, the monadic CSPD types are polymorphic in the simple definable types.

The third language, SKJO , extends combinatory algebra with join, a comonadic type of codes, and a carefully defined Π^1_1-complete oracle, so that exactly the Δ^1_1 predicates about SKJ -terms are realized as total terms of type $\mathsf{code}\to\mathsf{bool}$. This language is sufficient to represent all of predicative mathematics, and can thus serve to represent virtually any kind of abstract knowledge. As an example, I formulate the statement and proof of termination in Gödel's $\top$ in the language SKJO , but do not formally verify the proof.

The final component of this thesis is a system, Johann, for automated reasoning about equality and order in the above languages. Johann was used to formally verify many of the theorems in this thesis (and even conjecture some simple theorems).

The general design focus is on efficient knowledge representation, rather than proof search strategies. Johann maintains a database of all facts about a set of (say 10k) objects, or terms-modulo-equivalence. The database evolves in time by randomly adding or removing objects. Each time an object is added, the database is saturated with facts using a forward chaining algorithm.

A specific design goal is to be able to run Johann for long periods of time (weeks) and accumulate useful knowledge, subject to limited memory. This requires statistical analysis of a corpus of interest (e.g. the set of problems to be verified in this thesis), statistical search for missing equations (from our Σ^0_1 approximation to a Π^0_2-complete theory), and careful choice of sampling distributions from which to draw objects to add-to and remove-from the database. The (add,remove) pair of distributions is chosen to achieve a *detailed balance* theorem, so that, at steady state, Johann (provably) remembers simple facts relevant to the corpus.

Contents

Acknowledgements

The author is most indebted to Rick Statman and Matthew Szudzik, for years of help in gaining a perspective on λ-calculus and mathematical logic.

Thanks also to Kevin Kelly, Dana Scott, James Cummings, Frank Pfenning, Peter Lumsdaine, Moritz Hardt, Jeremy Avigad, Bob Harper, Steve Awodey, Henk Barendregt, Bob Liebler, and Alexander Hulpke for many fruitful discussions.

Thanks to Karl and Taya for putting up with obsessively bindboggled family. Thanks to Shaz for helping me think from other people's perspectives. Thanks to Matt Kahle for the random discussion back in, like, 1999 that led to the first koan below, and this thesis topic.

And thanks to PJ for help in obtaining the office chair, and thus the foundations of this mathematician.

Fritz Obermeyer,
2009:03:13

Chapter 1

Overview

Koan: If thoughts accumulated onto a structure,
as atoms accumulate to form a crystal,
what would that structure look like?

Koan: If we could store general results of mathematics in a table,
as we remember arithmetic in a multiplication table,
what would that table look like?

1.1 Motivation and Philosophy

Suppose we want to reason about the world by drawing analogies with mathematics, with some fixed universal mathematical structure. But we can only understand a finite approximation of this infinite universal formal structure, e.g. all sets definable with n symbols.

What is the best universal structure to work with? The structure should, like mathematics, be something whose properties are discoverable. It should serve as a repository of scientifically acquired formal knowledge. It should be an ontology of formal systems. There should be no place for uncertainty in the structure itself, only in our finite approximation to it.

An example of such a structure is Gödel's universe of constructible sets L. We can locate within L any other formal system (r.e. theory, model, logic, etc.). We can translate questions about other formal systems to questions about L. More importantly, we can translate knowledge and intuition from various formal systems *via* L to other formal systems.

Suppose we built up a finite approximation to L, say an ontology, whose classes are sets and whose relations between classes are the constructive axioms of set theory. When two sets are proved equal, we ensure that they are represented by a single node in the ontology. We could in principle build all sets definable in $\leq \mathsf{n}$ symbols, construct all formulae expressible in $\leq \mathsf{m}$ symbols, and automatically prove all statements whose proofs use formulae bounded in size by m (*narrow* proofs rather than *short* proofs).

At any stage in growing such a structure, an approximation of L (imagine this process as crystal growth), we would have sets that are provable equal, but only using wider proofs than we currently admit. Thus as proof width increases, some of the ontology's nodes are merged. We also have candidate expressions using fewer than n symbols, but that are not narrowly provably sets (e.g. when constructing a set using choice, we may not yet have proved the condition for existence). Thus as proof width increases, set candidates are accepted or rejected.

Since set equality in L is undecidable, as set size grows, we find increasingly more pairs of sets whose equality is unprovable, and increasingly more set candidates whose existence is unprovable. Are there general reasoning principles whereby we can accumulate statistical evidence for equations and existence decisions? Is there a physical basis for L?

Perhaps counting symbols or the number of axioms employed in constructing a set is not the best measure of complexity. We ought really weigh frequently-used axioms less and seldom-used axioms more.

And what about variable names? Is a set definable with 2 variables simpler than a set definable only with at least 5?

How many symbols does it take to represent a typical mathematical entity, say the set of primes, or the real number π? How many sets must we enumerate (uniformly, i.e. all sets definable with $\leq \mathtt{n}$ symbols for some n) before we come upon the number π? Could a machine automatically enumerate such sets, and accumulate for us mathematical knowledge?

In this thesis we design a universal structure that serve these goals better, and for which it is easier to answer these questions. The structure should have a simple grammar, so that its physical representation is simple, and so expression complexity is easy to analyze. The structure should be *real* in the sense that although we can't finitely axiomatize it, we can look in the real world for statistical evidence partially answering questions about it.[1] More practically, we should be able to gain knowledge about the structure using simple efficient reasoning principles, so that the acquisition of this knowledge can be automated. The structure should be *universal* in the sense that all real formal structures can be found in it.[2] And the structure should be *dense* in the quantitative sense that common "natural" mathematical structures should have simple representations, so that we can practically construct the smallest $\mathtt{n}$ expressions and still see many useful objects.

1.2 Summary

Untyped λ-calculus is the backbone of many theoretical and real-world programming languages. One generally starts with pure untyped λ-calculus and adds things (like numbers, booleans, etc. with reduction rules) and takes things away (like ill-typed terms or non-convergent terms). The advantage of λ-calculus and functional programming over imperative programming languages is the ease of equational reasoning: we can reason about λ-terms —programs— as we reason about numbers: writing equations, solving equations, examining solution sets, etc.

This thesis follows the standard path of adding things to pure untyped λ-calculus, attempting to get as much as possible out of as little extension as possible. Our main theorem (in 3.6) demonstrates that Dana Scott's λ-calculus of types-as-closures (from [Sco76]) can be interpreted in λ-calculus extended with a semilattice operation, the join w.r.t. Scott's information ordering. In that theorem we show that there are λ-join-definable closures for each simple type, so that we can simulate the simply-typed λ calculus within the untyped λ-join-calculus, implementing types as λ-join-terms.

By developing a typed λ-calculus within an untyped system, we can leverage simple equational reasoning principles to address type-conscious verification problems, e.g., typechecking, type-restricted equality, and type-restricted convergence. In particular, we adapt the forward-chaining Todd-Coxeter algorithms from computational group theory to the problem of program verification. The first two chapters have been debugged and partially verified with our verification system Johann, described in Chapter 7.

All of our reasoning is done in $\mathcal{H}^*$ = the coarsest equational theory never identifying a divergent term with a normal form = the theory of Dana Scott's D_∞ and $\mathbb{P}(\omega)$ models. The moderate success of our automated reasoning system provides evidence that $\mathcal{H}^*$ can be of practical significance, despite being logically complicated (Π_2^0-complete) and admitting no operational semantics (you can't run two programs to see if they are the same, modulo $\mathcal{H}^*$).

The λ-join-calculus admits some convenient programming idioms in addition to definable types-as-closures. For example, we show in 3.9 that there are convenient-to-use closures corresponding to problems of various complexity classes, e.g., Π_1^0, Σ_1^0, Π_2^0, Σ_2^0, and differences of Π_2^0 sets. This is in contrast to pure untyped λ-calculus, where equational specifications are more difficult to construct. We also demonstrate a style of using closures to perform type-inference, in 3.7, 3.14, 6.1, and 6.2. Although the join operation

[1]Shapiro ([Sha97]) contrasts such theories, e.g. numbers, about specific real objects with "algebraic" theories about families of objects, e.g. group theory. In the former we have set down finitely much axiom information in the endless process of seeking the true theory of an object; in the latter we categorize all extensions of a particular finite theory, e.g. groups or topological spaces.

[2]Say a structure is *real* if it can be found in the real world, and is *formal*, if it can be found in many places, with computable transformations among these views.

admits no new Curry-Howard corresponding types, it does allow simpler proofs, whereby proofs can be "sketched", and then raised to full proofs via a closure, as in type inference.

After developing and verifying a small library of expressions in λ-join-calculus, we experiment with two further extensions of λ-join-calculus. (But we do not fully implement verifiers for these extensions.)

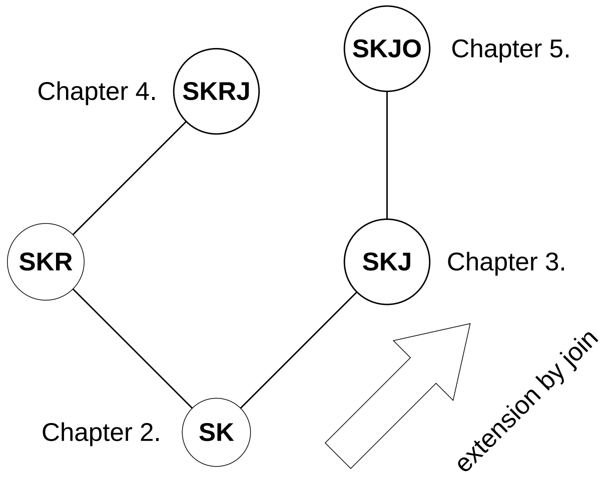

Figure 1.1: From pure untyped λ-calculus SK , we extend to a nondeterministic λ-calculus SKJ , stochastic λ-calculus SKR , a λ-calculus for convex sets of probability distributions SKRJ , and a λ-calculus for hyperarithmetic functions SKJO .

The second extension we study adds randomness to λ-calculus, and then adds join. We attempt but fail to prove that simple types are still definable in this system; however the attempt sheds light on the simpler proof in SKJ , and the nature of join in general. We do prove that the system where randomness distributes over join (i.e., $\mathbf{R}\ x(\mathbf{J}\ x\ y) = \mathbf{J}(\mathbf{R}\ x\ y)(\mathbf{R}\ x\ z)$) provides a language for convex sets of probability distributions (CSPDs), the natural system arising from interval-valued probabilities. We also prove that there is a monadic type system for CSPDs, following Pfenning's type system $\lambda_{\mathbf{O}}$ ([PPT05]). We show that, in case the simple-types theorem does hold in SKRJ , the simple types and the CSPD monad combine gracefully.

Our third extension attempts to capture logically complex statements in the clean verification language of SKJ , by adding a Π^1_1-complete oracle $\mathbf{O}$. By carefully limiting what kinds of questions $\mathbf{O}$ can answer, we are able to extend the problem classes of 3.9 to hyperarithmetic analogs in 5.6. Our main innovation in this extension is a type of Gödel-coded SKJO -terms, modulo the theorem-prover's notion of equality —a sort of provably extensional code type. This code type forms a computational comonad, as studied by Brookes and Geva ([BG92]), and satisfies much better categorical properties than other types of quoted or inert terms. More practically, these coded types allow a nearly 1-1 correspondence between terms and codes for those terms, thus conserving space in our very memory-intensive verification algorithm (though we have not implemented verifier for this fragment).

Finally, we describe in 7 algorithms for probabilistic proof search and statistical analysis, as part of the Johann reasoning system. The first of these algorithms is a randomized database annealing algorithm, based on the Metropolis-Hastings algorithm. This allows us to randomly acquire knowledge in service of a probabilistically specified motive, such as "simplify this term" or "answer this query" or "prove this assertion". Our main theorem shows *detailed balance*: that we can achieve a desired steady state distribution over databases by using a pair of easily-computable sampling distributions, one for adding terms to- and one for removing terms from- the database.

A second algorithm searches for missing equations in our forward-chaining database. This "automated conjecturing" algorithm was instrumental in developing the axioms in the first chapter (see subsections entitled "Theorems conjectured by Johann"). It is tempting to use such an algorithm to automatically add equations to the theory. However, we prove that this is impossible: automated conjecturing can focus attention on interesting questions, but cannot provide probabilistic guesses as to the likelihood of truth.

The final algorithm tunes the probabilistic basis, or dually, the complexity norm with which we define "simplicity". This nonlinear optimization algorithm allows the verifier to assess which λ-terms should be considered basic constants, and locally optimizes their weights. The goal of this optimizer is to decrease the complexity of an example corpus —in our case this thesis. Although the local optimization was crucial in identifying reasonable basis weights for successful proof search, global optimization is provably impossible.

1.3 Future directions

Operational semantics

Although λ-join-calculus modulo $\mathcal{H}^*$ proves to be a succinct language for computable mathematics, it fails as a true programming language, as it lacks an operational semantics. By finding a reasonable operational semantics, we would be able to verify not only mathematics, but also programs.

Already the join operation can be interpreted as concurrency, but this proves difficult in the presence of infinite joins (as in proof search in 3.14). A possibility is fault-tolerant semantics, where joins of threads each compete to find the first total result, and additional results (e.g. error) are discarded thereafter. This would require hard-coding a few datatypes, and restricting joins to domains with totality tests. In this semantics we essentially take a minimal Σ^0_1 fragment from $\mathcal{H}^*$ (exactly what is needed for type definability) and discard the rest.

Additional knowledge representations

The Johann reasoning system does not scale well, with the most expensive inference rules requiring quintic time in size of formula to be verified. Although the constant factor for these inference rules is extremely small, it necessarily limits the size of databases we can build. For example, we can currently build databases with $N = 8000$ obs overnight, but extending to $N = 10000$ obs takes two more days. Indeed it is typically time and not memory that limits the system.

There are two problems to be addressed here. The first is that, since Johann searches for narrow proofs only within the current database, it takes many annealing steps to acquire proof information. That is, to increase the database size by one ob, while maintaining a specified knowledge density (e.g. percent of order relations which are decided), Johann must add and remove many obs in the process of proof search.

The second problem is that about 90% of the theorem-proving time is taken spent enforcing about 8 axiom schemata instances. Of these, the associativity schema (for $\mathbf{B}$, $\mathbf{J}$, and $\mathbf{P}$) is most expensive. In addition to time-cost, associativity is also very space-intensive; e.g. to prove $\mathsf{x}\circ(\mathsf{y}\circ\mathsf{z}) = (\mathsf{x}\circ\mathsf{y})\circ\mathsf{z}$ requires 8 obs: $\mathbf{B}\ \mathsf{x}$, $\mathbf{B}\ \mathsf{y}$, $\mathbf{B}\ \mathsf{y}\ \mathsf{z}$, $\mathbf{B}\ \mathsf{x}\ \mathsf{y}$, $\mathbf{B}(\mathbf{B}\ \mathsf{x}\ \mathsf{y})$, and either $\mathbf{B}\ \mathsf{x}(\mathbf{B}\ \mathsf{y}\ \mathsf{z})$ or $\mathbf{B}(\mathbf{B}\ \mathsf{x}\ \mathsf{y})\mathsf{z}$.

One possible solution to both of these problems is to add new knowledge representations: tables for both composition and join. Since proofs in this augmented database would be narrower, Johann would not need as many annealing steps to achieve a given density. Moreover, the associativity schemata for $\mathbf{B}$ and $\mathbf{J}$ could be re-implemented to use the more appropriate data structures. It is also likely that adding composition to the probabilistic basis (and randomly generating compositions of terms $\mathsf{x}\circ\mathsf{y}$) would be more optimal in simplifying the corpus. This is suggested by the empirical observation that the composition atom $\mathbf{B}$ havs very high weight in the optimized basis found by Johann —$\mathbf{B}$ was used twice as often as any other atom.

An interface for problem-solving

The Johann system is better suited to statistical analyses and term search algorithms than verification tasks. Thus we would like to develop a natural interface to harness this computing power. On example such interface is that of *hole filling* in a language with partially defined terms.

Example 1.3.1. Say you're defining/constructing a mathematical entity, say a function. You already know

- its input- and output- types;
- some facts about- and relating- its inputs and outputs, e.g.
  - some of its symmetries (preserving transformations) and
  - some example inputs and outputs; and
- a rough idea of how to build it.

You sketch out what you know

```
gcd := (
    P (nat → nat → nat)                    input and output types
      (λg, x, y. g y x)                    symmetric binary function
) (
    λx, y.                                 inputs a couple numbers
    is_zero x 0.
    is_zero y 0.
    ???                                    then what?
).
!assert gcd 3 6 = 3.                       an example input
!assert x:nat ⊢ gcd 0 x = 0.               zero is a special case
!assert x:nat ⊢ gcd 1 x = x.               one is a special case
!assert x:nat ⊢ gcd x x = x.               idempotent
!assert gcd x(gcd y z) = gcd(gcd x y)z.    associative
```

But there's still a hole (???) in your definition. How could you fill the hole? Could a computer create a short list of completions? Or break the hole into smaller parts? Or fill in some of the hole and ask for help?

Johann seems to have just the right information to solve such problems: a large database of terms to search through, basic equational theorem proving to cut down the search space, and very accurate complexity estimates to rank solution candidates.

If combined with a reasonable operational semantics as above, such a tool for programming-by-menu-selection would be very powerful.

Practical Solomonoff inference

Historically the first application of Kolmogorov-Solomonoff-Chaitin complexity (even preceding Kolmogorov's publication [Kol65]) was Ray Solomonoff's pseudo-algorithm for inferring the program generating a partially-observed infinite binary stream ([Sol64], [Sol78]). The pseudo-algorithm basically runs all possible programs at once, assigning to each a probability exponentially decreasing in its size. However, Solomonoff's work was posed in terms of Turing machines, which, while easy to describe, are theoretically cumbersome to work with.

An original motivation of the Johann system was to determine how many of the programs in Solomonoff's ensemble could be enumerated. Naturally, programs with similar behavior may be lumped into a single term in the probability distribution; thus the more equations we can enumerate, the smaller the space of programs becomes, and conversely the more *program behaviors* we can enumerate within given memory limitations. In fact this was our original motivation to study the theory $\mathcal{H}^*$, as it is a maximally-coarse semantics of λ-calculus.

As future work, we would like to implement Solomonoff inference (namely the prediction/transduction problem) using Johann. The first research to be done is determine how to parametrize probability distributions over the entire space by the values they take on the finite database support, apply existing theory of extrapolation or kriging.

1.4 Format

Many of the theorems in this document are verified by the automated reasoning system Johann. Displayed formulas for casual readers are printed plainly like

$\mathsf{nat} := (\mathsf{Simple}\lambda a, a'.\ (a' \to a) \to a \to a').$ *a definition*

whereas formulas for Johann are displayed with a vertical bar on the left

$\mathsf{zero} := (\lambda f, x.\ x).$ *more definitions*
$\mathsf{succ} := (\lambda n, f, x.\ n\ f(f\ x)).$
$!\mathsf{check}\ \mathsf{succ}\ \mathsf{zero} = (\lambda x.\ x).$ *a Johann command, preceeded by !*

Comments are italicized and, in color versions, displayed in blue. Some exercises for Johann are not printed in the latex'd version of this thesis; interested readers may consult the `.jtext` source or the html documentation at `http://www.askjohann.org`.

1.5 Notation

This section is a brief notational reference. Syntactic algorithms are detailed in Appendix A, e.g., compilation from λ-calculi to combinator algebras, and interpretation of statements as sets of equations and order relations.

Constants

All structures will have least and greatest elements

$!\mathsf{using}\ \bot\ \top.$
$\mathsf{Top} := \top.$
$\mathsf{Bot} := \bot.$

We use an abundance of proper combinators: to keep proof length short.

$!\mathsf{using}\ \mathbf{I}\ \mathbf{K}\ \mathbf{B}\ \mathbf{C}\ \mathbf{W}\ \mathbf{S}.$

To define a term and add it to the basis, we write

$!\mathsf{define}\ \mathbf{F} := \mathbf{K}\ \mathbf{I}.$ *adds* $\mathbf{F}$ *to basis*

which abbreviates

$!\mathsf{using}\ \mathbf{F}.$
$!\mathsf{assume}\ \mathbf{F} = \mathbf{K}\ \mathbf{I}.$

To simply define syntax we use :=, e.g. in naming the standard booleans

$\mathsf{true} := \mathbf{K}.$ *define* true *as* $\mathbf{K}$
$\mathsf{false} := \mathbf{F}.$ *define* false *as* $\mathbf{F}$

Fixedpoints and other combinators are quite useful ($\mathbf{B}'$, Φ, Ψ, Jay are defined by Curry in CurryFeys58).

$\mathbf{B}' := (\lambda x, y, z.\ y(x\ z)).$
$!\mathsf{define}\ \mathbf{Y} := \mathbf{S}\ \mathbf{B}\ \mathbf{B}'(\mathbf{W}\ \mathbf{I}).$
$\Theta := (\lambda x, y.\ y(x\ x\ y))(\lambda x, y.\ y(x\ x\ y)).$
$\Phi := (\lambda a, f, g, x.\ a(f\ x)(g\ x)).$
$\Psi := (\lambda a, f, x, y.\ a(f\ x)(f\ y)).$
$\mathsf{Jay} := (\lambda a, x, y, z.\ a\ x(a\ z\ y)).$
$\mathbf{S}' := (\lambda x, y, z.\ x\ y(x\ z)).$
$\mathsf{exp} := (\lambda a, b, f.\ b \circ f \circ a).$
$\omega := \mathbf{B}\ \mathbf{Y}\ \mathbf{B}.$

We also use optional values and sum types with values

$$\begin{aligned} &\mathsf{none} := (\lambda \mathsf{f}, -.\ \mathsf{f}).\\ &\mathsf{some} := (\lambda \mathsf{x}, -, \mathsf{g}.\ \mathsf{g}\ \mathsf{x}).\\ &\mathsf{!define\ inl} := (\lambda \mathsf{x}, \mathsf{f}, -.\ \mathsf{f}\ \mathsf{x}).\\ &\mathsf{!define\ inr} := (\lambda \mathsf{x}, -, \mathsf{g}.\ \mathsf{g}\ \mathsf{x}).\end{aligned}$$

Convergence testing has three values

$$\begin{aligned} &\mathsf{fail} := \bot.\\ &\mathsf{success} := \mathbf{I}.\\ &\mathsf{error} := \top.\end{aligned}$$

Other combinators common in literature are

$\Delta := \mathbf{W}\ \mathbf{I}.$	*the doubling combinator*
$\Omega := \Delta\ \Delta.$	*the cannonical unsolvable*

We list equational definitions for reference

$$\begin{aligned}
\mathbf{I}\ \mathsf{x} &= \mathsf{x}\\
\mathbf{K}\ \mathsf{x}\ \mathsf{y} &= \mathsf{x}\\
\mathbf{F}\ \mathsf{x}\ \mathsf{y} &= \mathsf{y}\\
\mathbf{W}\ \mathsf{x}\ \mathsf{y} &= \mathsf{x}\ \mathsf{y}\ \mathsf{y}\\
\mathbf{B}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{x}(\mathsf{y}\ \mathsf{z})\\
\mathbf{B}'\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{y}(\mathsf{x}\ \mathsf{z})\\
\mathbf{C}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{x}\ \mathsf{z}\ \mathsf{y}\\
\mathbf{S}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{x}\ \mathsf{z}(\mathsf{y}\ \mathsf{z})\\
\mathbf{S}'\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{x}\ \mathsf{y}(\mathsf{x}\ \mathsf{z})\\
\Phi\ \mathsf{f}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{f}(\mathsf{x}\ \mathsf{z})(\mathsf{y}\ \mathsf{z})\\
\Psi\ \mathsf{f}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{f}(\mathsf{x}\ \mathsf{y})(\mathsf{x}\ \mathsf{z})\\
\mathbf{Y}\ \mathsf{f} &= \mathsf{f}(\mathsf{f}(\mathsf{f}(\ldots)))\\
\mathbf{U}\ \mathsf{f} &= \mathsf{f} \mid \mathsf{f}\circ\mathsf{f} \mid \mathsf{f}\circ\mathsf{f}\circ\mathsf{f} \mid \ldots\\
\mathbf{V}\ \mathsf{f} &= \mathbf{I} \mid \mathsf{f} \mid \mathsf{f}\circ\mathsf{f} \mid \mathsf{f}\circ\mathsf{f}\circ\mathsf{f} \mid \ldots\\
\mathbf{P}\ \mathsf{x}\ \mathsf{y} &= \mathbf{V}(\mathsf{x} \mid \mathsf{y})
\end{aligned}$$

Remark. β- and η- reduction rules must be defined for all =:-defined atoms.

Operators

We use the following infix operators, ordered by precedence.

name	associativity	syntax		semantics
composition	left (=right)	$\mathsf{M}\circ\mathsf{M}'$	$:=$	$\lambda \mathsf{x}.\mathsf{M}(\mathsf{M}'\ \mathsf{x}) = \mathbf{B}\ \mathsf{M}\ \mathsf{M}'$
type exponential	right	$\mathsf{M}\to\mathsf{M}'$	$:=$	$\lambda \mathsf{f}.\mathsf{M}'\circ\mathsf{f}\circ\mathsf{M}$
application	left	$\mathsf{M}\ \mathsf{M}'$	$:=$	$\mathsf{M}(\mathsf{M}')$
randomness	none	$\mathsf{M}+\mathsf{M}'$	$:=$	$\mathbf{R}\ \mathsf{M}\ \mathsf{M}'$
join/ambiguity	right (=left)	$\mathsf{M} \mid \mathsf{M}'$	$:=$	$\mathbf{J}\ \mathsf{M}\ \mathsf{M}'$
precomposition	right (=left)	$\mathsf{M}; \mathsf{M}'$	$:=$	$\lambda \mathsf{x}.\mathsf{M}'(\mathsf{M}\ \mathsf{x}) = \mathbf{B}\ \mathsf{M}'\ \mathsf{M}$
Church's dot	right	$(\mathsf{M}.\mathsf{M}')$	$:=$	$\mathsf{M}\ \mathsf{M}'$

Note that where operators are semantically associative, their syntactic associativity is set to a default, e.g. left for composition and right for join and precomposition.. Note that the comma has lowest precedence wherever it appears.

Tuples

We use the standard tuple notation

$$\begin{aligned} \langle \mathsf{M}_1, \ldots, \mathsf{M}_n \rangle &:= \lambda \mathsf{f.f}\ \ \mathsf{M}_1\ \ \ldots\ \ \mathsf{M}_n \\ \langle\rangle &:= \lambda \mathsf{f.f}\ = \mathbf{I} \end{aligned}$$

As observed by Corrado Böhm, in this notation, church numerals multiply tuples, e.g.,

$$\begin{aligned} 3\langle \mathsf{M}\rangle = \langle \mathsf{M}\rangle; \langle \mathsf{M}\rangle; \langle \mathsf{M}\rangle & \\ &= (\lambda \mathsf{f.f}\ \ \mathsf{M}); (\lambda \mathsf{f.f}\ \ \mathsf{M}); (\lambda \mathsf{f.f}\ \ \mathsf{M}) \\ &= (\lambda \mathsf{f.f}\ \ \mathsf{M}); (\lambda \mathsf{f.f}\ \ \mathsf{M}\ \ \mathsf{M}) \\ &= (\lambda \mathsf{f.f}\ \ \mathsf{M}\ \ \mathsf{M}\ \ \mathsf{M}) \\ &= \langle \mathsf{M}, \mathsf{M}, \mathsf{M}\rangle \end{aligned}$$

and the precomposition operator concatenates tuples

$$\begin{aligned} \langle \mathsf{M}_1, \ldots, \mathsf{M}_m\rangle; \langle \mathsf{N}_1, \ldots, \mathsf{N}_n\rangle = (\lambda \mathsf{f.f}\ \ \mathsf{M}_1 \ldots \mathsf{M}_m); (\lambda \mathsf{f.f}\ \ \mathsf{N}_1 \ldots \mathsf{N}_n) & \\ &= (\lambda \mathsf{f.f}\ \ \mathsf{N}_1 \ldots \mathsf{N}_n) \circ (\lambda \mathsf{f.f}\ \ \mathsf{M}_1 \ldots \mathsf{M}_m) \\ &= (\lambda \mathsf{f.f}\ \ \mathsf{M}_1 \ldots \mathsf{M}_m\ \ \mathsf{N1} \ldots \mathsf{N}_n) \\ &= \langle \mathsf{M}_1, \ldots, \mathsf{M}_m, \mathsf{N}_1, \ldots, \mathsf{N}_n\rangle \end{aligned}$$

We often use this to denote long argument lists as in

$$\mathsf{f\ \ a\ \ b\ \ c\ \ M^{\sim m}\ \ u\ \ v\ \ N^{\sim n}\ \ x\ \ y\ \ z} = \langle \mathsf{f}\rangle.\ \ \langle \mathsf{a,b,c}\rangle;\ \ \mathsf{n}\langle \mathsf{M}\rangle;\ \ \langle \mathsf{u,v}\rangle;\ \ \mathsf{n}\langle \mathsf{N}\rangle;\ \ \langle \mathsf{x,y,z}\rangle$$

Binders

We interpret typed abstraction (in 3.7), convergence-tested abstraction (in 3.10), and patterned abstraction into pure untyped λ calculus.

Definition 1.5.1. Typed, tested, and patterned abstraction.

$$\begin{aligned} \lambda \mathsf{x}{:}\mathsf{a.M} &:= (\lambda \mathsf{x.M}) \circ (\mathbf{V}\ \ \mathsf{a}) && \textit{pre-compose with the closure } \mathsf{a} \\ \lambda \mathsf{x}{::}\mathsf{t.M} &:= \lambda \mathsf{x.}\ \ \mathsf{semi(t\ \ x)}\ \ \mathsf{M} && \textit{test with } \mathsf{t} \\ \lambda\langle \mathsf{x1}, \ldots, \mathsf{xn}\rangle.\mathsf{M} &:= \langle \lambda \mathsf{x1}, \ldots, \mathsf{xn.M}\rangle && \textit{sugar for currying} \end{aligned}$$

This allows convenient repartitioning of vector functions as in

$$\mathsf{split}_{\mathsf{mn}} := \lambda \mathsf{f}, \langle \mathsf{x1}, \ldots, \mathsf{xm}\rangle, \langle \mathsf{y1}, \ldots, \mathsf{yn}\rangle.\ \ \mathsf{f}\ \ \langle \mathsf{x1}, \ldots, \mathsf{xm},\ \ \mathsf{y1}, \ldots, \mathsf{yn}\rangle$$

Remark. An alternative definiton of patterned abstraction is as

$$\lambda\langle \mathsf{x1}, \ldots, \mathsf{xn}\rangle.\mathsf{M} := \lambda \mathsf{X}.(\lambda \mathsf{x1}, \ldots, \mathsf{xn.M})\ \ (\mathsf{X.sel\ \ 1\ \ n})\ \ \ldots\ \ (\mathsf{X.sel\ \ n\ \ n})$$

This definiton is "safer" in the sense that a poorly-typed M can't "crash" a program not depending on the tuple's components, but translates to more complicated combinators.

Definition 1.5.2. Quantification.

$$\begin{aligned} \forall \mathsf{a.M} &:= \lambda \mathsf{a,x.M(x\ \ a)} && \textit{where } \mathsf{x} \textit{ is not free in } \mathsf{M} \\ \exists \mathsf{a.M} &:= \lambda\langle \mathsf{a,x}{:}\mathsf{M}\rangle.\langle \mathsf{a,x}\rangle && \textit{where } \mathsf{x} \textit{ is not free in } \mathsf{M} \end{aligned}$$

or equivalently, letting f be a free variable,

$$\begin{aligned} \forall \mathsf{a.f\ \ a} &:= \lambda \mathsf{a,x.f\ \ a(x\ \ a)} \\ \exists \mathsf{a.f\ \ a} &:= \lambda\langle \mathsf{a,x}{:}\mathsf{f\ \ a}\rangle.\langle \mathsf{a,x}\rangle && = \lambda\langle \mathsf{a,x}\rangle.\langle \mathsf{a}, \mathbf{V}(\mathsf{f\ \ a})\mathsf{x}\rangle \end{aligned}$$

Both definitions are typically closed with $\mathbf{V}$, as described in 3.3 and employed in 3.7.

Relations and Statements

We use the following semantics for atomic relations:

$\mathsf{x}:\mathsf{y}$	$\iff$	$\mathsf{x} = \mathbf{V}\ \mathsf{y}\ \mathsf{x}$	x is a fixed-point of $\mathbf{V}\ \mathsf{y}$
$\mathsf{x}!:\mathsf{y}$	$\iff$	$\mathsf{x} \neq \mathbf{V}\ \mathsf{y}\ \mathsf{x}$	x is not a fixed-point of $\mathbf{V}\ \mathsf{y}$
$\mathsf{x} <: \mathsf{y}$	$\iff$	$\mathbf{V}\ \mathsf{x}:\mathbf{P}\ \mathsf{y}$	x is a subtype of y
$\mathsf{x}!<: \mathsf{y}$	$\iff$	$\mathbf{V}\ \mathsf{x}!:\mathbf{P}\ \mathsf{y}$	x is not a subtype of y
$\mathsf{x}::\mathsf{y}$	$\iff$	$\mathsf{semi}(\mathsf{y}\ \mathsf{x}) = \mathbf{I}$	x passes test y
$\mathsf{x}!::\mathsf{y}$	$\iff$	$\mathsf{semi}(\mathsf{y}\ \mathsf{x}) \neq \mathbf{I}$	x fails test y

In proofs of typing we will often use the (meta-level) set of inhabitants of a type $\mathrm{inhab}(\mathsf{a}) := \{\mathsf{x} \mid \mathsf{x}:\mathsf{a}\}$, as a subset of whatever finitely generated algebra we are working in.

Universally quantifed statements are interpreted by λ abstraction (where $\sim$ is one of the basic relatoins $=$, $\sqsubseteq$, or $\sqsupseteq$)

$$\begin{aligned} \forall \mathsf{x}.\ \mathsf{M} \sim \mathsf{N} &\iff (\lambda \mathsf{x}.\mathsf{M}) \sim (\lambda \mathsf{x}.\mathsf{N}) \\ \forall \mathsf{x}:\mathsf{a}.\ \mathsf{M} \sim \mathsf{N} &\iff (\lambda \mathsf{x}:\mathsf{a}.\mathsf{M}) \sim (\lambda \mathsf{x}:\mathsf{a}.\mathsf{N}) \\ \forall \mathsf{x}::\mathsf{t}.\ \mathsf{M} \sim \mathsf{N} &\iff (\lambda \mathsf{x}::\mathsf{t}.\mathsf{M}) \sim (\lambda \mathsf{x}::\mathsf{t}.\mathsf{N}) \end{aligned}$$

See A for details of statements.

Naming Conventions

We will use the following naming conventions for common idioms, e.g. types as closures in 3.7, convergence tests and correctness checks in 3.10.

$\mathsf{xxx}\ :\ \mathbf{V}$	*a type of, say, xxx's*
$\mathsf{Xxx}\ :\ \mathsf{any} \to \mathbf{V}$	*a polymorphic version of* xxx
$\mathsf{test\_xxx}\ :\ \mathsf{xxx} \to \mathsf{semi}$	*a convergence test for* xxx*'s*
$\mathsf{check\_xxx}\ :\ \mathsf{xxx} \to \mathsf{unit}$	*a correctness check for* xxx*'s*
$\mathsf{Join\_xxx}\ :\ \mathsf{Sset}\ \mathsf{xxx}$	*joins over all total* xxx*'s*
$\mathsf{eq\_xxx}\ :\ \mathsf{xxx} \to \mathsf{xxx} \to \mathsf{bool}$	*equality predicate for* xxx*'s*
$\mathsf{ppp}\ :\ \mathsf{xxx} \to \mathsf{bool}$	*a predicate of* xxx*'s*
$\mathsf{if\_ppp}\ :\ \mathsf{xxx} \to \mathsf{semi}$	*a test for* ppp
$\mathsf{assert\_ppp}\ :\ \mathsf{xxx} \to \mathsf{unit}$	*a check for* ppp

Chapter 2

Equational deduction in untyped λ-calculus (SK)

This chapter introduces untyped λ-calculus and combinatory algebra (SK) and basic inference rules for reasoning about equality and Scott's information ordering relation ($\sqsubseteq$).

A major component of this thesis is the Johann system, a piece of software to prove and verify statements about various untyped λ-calculi. We begin in this chapter by proving very simple statements, equations and order relations between untyped λ-terms. In later chapters 3, 4, 5, and 6, we progressively extend to more expressive extended untyped λ-calculi. The extensions allow us to interpret progressively stronger logics into our present equational logic. For example in 3.3 and 3.7 we show how to interpret typed equations or equations with typing contexts into a λ calculus with join (and types-as-closures). Then in 3.10 do the same with a different notion of types (types-as-convergence-tests). Finally in 5.6 we show how to interpret first-order logic and even $L_{\omega_1^{CK},\omega}$ into the equational logic of a λ-calculus extended with an oracle.

One of the advances of this thesis is the demonstration that combinatory algebra / combinatory logic is not as prohibitively unreadable and combinatorially explosive as is commonly thought. Our demonstration consists of two components:

- a "decompilation" algorithm to translate combinators back to λ-let-terms, modulo an extensional theory; and
- an generalization of the Todd-Coxeter algorithm from finitely presented groups to finitely generated combinatory algebras.

The decompilation algorithm, detailed in A.3, is not used in this printed thesis, but is a main tool of the Johann system, translating the system's internal representation to a human-readable form.

The generalized Todd-Coxeter algorithm (discussed in 7.2) is the main component of the Johann system (the remainder being syntactic algorithms, in A, and probability and statistics calculations in 7). The T-C algorithm is essentially a forward-chaining algorithm for proving equations in a finitely generated algebra, and is very similar to Robinson's unification algorithm ([CDJK99], [BS01]).

Readers interested in verification and implementation aspects of this thesis should read 7.1 and 7.2 before continuing. Readers interested only in λ-calculus and denotational semantics may safely skip the verification related material in this chapter; chapters 3, 4, 5, and 6 make sense even without an understanding of the Johann system.

2.1 Axioms for Scott's information order relation ($\sqsubseteq$ and $\not\sqsubseteq$)

```
!using ⊥ ⊤.
```

We enforce the following partial order and monotonicity axiom schemata for the ordering relation.

$\boxed{\text{partial order}}$

$$\frac{}{x \sqsubseteq x}\,(\text{refl}) \qquad \frac{x \sqsubseteq y \quad y \sqsubseteq x}{x = y}\,(\text{antisym}) \qquad \frac{x \sqsubseteq y \quad y \sqsubseteq z}{x \sqsubseteq z}\,(\text{trans})$$

$$\frac{x \sqsubseteq y \quad x \not\sqsubseteq z}{y \not\sqsubseteq z}\,(\text{trans}-\mathbf{R}) \qquad \frac{x \not\sqsubseteq y \quad y \sqsubseteq z}{x \not\sqsubseteq z}\,(\text{trans}-\text{L})$$

$\boxed{\text{monotonicity}}$

$$\frac{f \sqsubseteq g}{f\ x \sqsubseteq g\ x}\,(\mu) \qquad \frac{f\ x \not\sqsubseteq g\ x}{f \not\sqsubseteq g}\,(\mu') \qquad \frac{x \sqsubseteq y}{f\ x \sqsubseteq f\ y}\,(\nu) \qquad \frac{f\ x \not\sqsubseteq f\ y}{x \not\sqsubseteq y}\,(\nu')$$

We had originally implemented the stronger monotonicity schema

$$\frac{f \sqsubseteq g \quad x \sqsubseteq y}{f\ x \sqsubseteq g\ y}\,(\mu+\nu)$$

whose enforcement takes quartic time (searching through four variables f, g, x, y; see [McA99] for complexity analysis techniques). However this was very slow, taking far more time than all other schemata combined. Thus we split the rule $(\mu+\nu)$ into two cubic-time schemata, (μ) and (ν) above.

We also assume that the Scott ordering has least- and greatest elements

```
!assume ⊥ ⊑ x ⊑ ⊤.
!assume ⊥ ⋣ ⊤.          for absolute consistency
```

This latter is the only negative axiom we assume; two terms M, N are distinct modulo $\mathcal{H}^*$ iff $M = N \vdash \bot \sqsupseteq \top$.

Both ends are constant

```
!assume ⊥ x = ⊥.
!assume ⊤ x = ⊤.
```

2.2 Axioms for basic combinators

Axiom Schemata

A few axiom schemata are hard-coded in Johann: β-reduction

$$\frac{}{\mathbf{I}\ x = x} \qquad \frac{}{\mathbf{Y}\ g = f(\mathbf{Y}\ f)} \qquad \frac{}{\mathbf{K}\ x\ y = x} \qquad \frac{}{\mathbf{W}\ x\ y = x\ y\ y} \qquad \frac{}{\mathbf{B}\ x\ y\ z = x(y\ z)} \qquad \frac{}{\mathbf{C}\ x\ y\ z = x\ z\ y}$$

$$\frac{}{\mathbf{S}\ x\ y\ z = x\ z(y\ z)}$$

and some η schemata, including two fixed-point schemata for $\mathbf{Y}$

$$\frac{}{x\circ(y\circ z) = (x\circ y)\circ z}\,(\mathbf{B}-\text{assoc}) \qquad \frac{y = \mathbf{S}\ \mathbf{I}\ y}{y = \mathbf{Y}}\,(\text{Y1}) \qquad \frac{f\ x \sqsubseteq x}{\mathbf{Y}\ f \sqsubseteq x}\,(\text{Y2})$$

where (Y2) is just a pointwise version of (Y1). (The (Y1) rule was proved independently by Böhm and van der May; see [Bar84] Lemma 6.5.3)

Axioms for extensionality

First let us define all other atoms in terms of $\mathbf{S}$ and $\mathbf{K}$.

```
!assume  $\mathbf{I} = \mathbf{S}\ \mathbf{K}\ \mathbf{K}$.
!assume  $\mathbf{F} = \mathbf{K}\ \mathbf{I}$.
!assume  $\mathbf{B} = \mathbf{S}(\mathbf{K}\ \mathbf{S})\mathbf{K}$.
!assume  $\mathbf{C} = \mathbf{S}(\mathbf{S}(\mathbf{K}\ \mathbf{B})\mathbf{S})(\mathbf{K}\ \mathbf{K}) = \mathbf{S}(\mathbf{B}\ \mathbf{B}\ \mathbf{S})(\mathbf{K}\ \mathbf{K})$.
!assume  $\mathbf{W} = \mathbf{C}\ \mathbf{S}\ \mathbf{I}$.
!assume  $\mathbf{Y} = \mathbf{S}\ \mathbf{B}\ \mathbf{B}'(\mathbf{W}\ \mathbf{I}) = \mathbf{B}(\mathbf{W}\ \mathbf{I})(\mathbf{B}'.\mathbf{W}\ \mathbf{I})$.
!assume  $\mathbf{Y} = \mathbf{S}(\mathbf{B}'.\mathbf{W}\ \mathbf{I})(\mathbf{B}'.\mathbf{W}\ \mathbf{I})$.               a la Rosenbloom
!assume  $\mathbf{Y} = \mathbf{B}(\mathbf{S}\ \mathbf{I})(\mathbf{W}\ \mathbf{I})(\mathbf{B}(\mathbf{S}\ \mathbf{I}).\mathbf{W}\ \mathbf{I})$.          a la Turing
!assume  $\mathbf{Y} = \mathbf{S}\ \mathbf{S}\ \mathbf{K}(\mathbf{S}(\mathbf{K}.\mathbf{S}\ \mathbf{S}.\mathbf{S}.\mathbf{S}\ \mathbf{S}\ \mathbf{K})\mathbf{K})$.        a la Tromp (see [Tro02])
```

For extensionality we need β-reduction to commute with Curry's bracket abstraction algorithm, i.e., $\mathsf{M} \twoheadrightarrow \mathsf{N} \implies \lambda x.\mathsf{M} = \lambda x.\mathsf{N}$. This entails

- β-reducuction axioms, where each reduction leads to an η axiom, and
- (bracket) abstraction axioms, so that the abstraction algorithm commutes with β-reduction.

First, the β-reduction axioms ensure that $\mathsf{M} \twoheadrightarrow \mathsf{N} \implies \mathsf{M} = \mathsf{N}$.

```
!assume  $0\ x\ y = y$.                  i.e., $\mathbf{F}\ x\ y = y$.
!assume  $1\ x\ y = x\ y$.               i.e., $\mathbf{I}\ x = x$.
!assume  $2\ x\ y = x(x\ y)$.            i.e., $\mathbf{W}\ \mathbf{B}\ x = x \circ x$.
!assume  $\mathbf{K}\ x\ y = x$.
!assume  $\mathbf{W}\ \mathbf{I}\ x = x\ x$.
!assume  $\mathbf{W}\ x\ y = x\ y\ y$.
!assume  $\mathbf{B}\ x\ y\ z = x(y\ z)$.
!assume  $\mathbf{C}\ \mathbf{I}\ x\ y = y\ x$.
!assume  $\mathbf{C}\ x\ y\ z = x\ z\ y$.
!assume  $\mathbf{S}\ x\ y\ z = x\ z(y\ z)$.        very expensive!
!assume  $\mathbf{Y}\ x = x(\mathbf{Y}\ x)$.
!assume  $\mathbf{B}'\ x\ y\ z = y(x\ z)$.
!assume  $\mathbf{S}'\ x\ y = (x\ y) \circ x$.
```

Second, the abstraction axioms ensure that $\mathsf{M} \twoheadrightarrow \mathsf{N} \implies \lambda x.\mathsf{M} = \lambda x.\mathsf{N}$. We extend Hindley's extensionality proof ([Hin67]) to the (bracket) abstraction algorithm $\mathbf{S}, \mathbf{K}, \mathbf{I}, \mathbf{B}, \mathbf{C}, \mathbf{W}, \eta$ compiling $\lambda x.\mathsf{M}$ by the following rules

rule	$\lambda x.\mathsf{M}$	condition
K.	$\mathbf{K}\ \mathsf{M}$	if x not free in M
I.	$\mathbf{I}$	if $\mathsf{M} \equiv x$
		otherwise $\mathsf{M} = \mathsf{M}'\ \mathsf{M}''$ is an application:
η.	M'	if $\mathsf{M}'' \equiv x$
B.	$\mathbf{B}\ \mathsf{M}'(\lambda x.\mathsf{M}'')$	if x free in M'' but not M'
C.	$\mathbf{C}(\lambda x.\mathsf{M}')\mathsf{M}''$	if x free in M' but not M''
W.	$\mathbf{W}(\lambda x.\mathsf{M}')$	if x free in M' and $\mathsf{M}'' = x$
S.	$\mathbf{S}(\lambda x.\mathsf{M}')(\lambda x.\mathsf{M}'')$	if x free in M' and M'' and $\mathsf{M}'' \not\equiv x$

each abstraction axiom identifies a pair of paths through this algorithm, which are not otherwise independent of β reduction. E.g., for $\mathsf{M} \twoheadrightarrow \mathsf{N}$, certainly $\mathsf{FV}(\mathsf{N}) \subseteq \mathsf{FV}(\mathsf{M})$, but x may appear free in M but not N.

!assume $\mathbf{K}$(a b)	λx.a b
$=$ $\mathbf{C}$($\mathbf{K}$ a)b	$\mathbf{K} \twoheadleftarrow \mathbf{B}$
$=$ $\mathbf{B}$ a($\mathbf{K}$ b)	$\mathbf{K} \twoheadleftarrow \mathbf{C}$
$=$ $\mathbf{S}$($\mathbf{K}$ a)($\mathbf{K}$ b).	$\mathbf{K} \twoheadleftarrow \mathbf{S}$
!assume $\mathbf{I}$	λx.x
$=$ $\mathbf{C}$ $\mathbf{K}$ a	$\mathbf{I} \twoheadleftarrow \mathbf{C}$
$=$ $\mathbf{W}$ $\mathbf{K}$	$\mathbf{I} \twoheadleftarrow \mathbf{W}$
$=$ $\mathbf{S}$ $\mathbf{K}$ a.	$\mathbf{I} \twoheadleftarrow \mathbf{S}$
!assume a	λx.a x
$=$ $\mathbf{B}$ a $\mathbf{I}$	$\eta \twoheadleftarrow \mathbf{B}$
$=$ $\mathbf{B}$ $\mathbf{I}$ a	$\eta \twoheadleftarrow \mathbf{B}$
$=$ $\mathbf{W}$($\mathbf{K}$ a)	$\eta \twoheadleftarrow \mathbf{W}$
$=$ $\mathbf{S}$($\mathbf{K}$ a)$\mathbf{I}$	$\eta \twoheadleftarrow \mathbf{S}$
$=$ $\mathbf{S}$($\mathbf{K}$ $\mathbf{I}$)a.	$\eta \twoheadleftarrow \mathbf{S}$

!assume $\mathbf{B}$ a b	λx.a(b x)
$=$ $\mathbf{S}$($\mathbf{K}$ a)b.	$\mathbf{B} \twoheadleftarrow \mathbf{S}$
!assume $\mathbf{C}$ a b	λx.a x b
$=$ $\mathbf{S}$ a($\mathbf{K}$ b).	$\mathbf{C} \twoheadleftarrow \mathbf{S}$
!assume $\mathbf{W}$ a	λx.a x x
$=$ $\mathbf{S}$ a $\mathbf{I}$.	$\mathbf{W} \twoheadleftarrow \mathbf{S}$

In addition to the "syntactic schemata" above, Hindley's finite axiomatization of strong reduction requires a semantics of the closure operation on equations. The simplest such closure would be

$$\mathsf{closure}(\mathsf{M} = \mathsf{N}) = \{\lambda \mathsf{x1}, \ldots, \mathsf{xn}.\mathsf{M} = \lambda \mathsf{x1}, \ldots, \mathsf{xn}.\mathsf{N}\}$$

where $\{\mathsf{x1}, \ldots, \mathsf{xn}\} = \mathsf{FV}(\mathsf{M}) + \mathsf{FV}(\mathsf{N})$. For example from above,

$$\mathsf{closure}(\mathbf{K}(\mathsf{a}\ \mathsf{b}) = \mathbf{B}\ \mathsf{a}(\mathbf{K}\ \mathsf{b})) = \{\lambda \mathsf{a}, \mathsf{b}.\mathbf{K}(\mathsf{a}\ \mathsf{b}) = \lambda \mathsf{a}, \mathsf{b}.\mathbf{B}\ \mathsf{a}(\mathbf{K}\ \mathsf{b})\}$$

where as always λx.M is the bracket abstraction of x out of M.

Hindley's closure operation produces a set of equations, each of which must be "assumed". The set is constructed by allowing each free variable to also be "wrapped" in a fresh free variable, so, e.g.,

$$\begin{array}{ll}
\mathsf{closure}(\mathbf{K}(\mathsf{a}\ \mathsf{b}) = \mathbf{B}\ \mathsf{a}(\mathbf{K}\ \mathsf{b})) = \{ & \\
\quad \lambda \mathsf{a}, \mathsf{b}.\mathbf{K}(\mathsf{a}\ \mathsf{b}) = \lambda \mathsf{a}, \mathsf{b}.\mathbf{B}\ \mathsf{a}(\mathbf{K}\ \mathsf{b}), & \textit{neither are wrapped} \\
\quad \lambda \mathsf{f}, \mathsf{a}, \mathsf{b}.\mathbf{K}(\mathsf{f}\ \mathsf{a}\ \mathsf{b}) = \lambda \mathsf{f}, \mathsf{a}, \mathsf{b}.\mathbf{B}(\mathsf{f}\ \mathsf{a})(\mathbf{K}\ \mathsf{b}), & \mathsf{a}\ \textit{is wrapped} \\
\quad \lambda \mathsf{f}, \mathsf{a}, \mathsf{b}.\mathbf{K}(\mathsf{a}(\mathsf{f}\ \mathsf{b})) = \lambda \mathsf{f}, \mathsf{a}, \mathsf{b}.\mathbf{B}\ \mathsf{a}(\mathbf{K}(\mathsf{f}\ \mathsf{b})), & \mathsf{b}\ \textit{is wrapped} \\
\quad \lambda \mathsf{f}, \mathsf{a}, \mathsf{b}.\mathbf{K}(\mathsf{f}\ \mathsf{a}(\mathsf{f}\ \mathsf{b})) = \lambda \mathsf{f}, \mathsf{a}, \mathsf{b}.\mathbf{B}(\mathsf{f}\ \mathsf{a})(\mathbf{K}(\mathsf{f}\ \mathsf{b})) & \textit{Sub are wrapped} \\
\} &
\end{array}$$

As the combinatory axioms form the core of Johann, and proof search capabilities are limited, we use a much stronger closure semantics. Our closure is in two parts: first a wrapping operation of order $m = 2^{|FV|}$ and then a mapping operation of factorial order, where every permutation and combination of variables is considered. Extending the example above, the first operation yields

$$
\begin{array}{ll}
\mathsf{wrap}(\mathbf{K}(\mathsf{a}\ \mathsf{b}) = \mathbf{B}\ \mathsf{a}(\mathbf{K}\ \mathsf{b})) = \{ & \\
\quad \mathbf{K}(\mathsf{a}\ \mathsf{b}) = \mathbf{B}\ \mathsf{a}(\mathbf{K}\ \mathsf{b}), & \textit{neither are wrapped} \\
\quad \mathbf{K}(\mathsf{f}\ \mathsf{a}\ \mathsf{b}) = \mathbf{B}(\mathsf{f}\ \mathsf{a})(\mathbf{K}\ \mathsf{b}), & \mathsf{a}\ \textit{is wrapped} \\
\quad \mathbf{K}(\mathsf{a}(\mathsf{f}\ \mathsf{b})) = \mathbf{B}\ \mathsf{a}(\mathbf{K}(\mathsf{f}\ \mathsf{b})), & \mathsf{b}\ \textit{is wrapped} \\
\quad \mathbf{K}(\mathsf{f}\ \mathsf{a}(\mathsf{f}\ \mathsf{b})) = \mathbf{B}(\mathsf{f}\ \mathsf{a})(\mathbf{K}(\mathsf{f}\ \mathsf{b})) & \textit{Sub are wrapped} \\
\} &
\end{array}
$$

simply the open versions of Hindley's, and the second operation yields, say on the second in the list

$$
\begin{array}{ll}
\mathsf{closure}(\mathbf{K}(\mathsf{f}\ \mathsf{a}\ \mathsf{b}) = \mathbf{B}(\mathsf{f}\ \mathsf{a})(\mathbf{K}\ \mathsf{b})) = \{ & \\
\quad \lambda\{\mathsf{f},\mathsf{a},\mathsf{b}\}.\mathbf{K}(\mathsf{f}\ \mathsf{a}\ \mathsf{b}) = \lambda\{\mathsf{f},\mathsf{a},\mathsf{b}\}.\mathbf{B}(\mathsf{f}\ \mathsf{a})(\mathbf{K}\ \mathsf{b}), & \textit{6 permutations of } \{\mathsf{f},\mathsf{a},\mathsf{b}\} \\
\quad \lambda\{\mathsf{x},\mathsf{b}\}.\mathbf{K}(\mathsf{x}\ \mathsf{x}\ \mathsf{b}) = \lambda\{\mathsf{x},\mathsf{b}\}.\mathbf{B}(\mathsf{x}\ \mathsf{x})(\mathbf{K}\ \mathsf{b}), & \textit{2 permutations when } \mathsf{f}=\mathsf{a}=\mathsf{x} \\
\quad \lambda\{\mathsf{x},\mathsf{a}\}.\mathbf{K}(\mathsf{x}\ \mathsf{a}\ \mathsf{x}) = \lambda\{\mathsf{x},\mathsf{a}\}.\mathbf{B}(\mathsf{x}\ \mathsf{a})(\mathbf{K}\ \mathsf{x}), & \textit{2 permutations when } \mathsf{f}=\mathsf{b}=\mathsf{x} \\
\quad \lambda\{\mathsf{x},\mathsf{f}\}.\mathbf{K}(\mathsf{f}\ \mathsf{x}\ \mathsf{x}) = \lambda\{\mathsf{x},\mathsf{a}\}.\mathbf{B}(\mathsf{f}\ \mathsf{x})(\mathbf{K}\ \mathsf{x}), & \textit{2 permutations when } \mathsf{a}=\mathsf{b}=\mathsf{x} \\
\quad \lambda\mathsf{x}.\mathbf{K}(\mathsf{x}\ \mathsf{x}\ \mathsf{x}) = \lambda\mathsf{x}.\mathbf{B}(\mathsf{x}\ \mathsf{x})(\mathbf{K}\ \mathsf{x}) & \textit{the case when all coinincide} \\
\} &
\end{array}
$$

This closure semantics is prohibitively expensive for large numbers of variables, say four or more (see Sloan's A000670, [Slo])

number of free variables	0	1	2	3	4	5	6	7
number of closures	1	3	13	75	541	4683	47293	545835

so axioms involving a large number of variables should be partially abstracted by hand. This was done in the reduction axiom for $\mathbf{S}'$ above, where the variable z is abstracted from the schema $\mathbf{S}'\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} = \mathsf{x}\ \mathsf{y}(\mathsf{x}\ \mathsf{z})$ to yield the simpler $\mathbf{S}'\ \mathsf{x}\ \mathsf{y} = (\mathsf{x}\ \mathsf{y})\circ\mathsf{x}$.

In this semantics, Johann's most expensive axioms all involve three variables

- the $\mathbf{S}$ axiom $\mathbf{S}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} = \mathsf{x}\ \mathsf{z}(\mathsf{y}\ \mathsf{z})$ costing 221 obs;
- the $\mathbf{J}$ linearity axioms $\mathbf{J}(\mathsf{x}\ \mathsf{y})(\mathsf{x}\ \mathsf{z}) \sqsubseteq \mathsf{x}(\mathbf{J}\ \mathsf{y}\ \mathsf{z})$ and $\mathbf{J}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} = \mathbf{J}(\mathsf{x}\ \mathsf{z})(\mathsf{y}\ \mathsf{z})$ costing 216 and 173 obs, respectively (see 3.1);
- the associativity of composition axiom

  ```
  !assume x∘(y∘z) = (x∘y)∘z.
  ```

  costing 131 obs; and
- the $\mathbf{B}$ axiom $\mathbf{B}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} = \mathsf{x}(\mathsf{y}\ \mathsf{z})$ costing 119 obs;
- the $\mathbf{C}$ axiom $\mathbf{C}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} = \mathsf{x}\ \mathsf{z}\ \mathsf{y}$ costing 83 obs.

Relations to other axiomatizations of η

Barendregt's axioms for extensionality are immediately satisfied (see [Bar84], pp. 158-161):

!check $\mathbf{S}\ \mathbf{B}(\mathbf{K}\ \mathbf{I}) = \mathbf{I}$.	$SKIBC\eta = SKIBC$
!check $\mathbf{S}(\mathbf{K}\ \mathsf{x})(\mathbf{K}\ \mathsf{y}) = \mathbf{K}(\mathsf{x}\ \mathsf{y})$.	$SKI(=\lambda^*) = \lambda_1$
!check $\mathbf{S}(\mathbf{K}\ \mathsf{x})\mathsf{y} = \mathbf{B}\ \mathsf{x}\ \mathsf{y}$.	$SKIB = SKI$
!check $\mathbf{S}\ \mathsf{x}(\mathbf{K}\ \mathsf{y}) = \mathbf{C}\ \mathsf{x}\ \mathsf{y}$.	$SKIBC = SKIB$

Hindley's axioms for strong reduction are immediately satisfied (see [Hin67]):

```
!check S x y z = (x z)(y z).
!check K x y = x.
!check I x = x.
!check S(K x)I = x.
!check S(K x)(K y) = K(x y).
!check S(K I) = I.
!check x = x.
```

The Meyer-Scott axiom is immediately satisfied (see [Bar84], pp. 95):

$\text{!check}\ \mathbf{S}(\mathbf{K}\ \mathbf{I}) = \mathbf{I}.$

Derived properties

Definitions of common combinators

$\text{!check}\ 2 = \mathbf{W}\ \mathbf{B}.$
$\text{!check}\ \mathbf{B}' = \mathbf{C}\ \mathbf{B}.$
$\text{!check}\ \Delta = \mathbf{W}\ \mathbf{I}.$
$\text{!check}\ \Phi = (\mathbf{B}\ \mathbf{S})\circ\mathbf{B}.$
$\text{!check}\ \Psi = \mathbf{W}\circ(\mathbf{B}\ \mathbf{C})\circ(\mathbf{B}.\mathbf{B}\ \mathbf{B})\circ\mathbf{B}.$
$\text{!check}\ \mathsf{Jay} = \mathbf{S}\ \mathbf{C}\circ(\mathbf{W}\ \mathbf{B}.\mathbf{B}\ \mathbf{B})\mathbf{C}.$

Alternate definitions

$\text{!check}\ 2 = \mathbf{W}\ \mathbf{B}'.$
$\text{!check}\ \mathbf{S} = \Phi\ \mathbf{I}.$
$\text{!check}\ \mathbf{S}' = \Psi\ \mathbf{I}.$
$\text{!check}\ \mathbf{Y} = (\lambda x, f.\ f(x\ x\ f))(\lambda x, f.\ f(x\ x\ f)).$
$\text{!check}\ \mathbf{Y} = (\lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))).$ $\quad = \mathbf{S}(\mathbf{B}'.\mathbf{W}\ \mathbf{I})(\mathbf{B}'.\mathbf{W}\ \mathbf{I})$
$\text{!check}\ \mathbf{Y} = \mathbf{W}\ \mathbf{S}(\mathbf{B}'\ \Delta).$

2.3 Theorems proved by η-conversion

Finite η-expansion

$\mathbf{C}$ commutes with Curry's $\Phi = \lambda f, x, y, z.f(x\ z)(y\ z)$

$\text{!assume}\ \mathbf{C}\circ\Phi = \Phi\circ\mathbf{C}.$

Proof. (requiring four η-instances)

$$\begin{aligned} \mathbf{C}\circ\Phi\ f\ x\ y\ z &= \mathbf{C}(\Phi\ f)x\ y\ z \\ &= \Phi\ f\ y\ x\ z \\ &= f(y\ z)(x\ z) \\ &= \mathbf{C}\ f(x\ z)(y\ z) \\ &= \Phi(\mathbf{C}\ f)x\ y\ z \\ &= \Phi\circ\mathbf{C}\ f\ x\ y\ z \end{aligned}$$

Theorems conjectured by Johann and proved by finite η expansion:[1]

$\text{!assume}\ \mathbf{I} = \mathbf{C}\circ\mathbf{C} = \mathbf{B}\ \mathbf{I} = \mathbf{S}(\mathbf{C}\ \mathbf{I})\mathbf{K} = \mathbf{B}\ \mathbf{W}\ \mathbf{K} = \mathbf{B}\ \mathbf{Y}\ \mathbf{K} = \mathbf{S}(\mathbf{S}\ \mathbf{K}).$
$\text{!assume}\ \mathbf{W} = \mathbf{S}(\mathbf{C}\ \mathbf{I}).$
$\text{!assume}\ \mathbf{C}\ \mathbf{C} = \mathbf{B}\ \mathbf{B}(\mathbf{C}\ \mathbf{I}).$
$\text{!assume}\ \mathbf{W}\ \mathbf{S} = \mathbf{B}(\mathbf{W}\ \mathbf{I}).$
$\text{!assume}\ \mathbf{B}(\mathbf{C}\ \mathbf{I}) = \mathbf{C}\circ\mathbf{B}'.$
$\text{!assume}\ \mathbf{B}\ \mathbf{B}\ \mathbf{K} = \mathbf{B}\ \mathbf{K}\ \mathbf{K}.$
$\text{!assume}\ \langle\bot\rangle \sqsubseteq \mathbf{W}.$ *i.e.,* $f\ \bot\ x \sqsubseteq f\ x\ x$
$\text{!assume}\ \mathbf{C}\ \mathbf{C}\ \bot \sqsubseteq \mathbf{W}.$ *i.e.,* $f\ x\ \bot \sqsubseteq f\ x\ x$

[1]The conjecturing algorithm is discribedin 7.4.

Infinite η-expansion

Lemma 2.3.1.

> !assume $\mathbf{Y}\ \mathbf{B}' = \mathbf{I}$.

Proof. (by infinite η-expansion) This is like Wadsworth's **J** ([Bar84], remark 16.2.3, pp 420)

$$\begin{aligned}\mathbf{Y}\ \mathbf{B}'\ \mathsf{x}\ \mathsf{y} &\to \mathbf{B}'(\mathbf{Y}\ \mathbf{B}')\mathsf{x}\ \mathsf{y}\\ &\to \mathbf{B}\ \mathsf{x}(\mathbf{Y}\ \mathbf{B}')\mathsf{y}\\ &\to \mathsf{x}(\mathbf{Y}\ \mathbf{B}'\ \mathsf{y})\end{aligned}$$

□

Lemma 2.3.2.

> !assume $\mathbf{Y}\circ\mathbf{B}' = \bot$.

Proof. (by infinite precomposition)

$$\begin{aligned}\mathbf{Y}\circ\mathbf{B}'\ \mathsf{x} &= \mathbf{Y}(\mathbf{B}'\ \mathsf{x})\\ &= \mathbf{B}'\ \mathsf{x}(\mathbf{Y}.\mathbf{B}'\ \mathsf{x})\\ &= \mathbf{B}(\mathbf{Y}.\mathbf{B}'\ \mathsf{x})\mathsf{x}\\ &= (\mathbf{Y}.\mathbf{B}'\ \mathsf{x})\circ\mathsf{x}\\ &= (\mathbf{Y}.\mathbf{B}'\ \mathsf{x})\circ\mathsf{x}\circ\mathsf{x}\circ\mathsf{x}\circ\mathsf{x}\circ\mathsf{x}\circ\ldots \quad \textit{iterating the above argument}\\ &= \bot\end{aligned}$$

and hence $\mathbf{Y}\circ\mathbf{B}' = \mathbf{K}\ \bot = \bot$. □

The next theorem identifies two representations of the infinite extended church numeral ω

Lemma 2.3.3.

> !assume $\mathbf{Y}\circ\mathbf{B} = \mathbf{K}\circ\mathbf{Y}$.

Proof. we take the directed limit of a sequence of church numerals

$$\begin{aligned}\mathbf{Y}\circ\mathbf{B}\ \mathsf{f}\ \mathsf{x} &= \mathbf{Y}(\mathbf{B}\ \mathsf{f})\mathsf{x}\\ &= \mathsf{f}\circ(\mathbf{Y}(\mathbf{B}\ \mathsf{f}))\mathsf{x}\\ &= (\mathsf{f}\circ\mathsf{f}\circ\ldots)\mathsf{x}\\ &= \bigsqcup \mathsf{n}.\ \mathsf{n}\ \mathsf{f}\ \bot\\ &= \mathbf{Y}\ \mathsf{f}\end{aligned}$$

□

Theorems conjectured by Johann and proved by infinite η expansion:

> !assume $\mathbf{B}\ \mathbf{Y} = \mathbf{Y}(\mathbf{S}\ \mathbf{S})$.
> !assume $\mathbf{C}(\mathbf{K}\ \mathbf{Y}) = \mathbf{Y}\circ\mathbf{B}$.

2.4 Axioms for divergent terms (approaching $\mathcal{H}$)

The fixed-point combinator diverges on many arguments

> !assume $\bot = \mathbf{Y}\ \mathbf{K} = \mathbf{Y}\ \mathbf{S} = \mathbf{Y}\ \mathbf{W} = \mathbf{Y}\ \mathbf{B} = \mathbf{Y}\ \mathbf{C} = \mathbf{Y}\ \mathbf{I} = \mathbf{Y}\ \mathbf{Y}$.
> !assume $\bot = \mathbf{Y}(\mathbf{B}\ \mathbf{K})$. *(suggested by Johann)*

Lemma 2.4.1.

> !assume $\bot = \mathbf{W}\ \mathbf{W}\ \mathbf{W}$.

Proof. by $\bot$-reduction: $\mathbf{W}\ \mathbf{W}\ \mathbf{W}$ head-reduces to itself.

Theorems conjectured by Johann

Lemma 2.4.2.

| !assume $\bot = \mathbf{Y}\ \mathbf{K}{\circ}\mathbf{W}$.

Proof. $\mathbf{Y}\ \mathbf{K}{\circ}\mathbf{W}\ \mathsf{x}\ \mathsf{y}\ \to\ \mathbf{Y}\ \mathbf{K}{\circ}\mathbf{W}\ \mathsf{y}\ \mathsf{y}$

Lemma 2.4.3.

| !assume $\bot = \mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I})$.

Proof. by $\eta - \bot$-reduction: $\mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I})\ \mathsf{x}$ head-reduces to $\mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I})$

$$\begin{aligned}
\mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I})\ \mathsf{x} &\to \mathbf{S}\ \mathbf{C}(\mathbf{Y}(\mathbf{S}\ \mathbf{C}))(\mathbf{K}\ \mathbf{I})\mathsf{x}\\
&\to \mathbf{C}\ (\mathbf{K}\ \mathbf{I})\ (\mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I}))\mathsf{x}\\
&\to \mathbf{K}\ \mathbf{I}\ \mathsf{x}\ (\mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I}))\\
&\to \mathbf{Y}(\mathbf{S}\ \mathbf{C})(\mathbf{K}\ \mathbf{I})
\end{aligned}$$

Lemma 2.4.4.

| !assume $\bot = \mathbf{W}\ \mathbf{I}(\mathbf{W}\ \mathbf{W})$.

Proof. by reduction to super-term:

$$\begin{aligned}
\mathbf{W}\ \mathbf{I}(\mathbf{W}\ \mathbf{W}) &\to \mathbf{I}(\mathbf{W}\ \mathbf{W})(\mathbf{W}\ \mathbf{W})\\
&\to \mathbf{W}\ \mathbf{W}(\mathbf{W}\ \mathbf{W})\\
&\to \mathbf{W}(\mathbf{W}\ \mathbf{W})(\mathbf{W}\ \mathbf{W})\\
&\to \ldots
\end{aligned}$$

whose reduction sequence never terminates.

Lemma 2.4.5.

| !assume $\bot = \mathbf{Y}(\mathbf{C}\ \mathbf{I})(\mathbf{Y}(\mathbf{C}\ \mathbf{I}))$.

Proof. by reduction loop:

$$\mathbf{Y}(\mathbf{C}\ \mathbf{I})(\mathbf{Y}(\mathbf{C}\ \mathbf{I}))\quad\to\quad\langle\mathbf{Y}(\mathbf{C}\ \mathbf{I})\rangle(\mathbf{Y}(\mathbf{C}\ \mathbf{I}))\quad\to\quad\mathbf{Y}(\mathbf{C}\ \mathbf{I})(\mathbf{Y}(\mathbf{C}\ \mathbf{I}))$$

Lemma 2.4.6.

| !assume $\bot = \mathbf{W}\ \mathbf{Y}(\mathbf{C}\ \mathbf{W})$.

Proof. by reduction to super-term:

$$\begin{aligned}
\mathbf{W}\ \mathbf{Y}(\mathbf{C}\ \mathbf{W}) &\to \mathbf{Y}(\mathbf{C}\ \mathbf{W})(\mathbf{C}\ \mathbf{W}) && \to \mathbf{C}\ \mathbf{W}(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{C}\ \mathbf{W})\\
&\to \mathbf{W}(\mathbf{C}\ \mathbf{W})(\mathbf{Y}(\mathbf{C}\ \mathbf{W})) && \to \mathbf{C}\ \mathbf{W}(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))\\
&\to \mathbf{W}(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))\\
&\to \mathbf{Y}(\mathbf{C}\ \mathbf{W})(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))\\
&\to \mathbf{Y}(\mathbf{C}\ \mathbf{W})(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))\\
&\to \mathbf{Y}(\mathbf{C}\ \mathbf{W})(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))(\mathbf{Y}(\mathbf{C}\ \mathbf{W}))\\
&\to \ldots
\end{aligned}$$

Lemma 2.4.7.

| !assume $\bot = \mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S})$.

Proof. by reduction to everywhere-infinitely-deep tree

$$\begin{aligned}\mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S}) &\to \mathbf{S}\ \mathbf{S}\ \mathbf{S}(\mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S}))\\ &\to \mathbf{S}(\mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S}))(\mathbf{S}(\mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S})))\\ &\twoheadrightarrow \mathbf{S}(\mathbf{S}(\dots)(\dots))(\mathbf{S}(\dots)(\mathbf{S}(\dots))\end{aligned}$$

whence $\mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S})\mathsf{x} \to \mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S})\mathsf{x}(\mathbf{Y}(\mathbf{S}\ \mathbf{S}\ \mathbf{S})\ \mathsf{x})$

Lemma 2.4.8.

| !assume $\bot = \mathbf{Y}\circ(\mathbf{C}\ \mathbf{I})$.

Proof. by reduction to sub-term

$$\begin{aligned}\mathbf{Y}\circ(\mathbf{C}\ \mathbf{I})\ \mathsf{x} &\to \mathbf{Y}\langle\mathsf{x}\rangle &&\to \langle\mathsf{x}\rangle(\mathbf{Y}\langle\mathsf{x}\rangle)\\ &\to \mathbf{Y}\langle\mathsf{x}\rangle\mathsf{x} &&\to \langle\mathsf{x}\rangle(\mathbf{Y}\langle\mathsf{x}\rangle)\mathsf{x}\\ &\to \mathbf{Y}\langle\mathsf{x}\rangle\mathsf{x}\ \mathsf{x} &&\to \langle\mathsf{x}\rangle(\mathbf{Y}\langle\mathsf{x}\rangle)\mathsf{x}\ \mathsf{x}\\ &\to \mathbf{Y}\langle\mathsf{x}\rangle\mathsf{x}\ \mathsf{x}\ \mathsf{x} &&\to \dots\end{aligned}$$

Lemma 2.4.9.

| !assume $\bot = \mathbf{C}(\mathbf{Y}(\mathbf{C}(\mathbf{K}\ \mathbf{C})))\mathbf{B}$.

Proof. $\dots\ \mathsf{x} \to \dots$

Theorems suggested by solutions to equations

These were suggested by the fixed point definition of nil:

$$\mathsf{x} \in \{\top, \bot\} \iff \mathbf{K}\ \mathsf{x} = \mathsf{x}$$

| !assume $\bot = \mathbf{Y}(\mathbf{W}\ \mathbf{B}\ \mathbf{K}) = \mathbf{Y}\ \mathbf{K}\circ\mathbf{B}$.

Chapter 3

Untyped λ-calculus with ambiguity-as-join (SKJ)

This chapter introduces an untyped λ-calculus with a binary semilattice operation, the join with respect to Scott's information ordering. We call this system SKJ , as it is the fragment of Scott's D_∞ model generated by $\mathbf{S}$, $\mathbf{K}$, and binary join. λ-calculus with join has long been studied as a model for concurrency / parallelism and ambiguity / multiplicity; nondeterminism also satisfies the semilattice axioms, but is better modelled by meet ([DCL02]).

The main discovery of this thesis is that, under the theory $\mathcal{H}^*$, that there is an interpretation of typed λ-calculus SK into untyped λ-join-calculus SKJ , where types are interpreted as closure operations (SKJ -terms a satisfying $\mathbf{I} \sqsubseteq \mathsf{a} = \mathsf{a}\circ\mathsf{a}$). Specifically we show in 3.6 the following:

Theorem 3.0.10. *Let τ be a simple type (with any number of free type variables). Then there is an* SKJ *-definable closure $[\tau]$ whose fixedpoints are exactly the terms of type τ. Moreover, the interpretation $[-]$ is effective.*

For example, the boolean type $[\mathsf{a}\to\mathsf{a}\to\mathsf{a}]$ has five inhabitants $\{\bot,\ \lambda\mathsf{x},\mathsf{y}.\mathsf{x},\ \lambda\mathsf{x},\mathsf{y}.\mathsf{y},\ \lambda\mathsf{x},\mathsf{y}.(\mathsf{x} \mid \mathsf{y}),\ \top\}$. Dana Scott proved a similar definability theorem in [Sco76], showing that simple types are definable using step functions. The advantage of our approach is economy of language: we achieve the same rich type theory using only a finitely generated magma. Cardelli in [Car86] observes that Scott's types-as-closures models ω-order polymorphism and dependent types, although the Curry-Howard corresponding logic is inconsistent. These results do not depend on step functions, and hence also apply to SKJ .

One already well-known motivation for adding a join operation is to achieve a more robust notion of recursively enumerable set, as discussed by Constable and Smith [CS88]. It is a basic theorem of computability theory that the recursively enumerable sets are exactly the recursively semidecidable sets, and that these sets are closed under intersection and union. More generally the equivalence and intersection theorem results extend from SK to arbitrary extensions of SK ; however the proof of closure under union may fail in some extensions. [1] The join operation can be seen as an internalization of the closure-under-unions theorem, allowing a proof of closure that generalizes to arbitrary extensions.

Another motivation for join comes from type theory. Under the Curry-Howard correspondence, the join operation does not prove any new theorems, but allows us to add ambiguity to existing proofs. The human proof idiom most closely corresponding to join is probably the *without-loss-of-generality* construct. But ambiguity and types-as-closures also allows for a much more powerful proof technique, a sort of *semantic type inference* that allows us to raise untyped proof sketches up to complete well-typed proofs using the theorem-as-closure-operator they are intended to prove. We illustrate the WLOG construct here, but delay discussion of semantic type inference until we build up some dependent type theory in 3.14. Then in 6 we present two case studies makeing extensive use of proof sketching and type inference.

To see how the WLOG construct works in SKJ , let us consider a simple theorem.

Theorem 3.0.11. *In a list of three booleans (with values $\mathbf{K}, \mathbf{F}$), at least two items are the same.*

[1] The proof exploits details of β reduction to dovetail two computations in a virtual machine; general extensions may not allow the definition of such a virtual machine.

A purely sequential proof must case-analyze each item.

Proof. Let (x, y, z) be given.
Case: if $x = \mathbf{K}$,
    Subcase: if $y = \mathbf{K}$ then x=y.

    Subcase: if $y = \mathbf{F}$
        Subsubcase: if $z = \mathbf{K}$ then $x = z$.
        Subsubcase: if $z = \mathbf{F}$ then $y = z$.

Case: if $x = \mathbf{F}$,
    Subcase: if $y = \mathbf{K}$
        Subsubcase: if $z = \mathbf{F}$ then $x = z$.
        Subsubcase: if $z = \mathbf{K}$ then $y = z$.

    Subcase: if $y = \mathbf{F}$ then x=y.

□

Since the two cases $x = \mathbf{K}$ and $x = \mathbf{F}$ are symmetric under a $\mathbf{K} \leftrightarrow \mathbf{F}$ reversal, we can simplify the proof to

Proof. Let (x, y, z) be given. Note that the theorem is symmetric under $\mathbf{K} \leftrightarrow \mathbf{F}$ reversal. Thus assume WLOG that $x = \mathbf{K}$.
Case: if $y = \mathbf{K}$ then x=y.

Case: if $y = \mathbf{F}$
    Subcase: if $z = \mathbf{K}$ then $x = z$.

    Subcase: if $z = \mathbf{F}$ then $y = z$.

□

Now consider a Curry-Howard corresponding type.

```
thm  :=  ∀x:bool, y:bool, z:bool.
              Sum (iff x y unit nil).        an equality dependent type
              Sum (iff y z unit nil).
                  iff x z unit nil
```

A purely sequential inhabitant is

```
pf  :=  λx:bool, y:bool, z:bool.
            x ( y (inl ⟨⟩)
                  ( z (inr(inr ⟨⟩))
                      (inr(inl ⟨⟩)) ) )
              ( y ( z (inr(inl ⟨⟩))
                      (inr(inr ⟨⟩)) )
                  (inl ⟨⟩) )
```

but a simpler proof could use a join operation.

```
pf′  :=  (I | λp, x, y, z. p(not x)(not y)(not z))        express symmetry
            λx:bool, y:bool, z:bool.
              x ( y (inl ⟨⟩)                               WLOG x = K
                    ( z (inr(inr ⟨⟩))
                        (inr(inl ⟨⟩)) ) )
                ⊥                                          ignore other cases
```

3.1 Axioms for join-as-ambiguity (J)

```
!using J.
```

We write $\mathbf{J}\ x\ y = x \mid y$ for the ambiguous join operation. This follows Dijkstra's notation for guarded commands, as we can write, e.g.

$$\mathsf{fun} := (\lambda x.\ \mathsf{cond1}\ x\ \mathsf{action1}\ \mid\ \mathsf{cond2}\ x\ \mathsf{action2}\ \mid\ \ldots\ \mid\ \mathsf{condN}\ x\ \mathsf{actionN}).$$

where the conditions either succeed (with value $\mathbf{I}$) or fail (with value $\perp$).

Axioms and axiom schemata

The following axiom shemata are enforced for the atom $\mathbf{J}$:

$$\frac{}{x \mid x = x}\ (\mathsf{idem}) \qquad \frac{}{x \mid y = y \mid x}\ (\mathsf{comm}) \qquad \frac{}{x \mid (y \mid z) = (x \mid y) \mid z}\ (\mathsf{assoc}) \qquad \frac{}{x \mid y \sqsupseteq x}\ (\mathsf{join-L})$$

$$\frac{}{x \mid y \sqsupseteq y}\ (\mathsf{join}-\mathbf{R}) \qquad \frac{z \sqsupseteq x \qquad z \sqsupseteq y}{z \sqsupseteq x \mid y}\ (\mathsf{subconvex}) \qquad \frac{}{(x \mid y)z = x\ z \mid y\ z}\ (\mathsf{distrib}-\mathbf{R})$$

We first assume the semilattice axioms and schemata (idem), comm, and (assoc).

```
!assume x | x = x               AND  J x x = x.               idempotence
!assume x | y = y | x           AND  J x y = J y x.           commutativity
!assume x | (x | y) = x | y     AND  J x(J x y) = J x y.      idempotence+assoc
!assume x | (y | z) = (x | y) | z  AND  J x(J y z) = J(J x y)z.  associativity
!assume Y(J x) = x.                                           idempotence+fixedpoint
```

Next we relate join to the partial order with schemata (join) and (subconvex) and axioms

```
!assume ( x, y ⊑ x | y ).
!assume ⊥ | x = x    AND   J ⊥ x = x    AND   J x ⊥ = x.
!check ⊤ | x = ⊤     AND   J ⊤ x = ⊤    AND   J x ⊤ = ⊤.
```

Finally we assume that join is almost a combinatory algebra homomorphism, in that application right-distributes and almost left-distributes over joins.

```
!assume (x | y)z = x z | y z      AND   J x y z = J(x z)(y z).
!assume (x | y)∘z = x∘z | y∘z     AND   B(J x y) = J(B x)(B y).
!assume x(y | z) ⊒ x y | x z      AND   x(J y z) ⊒ J(x y)(x z).
!assume x∘(y | z) ⊒ x∘y | x∘z     AND   B′(J y z) ⊒ J(B′ y)(B′ z).
```

Note that left-distributivity and left-composition are only one-sided:

```
!check (K | F) ⊤ ⊥ = (K | F) ⊥ ⊤ = ⊤.
!check (
    probe_bool := (λx. x (x ⊥ ⊤) (x ⊤ ⊥)).
    probe_bool K = ⊥     AND
    probe_bool F = ⊥     AND
    probe_bool (K | F) = ⊤
).
!check x(y | z) ⋣ x y | x z.              left-distributivity
!check B′(x | y) ⋣ B′ x | B′ y.           left-composition-distributivity
```

Lemma 3.1.1. *$\top$ can be obtained by joining over all $\mathbf{S}, \mathbf{K}$-terms*

$$\text{!assume } \top = \mathbf{Y}(\lambda t.\mathbf{S} \mid \mathbf{K} \mid t\ t). \qquad \textit{simpler is just } \mathbf{Y}\ \mathbf{J}\textit{, as below}$$

Proof. We need to show $\forall x.\mathbf{Y}(\lambda t.\mathbf{S} \mid \mathbf{K} \mid t\ t) \sqsupseteq x$ for each x. First note that $\mathbf{K}$, $\mathbf{F}$, and hence $\mathbf{J}$ are in the join

$$\text{!check } (\ \mathbf{F}, \mathbf{K}, \mathbf{J}\ \sqsubseteq\ \mathbf{Y}(\lambda t.\mathbf{S} \mid \mathbf{K} \mid t\ t)\ ).$$

so that we're really joining over SKJ -terms

$$\text{!check } \mathbf{Y}(\lambda t.\mathbf{S} \mid \mathbf{K} \mid t\ t) = \mathbf{Y}(\lambda t.\mathbf{S} \mid \mathbf{K} \mid \mathbf{J} \mid t\ t).$$

whence every SKJ term is below the join. □

Lemma 3.1.2.

$$\text{!assume } \top = \mathbf{Y}(\mathbf{J}\ \mathbf{I})\circ\mathbf{K}. \qquad \textit{the ambiguous Ogre}$$

Proof. This uses the representation of $\top$ as the join of all its arguments

$$\begin{aligned}\top &= \mathbf{I} \mid \mathbf{K}\ \mathbf{I} \mid \mathbf{K}\circ\mathbf{K}\ \mathbf{I} \mid \mathbf{K}\circ\mathbf{K}\circ\mathbf{K}\ \mathbf{I} \mid \ldots \\ &= \mathbf{Y}\lambda x.\mathbf{I} \mid \mathbf{K}\ x \\ &= \mathbf{Y}\ (\mathbf{J}\ \mathbf{I})\circ\mathbf{K}\end{aligned}$$

□

Join is injective and has two simple left-inverses

$$\begin{aligned}&\text{!check } \mathbf{Y}\circ\mathbf{J} = \mathbf{I} = \langle\bot\rangle\circ\mathbf{J}.\\ &\text{!check } \mathbf{Y}(\mathbf{J}\ x) = x = \mathbf{J}\ x\ \bot.\end{aligned}$$

The basic combinators distribute over $\mathbf{J}$.

$$\begin{aligned}&\text{!assume } \mathbf{K}(x \mid y) = \mathbf{K}\ x \mid \mathbf{K}\ y.\\ &\text{!assume } \mathbf{F}(x \mid y) = \mathbf{F}\ x \mid \mathbf{F}\ y.\\ &\text{!assume } \mathbf{C}(x \mid y) = \mathbf{C}\ x \mid \mathbf{C}\ y.\\ &\text{!assume } \mathbf{B}(x \mid y) = \mathbf{B}\ x \mid \mathbf{B}\ y.\\ &\text{!assume } \mathbf{W}(x \mid y) = \mathbf{W}\ x \mid \mathbf{W}\ y.\\ &\text{!assume } \mathbf{S}(x \mid y) = \mathbf{S}\ x \mid \mathbf{S}\ y.\\ &\text{!assume } \mathbf{J}(x \mid y) = \mathbf{J}\ x \mid \mathbf{J}\ y.\end{aligned}$$

(these are all easily provable, but we want them in the core theory, so we !assume rather than !check). The parametrized binary join operation will be especially important

$$\text{!define } \mathbf{J}' = (\lambda f, x, y.\ f\ x \mid f\ y).$$

Theorems conjectured by Johann

Lemma 3.1.3.

$$\text{!assume } \mathbf{Y}\ \mathbf{J} = \top.$$

Proof. Expanding, $\mathbf{Y}\ \mathbf{J} = \mathbf{J}(\mathbf{Y}\ \mathbf{J})\ \sqsupseteq\ \mathbf{I}$, so $\mathbf{Y}\ \mathbf{J} = \mathbf{J}(\mathbf{Y}\ \mathbf{J})\ \sqsupseteq\ \mathbf{J}\ \mathbf{I}\ \not\sqsupseteq\ \mathbf{I}$. Continuing this way,

$$\mathbf{Y}\ \mathbf{J} = \mathbf{I} \mid \mathbf{J}\ \mathbf{I} \mid \mathbf{J}\circ\mathbf{J}\ \mathbf{I} \mid \cdots \mid n\ \mathbf{J}\ \mathbf{I} \mid \ldots$$

Now observing $\mathbf{Y}\ \mathbf{J}$ under any trace $\langle x1,\ \ldots,\ xn\rangle$

$$\begin{aligned}\mathbf{Y}\ \mathbf{J}\ x1\ \ldots\ xn\ &\sqsupseteq\ (n+1)\ \mathbf{J}\ \mathbf{I}\ x1\ \ldots\ xn\\ &\sqsupseteq\ \mathbf{J}\ \mathbf{I}\\ &\not\sqsupseteq\ \mathbf{I}\end{aligned}$$

whence $\mathbf{Y}\ \mathbf{J} = \top$. □

Böhm trees and Scott topology in the presence of join

Definition 3.1.4. A *join* is a J-term (closure under J), defined by the language

$$\frac{\mathsf{m\ \ term}}{\mathsf{m\ \ join}}\ (\mathsf{unary}) \qquad \frac{\mathsf{m\ \ join} \qquad \mathsf{n\ \ join}}{\mathsf{m} \mid \mathsf{n\ \ join}}\ (\mathsf{binary}) \qquad \frac{\mathcal{M}\ \ \mathsf{set(terms)}}{\bigsqcup \mathcal{M}\ \ \mathsf{join}}\ (\mathsf{infinitary})$$

For example the nullary join is $\bot = \bigsqcup \{\}$.

Definition 3.1.5. A λ-$\bigsqcup$-term M is in *head normal form* (h.n.f.) if it is of the form

$$\mathsf{M} = \lambda \mathsf{x1}, \ldots, \mathsf{xm}.\ \mathsf{h}\ \ \mathsf{M1}\ \ \ldots\ \ \mathsf{Mn}$$

for $\mathsf{m} \ \geq\ 0$ variables and $\mathsf{n}\ \geq\ 0$ terms. The variable h is called the *head variable* of M.

Requiring the $\mathsf{M1}, \ldots, \mathsf{Mn}$ to also be in head normal form gives rise to the notion of a Böhm tree.

Definition 3.1.6. A *J-Böhm tree* (J-BT or just BT) is the SKJ notion of Böhm tree, defined by limits in the language

$$\frac{\mathsf{x\ \ var}}{\mathsf{x\ \ BT}}\ (\mathsf{var}) \qquad \frac{\underline{\mathsf{x}}\ \ \mathsf{vars} \qquad \mathsf{h\ \ var} \qquad \underline{\mathsf{m}}\ \ \mathsf{join(BT)s}}{\lambda\underline{\mathsf{x}}.\ \mathsf{h}\ \ \underline{\mathsf{m}}\ \ \mathsf{BT}}\ (\mathsf{abs-app})$$

A *finite* BT consists of only finite joins and finitely many applications of rule abs-app.

Theorem 3.1.7. *(Böhm Tree) Every* SKJ *term is $\mathcal{H}^*$-equivalent to a join of J-BTs.*

Proof. By straightforward extension of the BT-theorem of SK . □

The Böhm tree theorem is crucial to our type-definability theorem below, but fails in Scott's D_∞ and $\mathbb{P}(\omega)$ models.

Corollary 3.1.8. *If* q *converges then* q *extends a h.n.f.*

Corollary 3.1.9. *(interpolation) if* $\mathsf{q} \not\sqsubseteq \mathsf{q}'$ *then* $\mathsf{q} \sqsupseteq \mathsf{m} \not\sqsubseteq \mathsf{q}'$ *for some h.n.f.* m.

Corollary 3.1.10. *(approximation) Every* SKJ *term is equivalent to a directed join of some sequence of finite J-BTs (with finite joins).*

SKJ is a much better-behaved algebra than SK , as evidenced by the following. First, SKJ avoids the range property ([Bar93], [Bar08]).

Theorem 3.1.11 (Myhill,Barendregt)**.** *(range property) In* SK *, every non-constant term has infinite range.*

Theorem 3.1.12. *(all ranges) In* SKJ *, every finite cardinality is achieved by the range of some closure.*

Second, SKJ avoids the indefinability of compact points.

Definition 3.1.13. (compactness) An element x of a complete join-semilattice L is a *compact point* iff

$$\forall \mathbf{Y} \subseteq \mathsf{L}.\ \mathsf{x} \sqsubseteq \bigsqcup \mathbf{Y} \implies \exists \mathsf{finite}\ \mathbf{Y}' \subseteq \mathbf{Y}.\ \mathsf{x} \sqsubseteq \bigsqcup \mathbf{Y}'$$

Theorem 3.1.14. *(noncompactness) In* SK *, no term is a compact point.*

Proof. By example: consider the infinite η-expansion of the identity.

$$\mathbf{I} = \lambda x_0.\ \lambda x_1.\ x_0\ \lambda x_2.\ x_1\ \lambda x_3.\ x_2\ \ldots$$

and its truncations at the nth level

$$\mathbf{I}_n = \lambda x_0.\ \lambda x_1.\ x_0\ \lambda x_2.\ x_1\ \lambda x_3.\ x_2\ \ldots\ \lambda x_n.\ \bot$$

Then $\mathbf{I} = \bigsqcup_n \mathbf{I}_n$, but $\mathbf{I}$ is not below any finite join of the $\mathbf{I}_n$s. □

By contrast in SKJ, the Simple type constructor defined in 3.6 furnishes closures with definable compact points.

Theorem 3.1.15. *(compactness) In* SKJ*, there is a closure whose range is a partial numeral system, and every of whose inhabitants is a definable compact point (except for a single limit point ω).*

Proof. Consider nat defined in 3.12. □

Definition 3.1.16. Let M be an SKJ -term. An SKJ -term q is *M- solvable* iff we can solve the equation $q\ M1\ \ldots\ Mn = M$ for some sequence of arguments $M1, \ldots, Mn$. A term is *solvable* iff it is $\top$-solvable.

Contrasting solvability in SK, there are $\top$-solvable SKJ -terms terms that are not **I**-solvable.

Example 3.1.17. $\mathsf{div} = \mathbf{V}\langle\top\rangle$ has range $\{\bot, \top\}$, and since $\bot\ x = \bot$ and $\top\ x = \top$, div cannot be solved for **I**.

3.2 Axioms for a universal retract (U)

Definition 3.2.1. Let $(\mathbf{A}, \sqsubseteq)$ be a poset, $f : \mathbf{A} \to \mathbf{A}$ be an endomorphism. Let us say that f is a *retract* iff $f \circ f \sqsubseteq f$.

We can construct a universal retract operator in SKJ as

```
!define U := (λf. Yλy. f | y∘f).
!check U = Y∘(K | B').
```

An infinitary representation of **U** is

$$\begin{aligned}
\mathbf{U}\ f &= \mathbf{Y}\lambda y.\ f \mid y\circ f \\
&= f \mid (\mathbf{Y}\lambda y.\ f \mid y\circ f)\circ f \\
&= f \mid (\mathbf{U}\ f)\circ f \\
&= f \mid (f \mid (\mathbf{U}\ f)\circ f)\circ f \\
&= f \mid f\circ f \mid (\mathbf{U}\ f)\circ f\circ f \\
&\vdots \\
&= f \mid f\circ f \mid f\circ f\circ f \mid \ldots \qquad = \bigsqcup_{0<n<\omega} f^n
\end{aligned}$$

Theorem 3.2.2. **U** *is a universal retract, i.e.*
(a) **U** *is idempotent:* $\mathbf{U}\circ\mathbf{U} = \mathbf{U}$;
(b) *points in the range of* **U** *are retracts:* $x = \mathbf{U}\ y \implies x\circ x \sqsubseteq x$;
(c) *all retracts are in the image of* **U**: $x\circ x \sqsubseteq x \implies \mathbf{U}\ x = x$.

Proof.

(a) Using the infinitary representation, and rearranging terms of $\mathbf{U}\circ\mathbf{U}$, we have

$$\begin{aligned}\mathbf{U}(\mathbf{U}\ \mathsf{f}) &= \bigsqcup_{n>0} (\mathbf{U}\ \mathsf{f})^n \\ &= \bigsqcup_{n>0} \bigsqcup_{m>0} \mathsf{f}(n\ m) \\ &= \bigsqcup_{n>0} \mathsf{f}^n \\ &= \mathbf{U}\ \mathsf{f}\end{aligned}$$

whence $\mathbf{U}\circ\mathbf{U} = \mathbf{U}$.

(b) Suppose $\mathsf{x} = \mathbf{U}\ \mathsf{y}$ for some y. Since $\mathbf{U}$ is idempotent,

$$\mathsf{x} = \mathbf{U}\ \mathsf{y} = (\mathbf{U}\circ\mathbf{U})\mathsf{y} = \mathbf{U}(\mathbf{U}\ \mathsf{y}) = \mathbf{U}\ \mathsf{x} = \mathsf{x} \mid \mathsf{x}\circ\mathsf{x} \mid \ldots \quad \sqsupseteq \quad \mathsf{x}\circ\mathsf{x}$$

(c) If $\mathsf{x}\circ\mathsf{x} \sqsubseteq \mathsf{x}$ then all terms in the infinitary join collapse into the first term: $\mathbf{U}\ \mathsf{x} = \mathsf{x}$, as required.

□

Axioms and axiom schemata

The following axiom shemata are enforced for the atom $\mathbf{U}$:

$$\frac{\mathsf{x} = \mathbf{U}\ \mathsf{x}}{\mathsf{x}\circ\mathsf{x} \sqsubseteq \mathsf{x}}\ (\mathsf{retract}) \qquad \frac{\mathsf{x}\circ\mathsf{x} \sqsubseteq \mathsf{x}}{\mathsf{x} = \mathbf{U}\ \mathsf{x}}\ (\mathsf{retracts}) \qquad \frac{\mathsf{f}\ \mathsf{x} \sqsubseteq \mathsf{x}}{\mathbf{U}\ \mathsf{f}\ \mathsf{x} = \mathsf{f}\ \mathsf{x}}\ (\mathsf{fixed})$$

Remark. Together the retract, retracts schemata imply that $\mathbf{U}$ fixes exactly the retracts.

Question 3.2.3. *Is the* fixed *schema necessary?*

$\mathbf{U}$ is a closure

| !assume $\mathbf{I} \sqsubseteq \mathbf{U} = \mathbf{U}\circ\mathbf{U}$.

whose images are retracts

| !assume $(\mathbf{U}\ \mathsf{f})\circ(\mathbf{U}\ \mathsf{f}) \sqsubseteq (\mathbf{U}\ \mathsf{f})$.

An algebraic characterization:

Lemma 3.2.4.

| !assume $\mathbf{U}\ \mathsf{f} = \mathsf{f} \mid (\mathbf{U}\ \mathsf{f})\circ\mathsf{f}$.

Proof. Using the infinitary representation

$$\begin{aligned}\mathbf{U}\ \mathsf{f} = \mathsf{f} \mid \mathsf{f}\circ\mathsf{f} \mid \mathsf{f}\circ\mathsf{f}\circ\mathsf{f} \mid \ldots \\ = \mathsf{f} \mid (\mathsf{f} \mid \mathsf{f}\circ\mathsf{f} \mid \ldots)\circ\mathsf{f} \\ = \mathsf{f} \mid (\mathbf{U}\ \mathsf{f})\circ\mathsf{f}\end{aligned}$$

□

Indeed $\mathbf{U}$ is the least solution to the above, by definition. However

| !check $\mathbf{U}\ \mathsf{f} \neq \mathsf{f} \mid \mathsf{f}\circ(\mathbf{U}\ \mathsf{f})$.

Derived properties

Regarding types-as-idempotents,

```
!check U ⊒ Y∘B.
!check U = U U.          a type of types
!check I = U I.          a maximum type –everything is fixed
!check K x = U(K x).     minimal types / singletons –only x is fixed
```

Lemma 3.2.5. *Expnentials of idempotents are idempotents*

```
!assume (U a → U b) = U (U a → U b).
```

Proof. For any a, b, f, by definition $(\mathbf{U}\ \mathsf{a}){\rightarrow}(\mathbf{U}\ \mathsf{b})\ \mathsf{f} = (\mathbf{U}\ \mathsf{b}){\circ}\mathsf{f}{\circ}(\mathbf{U}\ \mathsf{a})$. To show retraction,

$$\begin{array}{rcll} 2\ (\mathbf{U}\ \mathsf{a}){\rightarrow}(\mathbf{U}\ \mathsf{b})\ \mathsf{f} & = & (2.\ \mathbf{U}\ \mathsf{b}){\circ}\mathsf{f}{\circ}(2.\ \mathbf{U}\ \mathsf{a}) & \textit{def of}{\rightarrow} \\ & \sqsubseteq & (\mathbf{U}\ \mathsf{b}){\circ}\mathsf{f}{\circ}(\mathbf{U}\ \mathsf{a}) & \textit{since } (\mathbf{U}\ \mathsf{a}), (\mathbf{U}\ \mathsf{b}) : \mathbf{U} \\ & = & (\mathbf{U}\ \mathsf{a}){\rightarrow}(\mathbf{U}\ \mathsf{b})\ \mathsf{f} & \textit{def of}{\rightarrow} \end{array}$$

□

We can now represent **J** in terms of **U**:

```
!check J = K | F = U C K = U C F.
```

3.3 Axioms for a universal closure (V)

Definition 3.3.1. Let $\langle \mathbf{A}, [\geq$ be a poset, $\mathsf{f} : \mathbf{A} \to \mathbf{A}$ be an endomorphism. Let us say that f is a *closure* iff $\mathsf{f}{\circ}\mathsf{f} = \mathsf{f} \sqsupseteq \mathbf{I}$, where **I** is the identity on **A**.

We construct the universal closure from the universal idempotent

```
!define V := (λa. U(I | a)).
!check V = (λa. U(a | I)).
!check V a = U(I | a).
```

An infinitary representation of **V** follows from that of **U**

$$\begin{array}{rcl} \mathbf{V}\ \mathsf{a} & = & (\mathsf{a} \mid \mathbf{I}) \ \mid\ (\mathsf{a} \mid \mathbf{I}){\circ}(\mathsf{a} \mid \mathbf{I}) \ \mid\ \ldots \\ & = & \mathbf{I} \ \mid\ (\mathsf{a} \mid \mathbf{I}) \ \mid\ (\mathsf{a} \mid \mathbf{I}){\circ}(\mathsf{a} \mid \mathbf{I}) \ \mid\ \ldots \\ & = & \bigsqcup\limits_{n<\omega} (\mathsf{a} \mid \mathbf{I})^n \end{array}$$

Theorem 3.3.2. **V** *is a universal closure, i.e.*

(a) **V** *is a closure:* $\mathbf{V} \sqsupseteq \mathbf{I},\ \mathbf{V}{\circ}\mathbf{V} = \mathbf{V}$;

(b) points in the range of **V** *are closures:* $\mathsf{x} = \mathbf{V}\ \mathsf{y} \implies \mathsf{x} \sqsupseteq \mathbf{I},\ \mathsf{x}{\circ}\mathsf{x} = \mathsf{x}$;

(c) all closures are in the image of **V**: $\mathsf{x} \sqsupseteq \mathbf{I},\ \mathsf{x}{\circ}\mathsf{x} = \mathsf{x} \implies \mathbf{V}\ \mathsf{x} = \mathsf{x}$;

Proof.

(a) Using the infinitary representation, $\mathbf{V} \sqsupseteq \mathbf{I}$ is immediate. Rearranging terms of $\mathbf{V}{\circ}\mathbf{V}$, we have

$$\begin{array}{rcl} \mathbf{V}(\mathbf{V}\ \mathsf{a}) & = & \bigsqcup\limits_{n<\omega} (\mathbf{V}\ \mathsf{a} \mid \mathbf{I})^n \\ & = & \bigsqcup\limits_{n<\omega} \bigsqcup\limits_{m<\omega} (\mathsf{a} \mid \mathbf{I})^{n\ m} \\ & = & \bigsqcup\limits_{n<\omega} (\mathsf{a} \mid \mathbf{I})^n \\ & = & \mathbf{V}\ \mathsf{a} \end{array}$$

whence $\mathbf{V}{\circ}\mathbf{V} = \mathbf{V}$.

(b) Suppose $x = \mathbf{V}\ a$ for some a. By the infinitary representation $x = \mathbf{V}\ a = \mathbf{I} \mid \ldots \sqsupseteq \mathbf{I}$. Since $\mathbf{V}$ is idempotent, and since $x \sqsupseteq \mathbf{I}$,

$$x \quad = \mathbf{V}\ a \quad = \mathbf{V}(\mathbf{V}\ a) \quad = \mathbf{V}\ x \quad \sqsupseteq \mathbf{I} \mid x \mid x{\circ}x \mid \ldots \quad \sqsupseteq x{\circ}x$$

On the other hand, $x = \mathbf{I}{\circ}x \sqsubseteq x{\circ}x$, whence $x = x{\circ}x$.

(c) If $x \sqsupseteq \mathbf{I}$ and $x = x{\circ}x$ then also

$$(x \mid \mathbf{I}){\circ}(x \mid \mathbf{I}) \quad = x{\circ}x \quad = x \quad = x \mid \mathbf{I}$$

so the infinitary representation of $\mathbf{V}\ x$ collapses to $(\mathbf{I} \mid x) = x$.

□

We have in addition a variety of equivalent definitions, e.g. Scott's ([Sco76]),

| !check $\mathbf{V} = (\lambda a, x.\ \mathbf{Y}\lambda y.\ x \mid a\ y)$. $\quad = (\mathbf{B}\ \mathbf{Y}){\circ}(\mathbf{C}\ \mathbf{B}{\circ}\mathbf{J})$

and as a least fixed-point,

| !check $\mathbf{V} = \mathbf{Y}(\lambda y, a.\ \mathbf{I} \mid a{\circ}(y\ a))$.

Types as closures

Closures allow an interpretation of various typed λ calculi in the untyped calculus SKJ . In particular, this provides convenient notation.

Definition 3.3.3. (types as closures)

$$\begin{array}{rcll} (\lambda x{:}a.\ M) & = & (\lambda x.\ M){\circ}(\mathbf{V}\ a) & \textit{typed abstraction} \\ x{:}a & \iff & \mathbf{V}\ a\ x = x & \textit{type inhabitation} \\ \forall x{:}a.\ M = N & \iff & (\lambda x{:}a.\ M) = (\lambda x{:}a.\ N) & \textit{universal closure of equations} \\ a <: b & \iff & \mathbf{V}\ a{:}\mathbf{P}\ b & \textit{subtyping} \end{array}$$

where the powertype $\mathbf{P}$ will be discussed in 3.4.

Lemma 3.3.4.

| !check $(\forall x{:}a.\ x{:}a)$.

Proof.

$$\begin{array}{rcl} \forall x{:}a.\ x{:}a & \iff & \forall x{:}a.\ \mathbf{V}\ a\ x = x \\ & \iff & (\lambda x{:}a.\ \mathbf{V}\ a\ x) = (\lambda x{:}a.\ x) \\ & \iff & (\lambda x.\ \mathbf{V}\ a(\mathbf{V}\ a\ x)) = (\lambda x.\ \mathbf{V}\ a\ x) \\ & \iff & \mathbf{V}\ a(\mathbf{V}\ a\ x) = \mathbf{V}\ a\ x \end{array}$$

□

Lemma 3.3.5. *Subtyping is a bounded preorder, and a partial order on closures.*
(a) $\mathsf{nil} <: a <: a <: \mathsf{any}$.
(b) *If* $a <: b$ *and* $b <: c$ *then* $a <: c$.
(c) *For* $a, b{:}\mathbf{V}$, *if* $a <: b$ *and* $b <: a$ *then* $a = b$.

We will prove this later in 3.4 we know more properties of $\mathbf{V}$ and $\mathbf{P}$.

A quirk of this system is that exponentials have unusual variance: they are covariant in both arguments (as discussed later in 3.7).

Later in 3.10 we will consider a similar interpretation of types-as-tests.

Axioms and axiom schemata

The following axiom shemata are enforced for the atom $\mathbf{V}$:

$$\frac{\mathsf{x}:\mathbf{V}}{\mathsf{x}\circ\mathsf{x}=\mathsf{x}}\ (\mathsf{idem})\qquad \frac{\mathsf{x}:\mathbf{V}}{\mathsf{x}\sqsupseteq\mathbf{I}}\ (\mathsf{incr})\qquad \frac{\mathsf{x}\circ\mathsf{x}=\mathsf{x}\qquad \mathsf{x}\sqsupseteq\mathbf{I}}{\mathsf{x}:\mathbf{V}}\ (\mathsf{closures})\qquad \frac{\mathsf{f}\ \mathsf{x}\sqsubseteq\mathsf{x}}{\mathbf{V}\ \mathsf{f}\ \mathsf{x}=\mathsf{x}}\ (\mathsf{fixed})$$

Remark. Together the idem, incr, closures schemata imply that $\mathbf{V}$ fixes exactly the closures.

Question 3.3.6. *Is the* fixed *schema necessary?*

Now we assume $\mathbf{V}$ is a closure

| !assume $\mathbf{I}\ \sqsubseteq\ \mathbf{V}=\mathbf{V}\circ\mathbf{V}$.

whose images are closures

| !assume $\mathbf{I}\ \sqsubseteq\ \mathbf{V}\ \mathsf{a}=(\mathbf{V}\ \mathsf{a})\circ(\mathbf{V}\ \mathsf{a})$.

and hence $\mathbf{V}$ fixes itself.

| !check $\mathbf{V}=\mathbf{V}\ \mathbf{V}$.

Lemma 3.3.7. *The* a *in* $\mathbf{V}$ a *is* `w. l.`*o.g. increasing.*

| !assume $\mathbf{V}\ \mathsf{a}=\mathbf{V}(\mathbf{I}\mid\mathsf{a})$.

Proof. $\mathbf{V}(\mathbf{I}\mid\mathsf{a})=\mathbf{U}(\mathbf{I}\mid\mathbf{I}\mid\mathsf{a})=\mathbf{U}(\mathbf{I}\mid\mathsf{a})=\mathbf{V}\ \mathsf{a}$ □

Theorem 3.3.8. *(algebraic characterization)*

| !assume $\mathbf{V}\ \mathsf{a}=\mathbf{I}\mid\mathsf{a}\circ(\mathbf{V}\ \mathsf{a})$.

Proof.

$$\begin{aligned}\mathbf{V}\ \mathsf{a}\ &=\ \lambda\mathsf{x}.\ \mathbf{Y}\lambda\mathsf{y}.\ \mathsf{x}\mid\mathsf{a}\ \mathsf{y}\\ &=\ \lambda\mathsf{x}.\ \mathsf{x}\mid\mathsf{a}(\mathbf{Y}\lambda\mathsf{y}.\ \mathsf{x}\mid\mathsf{a}\ \mathsf{y})\\ &=\ \lambda\mathsf{x}.\ \mathsf{x}\mid\mathsf{a}(\mathbf{V}\ \mathsf{a}\ \mathsf{x})\\ &=\ \mathbf{I}\mid\mathsf{a}\circ(\mathbf{V}\ \mathsf{a})\end{aligned}$$

□

Indeed $\mathbf{V}$ is the least solution to the above, by definition.
Note that

| !assume $\mathbf{V}=2\circ\mathbf{V}$.

however the transpose fails

Theorem 3.3.9. $\mathbf{V}\neq\mathbf{V}\circ 2$

First a lemma

Lemma 3.3.10.

| !assume $\mathbf{V}\ \mathbf{C}=\mathbf{C}\mid\mathbf{I}$.

Proof. Since $\mathbf{C}\circ\mathbf{C} = \mathbf{I}$, also

$$\begin{aligned}(\mathbf{C} \mid \mathbf{I})\circ(\mathbf{C} \mid \mathbf{I}) &= \lambda x.\ (\mathbf{C} \mid \mathbf{I})(\mathbf{C}\ x \mid x)\\ &= \lambda x.\ \mathbf{C}\circ\mathbf{C}\ x \mid \mathbf{C}\ x \mid x\\ &= \lambda x.\ x \mid \mathbf{C}\ x \mid x\\ &= \lambda x.\ \mathbf{C}\ x \mid x\\ &= \mathbf{C} \mid \mathbf{I}\end{aligned}$$

whence $\mathbf{C} \mid \mathbf{I}$ is a closure. Hence $\mathbf{V}\ \mathbf{C} = \mathbf{V}(\mathbf{C} \mid \mathbf{I}) = \mathbf{C} \mid \mathbf{I}$. □

We can now prove the theorem:

Proof.

```
!check 2 C = I.
!check I : V.
!check C !: V.          i.e., C is not fixed by V
!check V ≠ V∘2.
```

□

Derived properties

```
!check V ⊒ Y∘B.
!check V : V.
```

Note that $\top$ inhabits every closure

```
!check (∀a:V. ⊤:a).
```

Lemma 3.3.11.

```
!assume (V a)→(V b) : V.        exponential
```

Proof. for any a, b, f, by definition $(\mathbf{V}\ a)\to(\mathbf{V}\ b)\ f = (\mathbf{V}\ b)\circ f\circ(\mathbf{V}\ a)$. To show idempotence,

$$\begin{aligned}2\ (\mathbf{V}\ a)\to(\mathbf{V}\ b)\ f &= (2.\ \mathbf{V}\ b)\circ f\circ(2.\ \mathbf{V}\ a) && \textit{def of}\to\\ &= (\mathbf{V}\ b)\circ f\circ(\mathbf{V}\ a) && \textit{since } (\mathbf{V}\ a), (\mathbf{V}\ b):\mathbf{V}\\ &= (\mathbf{V}\ a)\to(\mathbf{V}\ b)\ f && \textit{def of}\to\end{aligned}$$

Now to show closure,

$$(\mathbf{V}\ b)\circ f\circ(\mathbf{V}\ a) \quad\sqsupseteq\quad \mathbf{I}\circ f\circ\mathbf{I} \quad=\quad f$$

whence $(\mathbf{V}\ a)\to(\mathbf{V}\ b) \sqsupseteq \mathbf{I}$. □

We can now represent $\mathbf{J}$ in terms of $\mathbf{V}$:

```
!check J = K | F.
!check J = V C K.
!check J = V C F.
```

Note that

Lemma 3.3.12.

```
!assume V J = ⊤.
```

Proof. Suppose $q:\mathbf{V}\ \mathbf{J}$. Then $q\ \top = \mathbf{J}\ q\ \top = \top$, so q is solvable. But $q\ \bot = \mathbf{J}\ q\ \bot = q$, so q is undivergable. Hence q is $\top$. □

Miscellaneous closures

The Maximal type.

$$
\begin{array}{|l}
\mathsf{any} \;:=\; \mathbf{I}. \\
\mathsf{!check\ any} \;:\; \mathbf{V}. \\
\mathsf{!check\ any} = \mathbf{V}\ \bot. \\
\mathsf{!check}\ \mathbf{V} \;:\; \mathsf{any}\to\mathbf{V}. \\
\mathsf{!check\ any} = \mathsf{any}\to\mathsf{any}. \\
\mathsf{!check}\ (\forall \mathsf{a}{:}\mathbf{V}.\ (\mathsf{a}\to\mathsf{any}, \mathsf{any}\to\mathsf{a}, \mathsf{a}\to\mathsf{a}\ :\ \mathbf{V}\ )). \\
\mathsf{!check}\ (\forall \mathsf{a}{:}\mathbf{V}, \mathsf{b}{:}\mathbf{V}.\ \mathsf{a}\to\mathsf{b}{:}\mathbf{V}).
\end{array}
$$

whence a reflexive object

Theorem 3.3.13. $\mathrm{inhab}(\mathsf{any}) = $ *"everything"*.

Proof. $\mathsf{any}\ \mathsf{x} = \mathbf{I}\ \mathsf{x} = \mathsf{x}$. □

The Minimal type nil has no consistent inhabitants

$$
\begin{array}{|l}
\mathsf{nil} \;:=\; \top. \\
\mathsf{!check\ nil} \;:\; \mathbf{V}. \\
\mathsf{!check}\ \bot\ \mathsf{!:}\ \mathsf{nil}.
\end{array}
$$

Theorem 3.3.14. $\mathrm{inhab}(\mathsf{nil}) = \{\top\}$.

Proof.

$$
\begin{array}{|l}
\mathsf{!check}\ (\forall \mathsf{x}{:}\mathsf{nil}.\ \mathsf{x} = \mathsf{nil}\ \mathsf{x} = \top\ \mathsf{x} = \top).
\end{array}
$$

□

We use the following lemma often.

Lemma 3.3.15. $\mathsf{x}{:}\mathbf{V}\ \mathsf{t} \iff \mathsf{t}\ \mathsf{x} \sqsubseteq \mathsf{x}$.

Proof.

- $(\implies)$

$$
\begin{aligned}
\mathsf{x} &= \mathbf{V}\ \mathsf{t}\ \mathsf{x} \\
&= \mathsf{x} \mid \mathsf{t}\ \mathsf{x} \mid 2\ \mathsf{t}\ \mathsf{x} \mid \ldots \\
&= \mathsf{x} \mid \mathsf{t}\ \mathsf{x}
\end{aligned}
$$

  whence $\mathsf{t}\ \mathsf{x} \sqsubseteq \mathsf{x}$.

- $(\impliedby)$ since $(\mathbf{I} \mid \mathsf{t})\mathsf{x} = \mathsf{x} \mid \mathsf{t}\ \mathsf{x} = \mathsf{x}$,

$$
\begin{aligned}
\mathbf{V}\ \mathsf{t}\ \mathsf{x} &= \mathbf{V}\ (\mathbf{I} \mid \mathsf{t})\ \mathsf{x} \\
&= \mathsf{x} \;\mid\; (\mathbf{I} \mid \mathsf{t})\mathsf{x} \;\mid\; 2(\mathbf{I} \mid \mathsf{t})\mathsf{x} \;\mid\; \ldots \\
&= \mathsf{x} \;\mid\; \mathsf{x} \mid \mathsf{t}\ \mathsf{x} \;\mid\; (\mathbf{I} \mid \mathsf{t})(\mathsf{x} \mid \mathsf{t}\ \mathsf{x}) \;\mid\; \ldots \\
&= \mathsf{x} \;\mid\; \mathsf{x} \;\mid\; (\mathbf{I} \mid \mathsf{t})\mathsf{x} \;\mid\; \ldots \\
&= \mathsf{x} \;\mid\; \mathsf{x} \;\mid\; \mathsf{x} \;\mid\; \ldots \\
&= \mathsf{x}
\end{aligned}
$$

The type of symmetric binary functions.

Lemma 3.3.16.

$$
\begin{array}{|l}
\mathsf{!assume}\ \mathbf{V}\ \mathbf{C} = \mathbf{C} \mid \mathbf{I}.
\end{array}
$$

Proof. $\mathbf{C}\circ\mathbf{C} = \mathbf{I}$, so expanding, $\mathbf{V}\ \mathbf{C} = \mathbf{I} \mid \mathbf{C} \mid \mathbf{C}\circ\mathbf{C}\circ(\mathbf{V}\ \mathbf{C}) = \mathbf{I} \mid \mathbf{C} \mid \mathbf{V}\ \mathbf{C} = \mathbf{I} \mid \mathbf{C}$.

Closure for upper sets are also definable as

$$\begin{array}{l} \mathsf{Above}\ :=\ \mathbf{J}. \\ \mathsf{!assume}\ \mathsf{Above}\ :\ \mathsf{any}\to\mathbf{V}. \qquad \textit{upper-sets are types} \end{array}$$

Proof. Idemopotence and closure are inherited from $\mathbf{J}$:

$$\begin{array}{rcll} \mathbf{J}\ \mathsf{a}\ \mathsf{x} & \sqsupseteq & \mathsf{x} & \textit{whence}\quad \mathbf{J}\ \mathsf{a}\ \sqsupseteq\ \mathbf{I} \\ \mathbf{J}\ \mathsf{a}(\mathbf{J}\ \mathsf{a}\ \mathsf{x}) & = & \mathbf{J}(\mathbf{J}\ \mathsf{a}\ \mathsf{a})\mathsf{x} & \textit{by associativity of } \mathbf{J} \\ & = & \mathbf{J}\ \mathsf{a}\ \mathsf{x} & \textit{by idempotence of } \mathbf{J} \end{array}$$

Lemma 3.3.17.

$$\mathsf{!check}\ \mathbf{V}\ \Delta\ :\ \mathbf{V}\ \mathbf{W}.$$

Proof. Suppose $\Delta\ \mathsf{x} \sqsubseteq \mathsf{x}$. Then

$$\mathbf{W}(\mathbf{V}\ \Delta)\mathsf{x} = \mathbf{V}\ \Delta\ \mathsf{x}\ \mathsf{x} = \mathsf{x}\ \mathsf{x}\ \sqsubseteq\ \mathsf{x}\ \sqsubseteq\ \mathbf{V}\ \Delta\ \mathsf{x}$$

3.4 Axioms for a power closure (P)

The space of closures has structure beyond that of the retracts in general: it is a complete meet-semilattice with bottom $\mathsf{nil} = \top$ and top $\mathsf{any} = \mathbf{I}$. We use symbols $\mathbf{P}\ \mathsf{x}\ \mathsf{y}$ for the meet operation (type intersection) and $\mathsf{x} <: \mathsf{y}$ for the partial ordering (subtyping). The subtype relation is defined generally as

$$\begin{array}{l} \forall \mathsf{x}{:}\mathbf{V}, \mathsf{y}{:}\mathbf{V}. \\ \qquad \begin{array}{rcll} \mathsf{x} <: \mathsf{y} & \iff & \mathsf{inhabs}(\mathsf{x})\ \subseteq\ \mathsf{inhabs}(\mathsf{y}) & \textit{containment of inhabitants} \\ & \iff & (\forall \mathsf{z}.\ \mathsf{z} = \mathsf{x}\ \mathsf{z}\ \implies\ \mathsf{z} = \mathsf{y}\ \mathsf{z}) & \\ & \iff & \mathsf{x}{:}\mathsf{y}\to\mathsf{y} & \\ & \iff & \mathsf{y}\circ\mathsf{x} = \mathsf{x} = \mathsf{x}\circ\mathsf{y} & \textit{explicitly} \end{array} \end{array}$$

Remark. Although idempotents are also a bounded poset with universal object (with top $\mathsf{any} = \mathbf{I}$, bottoms $\mathbf{K}\ \mathsf{x}$ for any x, and universe idem), no meet operation is definable for idempotents.

The type intersection operator can thus be defined as

$$\mathsf{!define}\ \mathbf{P}\ :=\ (\lambda \mathsf{x}, \mathsf{y}.\ \mathbf{V}(\mathsf{x} \mid \mathsf{y})).$$

so that more generally, $\mathsf{x} <: \mathsf{y} \iff \mathbf{V}\ \mathsf{x}{:}\mathbf{P}\ \mathsf{y}$, i.e., x is a subtype of y.

Among types ordered by information content, $\mathbf{P}\ \mathsf{x}\ \mathsf{y}$ is the least upper bound of types $\sqsubseteq \mathsf{x}, \mathsf{y}$, i.e., the join $\mathbf{J}$ restricted to the lattice of types. However, the usual ordering $<:$ on types is *dual* to the information ordering, and thus we regard $\mathbf{P}$ as an intersection operator.

Axioms and axiom schemata

The following axiom shemata are enforced for the atom $\mathbf{P}$:

$$\frac{}{\mathbf{P}\ \mathsf{x}\ \mathsf{x} = \mathbf{V}\ \mathsf{x}}\ (\mathsf{idem}) \qquad \frac{}{\mathbf{P}\ \mathsf{x}\ \mathsf{y} = \mathbf{P}\ \mathsf{y}\ \mathsf{x}}\ (\mathsf{comm}) \qquad \frac{}{\mathbf{P}\ \mathsf{x}(\mathbf{P}\ \mathsf{y}\ \mathsf{z}) = \mathbf{P}(\mathbf{P}\ \mathsf{x}\ \mathsf{y})\mathsf{z}}\ (\mathsf{assoc})$$

Algebraic properties

```
!assume P x x = V x.                  idempotence
!assume P x(P x y) = P x y.           idempotence+associativity
!assume Y(P x) = V x.                 idempotence+fixedpoint
!assume P x y = P y x.                commutativity
!assume P x(P y z) = P(P x y)z.       associativity
```

Idempotence and commutativity are clear, but associativity is nontrivial.

Lemma 3.4.1.

```
!assume P x y = P x(V y) = P(V x)y = P(V x)(V y).
```

Proof.

$$\begin{aligned}\mathbf{P}\ x\ y &= \mathbf{V}(x \mid y)\\ &= \mathbf{V}\circ\mathbf{V}(x \mid y)\\ &\sqsupseteq \mathbf{V}(\mathbf{V}\ x \mid \mathbf{V}\ y) && \textit{since } f(x \mid y) \sqsupseteq f\ x \mid f\ y\\ &\sqsupseteq \mathbf{V}(x \mid y) && \textit{since } \mathbf{V} \sqsupseteq \mathbf{I}\\ &= \mathbf{P}\ x\ y\end{aligned}$$

□

Theorem 3.4.2. **P** *defines a type intersection operator.*

```
!assume P x y:V.
!assume P : V→V→V.                     type intersection operator
!check (∀a:V, b:V. P a b <: a).
!check (∀a:V, b:V. P a b <: b).
!check P any x = V x.
!check P nil x = nil.
```

Theorem 3.4.3. **P** *defines a power-type operator.*

```
!assume P x : V.
!assume P : V→V.          powertype operator
```

Proof. (associativity)

$$\begin{aligned}\mathbf{P}\ x(\mathbf{P}\ y\ x) &= \mathbf{V}(x \mid (\mathbf{V}(y \mid z)))\\ &= \mathbf{V}(\mathbf{V}\ x \mid \mathbf{V}(\mathbf{V}\ y \mid \mathbf{V}\ z))\\ &= \mathbf{V}(\mathbf{V}\ x \mid \mathbf{V}\ y \mid \mathbf{V}\ z)\\ &= \mathbf{V}(\mathbf{V}(\mathbf{V}\ x \mid \mathbf{V}\ y) \mid \mathbf{V}\ z)\\ &= \mathbf{V}(\mathbf{V}(x \mid y) \mid z\\ &= \mathbf{P}(\mathbf{P}\ x\ y)z\end{aligned}$$

□

We also assume the order axioms

```
!assume P x ⊒ V.
!assume P ⊥ x = P I x = V x.
!check P ⊤ x = ⊤.
```

Derived properties

We can now prove the subtyping lemma from 3.3.

Lemma 3.4.4. *Subtyping is a bounded preorder, and a partial order on closures.*

(a) $\mathsf{nil} <: \mathsf{a} <: \mathsf{a} <: \mathsf{any}$.
(b) *If* $\mathsf{a} <: \mathsf{b}$ *and* $\mathsf{b} <: \mathsf{c}$ *then* $\mathsf{a} <: \mathsf{c}$.
(c) *For* $\mathsf{a}, \mathsf{b} : \mathbf{V}$, *if* $\mathsf{a} <: \mathsf{b}$ *and* $\mathsf{b} <: \mathsf{a}$ *then* $\mathsf{a} = \mathsf{b}$.

Proof. (a) Deferring to Johann,

```
!check nil <: a <: a <: any.
```

(b) Assume $\mathsf{a} <: \mathsf{b}$ and $\mathsf{b} <: \mathsf{c}$. Then

$$\begin{aligned}
\mathbf{V}\ \mathsf{a} &= \mathbf{P}\ \mathsf{a}\ \mathsf{b} && \textit{since } \mathsf{a} <: \mathsf{b}\\
&= \mathbf{P}\ \mathsf{a}\ (\mathbf{V}\ \mathsf{b})\\
&= \mathbf{P}\ \mathsf{a}\ (\mathbf{P}\ \mathsf{b}\ \mathsf{c}) && \textit{since } \mathsf{b} <: \mathsf{c}\\
&= \mathbf{P}(\mathbf{P}\ \mathsf{a}\ \mathsf{b})\mathsf{c}\\
&= \mathbf{P}\ \mathsf{a}\ \mathsf{c} && \textit{since } \mathsf{a} <: \mathsf{b}
\end{aligned}$$

whence $\mathsf{a} <: \mathsf{c}$.

(c) Assume $\mathsf{a}, \mathsf{b} : \mathbf{V}$, $\mathsf{a} <: \mathsf{b}$, and $\mathsf{b} <: \mathsf{a}$. Then

$$\mathsf{a} = \mathbf{V}\ \mathsf{a} = \mathbf{P}\ \mathsf{a}\ \mathsf{b} = \mathbf{V}\ \mathsf{b} = \mathsf{b}$$

□

```
!check Y∘P = V.
!check P ⊥ = V.
!check P : I→I→V.        intersections are types
```

Lemma 3.4.5.

```
!check P ⊒ J.
!check P ⊒ K∘V.
```

Proof.

- $\mathbf{P} = \lambda x, y.\ \mathbf{V}(x \mid y) \sqsupseteq \lambda x, y.\ x \mid y = \mathbf{J}$.
- $\mathbf{P} = \lambda x, y.\ \mathbf{P}\ x\ y \sqsupseteq \lambda x, y.\ \mathbf{V}\ y = \mathbf{K}\circ\mathbf{V}$.

□

Theorems conjectured by Johann

Lemma 3.4.6.

```
!check ⊤ = Y P = V P = P J = P P.
```

Proof. $\mathbf{P} \sqsupseteq \mathbf{J}$, and $\mathbf{Y}\ \mathbf{J} = \mathbf{V}\ \mathbf{J} = \top$. □

Lemma 3.4.7.

```
!assume B ⊑ P.
```

Proof.

$$\begin{array}{rcl} \mathbf{P}\ \mathsf{x}\ \mathsf{y} & = & \mathbf{V}\ (\mathbf{V}\ \mathsf{x} \mid \mathbf{V}\ \mathsf{y}) \\ & \sqsupseteq & \mathbf{V}\ (\mathsf{x} \mid \mathsf{y}) \\ & \sqsupseteq & 2\ (\mathsf{x} \mid \mathsf{y}) \\ & = & (\mathsf{x} \mid \mathsf{y}) \circ (\mathsf{x} \mid \mathsf{y}) \\ & \sqsupseteq & \mathsf{x} \circ \mathsf{y} \\ & = & \mathbf{B}\ \mathsf{x}\ \mathsf{y} \end{array}$$

□

3.5 Axioms for the least strict closure (div)

The type div of divergent computations is the minimal strict type, inhabited by only $\{\bot, \top\}$.

```
!define div := V ⟨⊤⟩.
!check ⊥ : div.
!check I !: div.
```

Theorem 3.5.1. $\text{inhab}(\mathsf{div}) = \{\bot, \top\}$

Proof. By definition of solvability in $\mathcal{H}^*$. □

Corollary 3.5.2. M *converges iff* $\mathsf{M} \neq \bot$ *iff* $\mathsf{div}\ \mathsf{M} \neq \top$.

The following axiom schemata are enforced for the atom div:

$$\frac{\mathsf{x}\ \not\sqsubseteq\ \bot}{\mathsf{div}\ \mathsf{x} = \top} \qquad\qquad \frac{\mathsf{x}\ \bot = \bot}{\mathsf{div}\ \sqsubseteq\ \mathsf{x}}$$

The latter holds since div is the largest strict function.

Remark. This is the first appearance of a disjunctive axiom, i.e., $\mathsf{x} \sqsubseteq \bot$ *or* $\top \sqsubseteq \mathsf{div}\ \mathsf{x}$, and thus the first axiom sensitive to language extension. Indeed this axiom fails in SKRJ (see 4.2).

Corollary 3.5.3. *Every inhabitant of* div *is constant.*

Thus we can assume

```
!assume div x ⊥ = div x y = div x z = div x ⊤.
```

Lemma 3.5.4.

```
!assume div = V K.
```

Proof. $\bot$ is fixed by $\mathbf{K}$, so let q : div be any convergent term, say solvable by $\mathsf{q}\ \underline{\mathsf{M}} = \top$.
Then $\mathsf{q}\ \bot\ \sqsupseteq\ \mathbf{K}\ \mathsf{q}\ \bot = \mathsf{q}$, so q can't be crashed. □

Lemma 3.5.5. *The closures of the following combinators are strict, i.e., are supertypes of* div.

```
!check ( B, C, W, S, S′, Φ, Ψ :> div ).
```

Proof. div has inhabitants $\bot, \top$, each of which is fixed by each of $\mathbf{B}, \mathbf{C}, \mathbf{W}, \mathbf{S}, \mathbf{S}', \Phi, \Psi$. □

3.6 A simple type constructor (Simple)

In this section we construct for each simple type τ an SKJ -definable closure $[\tau]$ whose fixedpoints (mod $\mathcal{H}^*$) are exactly the SKJ -terms typable by τ (for a suitable generalization of Curry-style typability from SK to SKJ , mod $\mathcal{H}^*$). The construction takes the form of (1) a translation from simple types to SK -definable type codes, and (2) an SKJ -definable term Simple that binds each free variable in the simple type. For example

$$[(a \to a) \to (a \to b) \to a \to b] = \mathsf{Simple}\lambda a, a'.\ \mathsf{Simple}\lambda b, b'.\ (a' \to a) \to (a' \to b) \to a \to b'$$

We conclude this section by showing that the same Simple type construction also works for simple recursive types, and types involving type constants for arbitrary closures $a : \mathbf{V}$.

As a corollary to definability of simple types, we would like an interpretation of typed λ-calculus (typed SK) into untyped λ-calculus with join (SKJ). However, Simple-definable types are inhabited not only by well-typed SK terms, but also SKJ -terms. We acheive the embedding theorem in 3.7 by "disambiguating" the Simple-definable closures to contain only the original SK -terms we would expect. Thus we can interpret a richly typed λ-calculus (SK) in the untyped SKJ .

Then in 4 we show how to narrow down the types to still allow fuzziness, but in a well-behaved way. By forcing fuzziness to the top of Böhm trees, we can define monadic types whose inhabitants are joins of random SK -terms. Thus we can interpret a typed λ-calculus (SK) with a monadic type of imprecise probabilities in the untyped SKRJ .

Warning: In this section only, $x :: \tau$ denotes a syntactic typing relation.

Interpreting simple types as closures

We begin with language of untyped terms m, a language of simple types τ, and rules for judging a term well-typed $m :: \tau$, as in Figure 3.1. We treat infinitary joins as basic, and finitary joins as derived: nullary $\bot := \bigsqcup \{\}$, and binary $\mathbf{J} := \lambda x, y. \bigsqcup \{x, y\}$. This way, our correctness proof extends unaltered to the join-completion of SKJ , and its subalgebras, e.g. SKJO .

Example 3.6.1. The inhabitants of the type $(a \to a) \to a \to a$ of Church numerals are (up to equality) arbitrary joins of the terms $\bot$, $\top$, zero $z = \lambda -, x.x$, and successors $s(n) = \lambda f, x.f(n\ f\ x)$ of other inhabitants n. So for example $s(s(z)) \mid s(z \mid s(\bot)) \mid s(s(s(\top))) : (a \to a) \to a \to a$ is an inhabitant that cannot be further simplified.

We interpret types as functors bivariate in each atomic type. The functors are implemented as SKJ -terms, where constants-as-closures appear as themselves, and exponentials are conjugation $a \to b = \lambda f. b \circ f \circ a$ covariant in b and contravariant in a (as in the Karoubi envelope; see [Bar84]). For example keeping track of variance, we interpret $[-]$

$$\begin{aligned}
[a] &= \mathsf{Simple}\lambda a, a'.\ a' \\
[a \to a] &= \mathsf{Simple}\lambda a, a'.\ a \to a' \\
[(a \to a) \to (a \to a)] &= \mathsf{Simple}\lambda a, a'.\ (a' \to a) \to (a \to a') \\
[a \to b] &= \mathsf{Simple}\lambda a, a'.\ \mathsf{Simple}\lambda b, b'.\ a \to b'
\end{aligned}$$

Closures (as type constants) are interpreted as themselves. Figure 3.1 formalizes this interpretation algorithm.

The interpretation algorithm (begin)s by looking at variables, traversing through the context and noting the (vari)ance of each variable, (skip)ping irrelevant variables, and ignoring the variance of type constants for closures$:\mathbf{V}$. Since the exponential reverses variance in the domain type, we build two type interpretations, one for covariant and one for contravariant uses, written as contravariant/covariant pairs.

Having interpreted the all the leaves on a type derivation tree, we next combine subtrees on each side of (exp)onentials, preserving the variance of codomain types and reversing the variance of domain types.

term formation

$$\frac{}{\mathbf{S}\ \mathsf{tm}} \qquad \frac{}{\mathbf{K}\ \mathsf{tm}} \qquad \frac{}{\top\ \mathsf{tm}} \qquad \frac{\mathsf{m}\ \mathsf{tm} \qquad \mathsf{n}\ \mathsf{tm}}{\mathsf{m}\ \mathsf{n}\ \mathsf{tm}} \qquad \frac{\forall \mathsf{m}\in\mathcal{M}.\ \mathsf{m}\ \mathsf{tm}}{\bigsqcup\mathcal{M}\ \mathsf{tm}}$$

type formation

$$\frac{\mathsf{a}\ \mathsf{var}}{\Gamma,\mathsf{a},\Gamma'\vdash \mathsf{a}\ \mathsf{tp}} \qquad \frac{\Gamma\vdash\sigma\ \mathsf{tp} \qquad \Gamma\vdash\tau\ \mathsf{tp}}{\Gamma\vdash\sigma\to\tau\ \mathsf{tp}} \qquad \frac{}{\Gamma\vdash\mathsf{any}\ \mathsf{tp}} \qquad \frac{\forall\tau\in\mathbf{A}.\ \tau\ \mathsf{tp}}{\bigcap\mathbf{A}\ \mathsf{tp}}$$

typing

$$\frac{\Gamma\vdash\rho\ \mathsf{tp} \qquad \Gamma\vdash\sigma\ \mathsf{tp} \qquad \Gamma\vdash\tau\ \mathsf{tp}}{\mathbf{S}::(\rho\to\sigma\to\tau)\to(\rho\to\sigma)\to\rho\to\tau} \qquad \frac{\Gamma\vdash\sigma\ \mathsf{tp} \qquad \Gamma\vdash\tau\ \mathsf{tp}}{\mathbf{K}::\sigma\to\tau\to\sigma} \qquad \frac{\Gamma\vdash\tau\ \mathsf{tp}}{\top::\tau} \qquad \frac{\mathsf{m}::\sigma\to\tau \qquad \mathsf{n}:\sigma}{\mathsf{m}\ \mathsf{n}::\tau}$$

$$\frac{\forall\mathsf{m}\in\mathcal{M}.\ \mathsf{m}::\tau}{\bigsqcup\mathcal{M}::\tau} \qquad \frac{\mathsf{m}\ \mathsf{tm}}{\mathsf{m}::\mathsf{any}} \qquad \frac{\forall\tau\in\mathbf{A}.\ \mathsf{m}::\tau}{\mathsf{m}::\bigcap\mathbf{A}}$$

subtyping

$$\frac{\tau\ \mathsf{tp}}{\mathsf{nil}<:\tau} \qquad \frac{\tau\ \mathsf{tp}}{\tau<:\mathsf{any}} \qquad \frac{\sigma<:\sigma' \qquad \tau<:\tau'}{\sigma\to\tau<:\sigma'\to\tau'} \qquad \frac{\bigcap\mathbf{A}\ \mathsf{tp} \qquad \tau\in\mathbf{A}}{\tau<:\bigcap\mathbf{A}} \qquad \frac{\forall\tau\in\mathbf{A}.\ \sigma<:\tau}{\sigma\ <:\ \bigcap\mathbf{A}}$$

type interpretation

$$\frac{\Gamma\vdash\mathsf{a}\ \mathsf{tp}}{\Gamma;\vdash\mathsf{a}\rhd\mathsf{a}/\mathsf{a}}\ (\mathsf{begin}) \qquad \frac{\Gamma,\mathsf{a};\Delta\vdash\mathsf{a}\rhd\mathsf{a}/\mathsf{a}}{\Gamma;\mathsf{a}/\mathsf{a}',\Delta\vdash\mathsf{a}\rhd\mathsf{a}/\mathsf{a}'}\ (\mathsf{vari}) \qquad \frac{\Gamma,\mathsf{a};\Delta\vdash\mathsf{b}\rhd\mathsf{b}/\mathsf{b}'}{\Gamma;\mathsf{a}/\mathsf{a}',\Delta\vdash\mathsf{b}\rhd\mathsf{b}/\mathsf{b}'}\ (\mathsf{skip}) \qquad \frac{}{;\Delta\vdash\mathsf{any}\rhd\mathbf{I}/\mathbf{I}}\ (\mathsf{any})$$

$$\frac{;\Delta\vdash\sigma\rhd\mathsf{m}/\mathsf{m}' \qquad ;\Delta\vdash\tau\rhd\mathsf{n}/\mathsf{n}'}{;\Delta\vdash\sigma\to\tau\rhd\mathsf{m}'\to\mathsf{n}/\mathsf{m}\to\mathsf{n}'}\ (\mathsf{exp}) \qquad \frac{;\Delta\vdash\sigma\rhd\mathsf{m}/\mathsf{m}'}{[\sigma]_\Delta=\mathsf{m}'}\ (\mathsf{root}) \qquad \frac{[\sigma]_{\Delta,\mathsf{a}/\mathsf{a}'}=\mathsf{m}}{[\sigma]_\Delta=\mathsf{Simple}\lambda\mathsf{a},\mathsf{a}'.\mathsf{m}}\ (\mathsf{bind})$$

Figure 3.1: Typing and type interpretation. Exponential subtyping is covariant in *both* domain and range. Contexts *do not* commute.

When we reach the (root) of the typing derivation tree, we (bind) each contravariant/covariant pair of type variables with a type constructor Simple, implemented as an SKJ term. Simple inputs a type code $\mathsf{f}=\lambda\mathsf{a},\mathsf{a}'.\mathsf{t}$ bivariate in a type variable, and produces a closure

$$\mathsf{Simple}\ \ \mathsf{f}=\mathbf{V}\ \ (\mathsf{f}\ \ \mathsf{s}_1\ \ \mathsf{r}_1\ \ |\ \ \mathsf{f}\ \ \mathsf{s}_2\ \ \mathsf{r}_2\ \ |\ \ \ldots)$$

by substituting for the contra- and co-variant occurrences of the variable various section-retract pairs s, r satisfying $r \circ s = \mathbf{I}$ and $s \circ r \not\sqsubseteq \mathbf{I}$. We can generate all the section-retract pairs from just two pairs

```
raise  :=  (λx, −. x).
lower  :=  (λx. x ⊤).
!check  raise = K    AND    lower = ⟨⊤⟩.
!check  lower∘raise = I  ⋣  raise∘lower.
```

```
pull  :=  (λx, y. x | div y).
push  :=  (λx. x ⊥).
!check  pull = K | K div    AND    push = ⟨⊥⟩.
!check  push∘pull = I  ⋣  pull∘push.
```

and using the facts that if s, r and s', r' are section-retract pairs, then so also are their composition $s \circ s', r' \circ r$ and their conjugation $r \to s', s \to r'$. Joining over all compositions (via $\mathbf{V}(\dots)$ and $f\ \mathbf{I}\ \mathbf{I}$) and conjugations (via $s\lambda a, a'.\ s\lambda b, b'.\ f\ (a' \to b)\ (a \to b')$) gives the type interpreter for a single atomic type

```
!define Simple  :=  (any→V) (
    Yλs, f. f I I
            | f raise lower
            | f pull push
            | sλa, a'. sλb, b'. f (a'→b) (a→b')
).
```

It will also be convenient to write Simple f as an infinitary join over a set $\mathbb{S}$ of section-retract pairs, which is by definition the smallest set such that

$$\frac{}{\langle \mathbf{I}, \mathbf{I}\rangle \in \mathbb{S}} \qquad \frac{}{\langle \mathsf{raise}, \mathsf{lower}\rangle \in \mathbb{S}} \qquad \frac{}{\langle \mathsf{pull}, \mathsf{push}\rangle \in \mathbb{S}} \qquad \frac{\langle a, a'\rangle \in \mathbb{S} \qquad \langle b, b'\rangle \in \mathbb{S}}{\langle a' \to b, a \to b'\rangle \in \mathbb{S}}$$

We can then write

$$\mathsf{Simple} = \lambda f.\ \mathbf{V}\ (\bigsqcup_{\langle s,r\rangle \in \mathbb{S}} f\ s\ r)$$

Now we can show that it doesn't matter how we order the variables in the noncommutative context Γ of a type derivation.

Lemma 3.6.2. *(commutativity)* $(\mathsf{Simple}\lambda a, a'.\ \mathsf{Simple}\lambda b, b'.\ M) = (\mathsf{Simple}\lambda b, b'.\ \mathsf{Simple}\lambda a, a'.\ M)$.

Proof. By associativity and commutativity of join. □

Lemma 3.6.3. *(weakening) If* b, b' *are not free in* M*, then* $(\mathsf{Simple}\lambda a, a'.\ \mathsf{Simple}\lambda b, b'.\ M) = (\mathsf{Simple}\lambda a, a'.\ M)$.

Proof. Generalizing the property

```
!check  V(V x | V y) = V(x | y).
```

to infinitary joins $\mathbf{V}\ \bigsqcup \mathbf{V} = \mathbf{V}\ \bigsqcup$, we show

$$\begin{aligned}
\mathsf{Simple}\lambda a, a'.\ \mathsf{Simple}\lambda b, b'.\ M &= \mathsf{Simple}\lambda a, a'.\ \mathbf{V}\ M \\
&= \mathbf{V}\ (\bigsqcup_{\langle a,a'\rangle \in \mathbb{S}} \mathbf{V}\ M) \\
&= \mathbf{V}\ (\bigsqcup_{\langle a,a'\rangle \in \mathbb{S}} M) \\
&= \mathsf{Simple}\lambda a, a'.\ M
\end{aligned}$$

□

We can also show that interpretation respects exponentials. First a

Lemma 3.6.4. *(conjugation)* $(\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \mathsf{Simple}\lambda \mathsf{b}, \mathsf{b}'.\ \mathsf{f\ a\ a}' \to \mathsf{g\ b\ b}') \sqsubseteq \mathsf{Simple\ f} \to \mathsf{Simple\ g}$.

Proof. Using infinitary versions of the following properties of conjugation

$$\begin{array}{|l}
!\mathsf{check}\ \ \mathsf{x} \to (\mathsf{y} \mid \mathsf{z}) = \mathsf{x} \to \mathsf{y} \mid \mathsf{x} \to \mathsf{z}. \\
!\mathsf{check}\ \ (\mathsf{x} \mid \mathsf{y}) \to \mathsf{z} \ \not\sqsupseteq\ \mathsf{x} \to \mathsf{z} \mid \mathsf{y} \to \mathsf{z}. \\
!\mathsf{check}\ \ \mathbf{V}(\mathsf{a} \to \mathsf{b}) \ \not\sqsubseteq\ \mathbf{V}\ \mathsf{a} \to \mathbf{V}\ \mathsf{b}.
\end{array}$$

we show

$$\begin{aligned}
&\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \mathsf{Simple}\lambda \mathsf{b}, \mathsf{b}'.\ \mathsf{f\ a\ a}' \to \mathsf{g\ b\ b}' \\
&\quad = \mathbf{V} \bigsqcup_{\langle \mathsf{a},\mathsf{a}'\rangle \in \mathbb{S}} \mathbf{V} \bigsqcup_{\langle \mathsf{b},\mathsf{b}'\rangle \in \mathbb{S}} \mathsf{f\ a\ a}' \to \mathsf{g\ b\ b}' \\
&\quad = \mathbf{V} \bigsqcup_{\langle \mathsf{a},\mathsf{a}'\rangle \in \mathbb{S}} \bigsqcup_{\langle \mathsf{b},\mathsf{b}'\rangle \in \mathbb{S}} \mathsf{f\ a\ a}' \to \mathsf{g\ b\ b}' \\
&\quad = \mathbf{V} \bigsqcup_{\langle \mathsf{a},\mathsf{a}'\rangle \in \mathbb{S}} (\mathsf{f\ a\ a}' \to \bigsqcup_{\langle \mathsf{b},\mathsf{b}'\rangle \in \mathbb{S}} \mathsf{g\ b\ b}') \\
&\quad \sqsubseteq \mathbf{V}\ (\ (\bigsqcup_{\langle \mathsf{a},\mathsf{a}'\rangle \in \mathbb{S}} \mathsf{f\ a\ a}') \to (\bigsqcup_{\langle \mathsf{b},\mathsf{b}'\rangle \in \mathbb{S}} \mathsf{g\ b\ b}')\ ) \\
&\quad \sqsubseteq (\mathbf{V} \bigsqcup_{\langle \mathsf{a},\mathsf{a}'\rangle \in \mathbb{S}} \mathsf{f\ a\ a}') \to (\mathbf{V} \bigsqcup_{\langle \mathsf{b},\mathsf{b}'\rangle \in \mathbb{S}} \mathsf{g\ b\ b}') \\
&\quad = \mathsf{Simple\ f} \to \mathsf{Simple\ g}
\end{aligned}$$

□

Question 3.6.5. *Does the converse hold?*

Now to express respect for exponentials, we need a notion of equality at a type.

Definition 3.6.6. Let x, y be arbitrary terms, and $\mathsf{a} : \mathbf{V}$ be a closure. We say $\mathsf{x} = \mathsf{y} \mod \mathsf{a}$ iff $\mathsf{a\ x} = \mathsf{a\ y}$, i.e. iff x and y are identical when seen as terms of type a.

Corollary 3.6.7. *($[-]$ respects exponentials) For σ, τ simple types with disjoint sets of variables variables, $[\sigma \to \tau] \sqsubseteq [\sigma] \to [\tau]$, and hence if $\mathsf{x} = \mathsf{y} \mod [\sigma \to \tau]$ then $\mathsf{x} = \mathsf{y} \mod [\sigma] \to [\tau]$.*

Consider the action of Simple on a few functions

Lemma 3.6.8.

$$\begin{array}{|l}
!\mathsf{check}\ \ (\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \mathbf{I}) = \mathbf{I}. \\
!\mathsf{check}\ \ (\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \bot) = \mathbf{I}. \\
!\mathsf{check}\ \ (\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \top) = \top. \\
!\mathsf{check}\ \ (\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \mathsf{a}) = \top. \\
!\mathsf{check}\ \ (\mathsf{Simple}\lambda \mathsf{a}, \mathsf{a}'.\ \mathsf{a}') = \mathsf{div}. \qquad \textit{inhabited by } \{\bot, \top\}
\end{array}$$

Proof. Left as exercise for Johann. □

Correctness of the simple type constructor

We need to show soundness $\mathsf{q} :: \tau \implies \mathsf{q} : [\tau]$ and completeness $\mathsf{q} : [\tau] \implies \mathsf{q} :: \tau$. Soundness of our Simple type interpretation is provable by an easy induction on derivations.

Theorem 3.6.9. *(soundness) If $\mathsf{q} :: \tau$ then $\mathsf{q} : [\tau]$.*

Proof. We consider the action of each generator of Simple on each term. So letting a/a', b/b', and c/c' be section/retract pairs so that $\mathsf{a}' \circ \mathsf{a} = \mathsf{b}' \circ \mathsf{b} = \mathsf{c}' \circ \mathsf{c} = \mathbf{I}$, we show that each term is fixed under the generators of simple. By induction on typing derivations (ignoring for now intersection types),

Case: $\mathbf{S} :: (\rho \to \sigma \to \tau) \to (\rho \to \sigma) \to \rho \to \tau$.

$$\begin{aligned}
&((\mathsf{a}' \to \mathsf{b}' \to \mathsf{c}) \to (\mathsf{a}' \to \mathsf{b}) \to \mathsf{a} \to \mathsf{c}')\ \mathbf{S} \\
&\quad = (\mathsf{a}' \to \mathsf{b}' \to \mathsf{c}) \to (\mathsf{a}' \to \mathsf{b}) \to \mathsf{a} \to \mathsf{c}'\ \ \lambda x, y, z.\ x\ z(y\ z) \\
&\quad = \lambda x, y, z.\ \mathsf{c}'.\ (\mathsf{a}' \to \mathsf{b}' \to \mathsf{c}\ x)\ (\mathsf{a}\ z)\ ((\mathsf{a}' \to \mathsf{b}\ y)\ \mathsf{a}\ z) \\
&\quad = \lambda x, y, z.\ \mathsf{c}' \circ \mathsf{c}.\ x\ ('\mathsf{a} \circ \mathsf{a}\ z)\ (\mathsf{b}' \circ \mathsf{b}.\ y.\ \mathsf{a}' \circ \mathsf{a}\ z) \\
&\quad = \lambda x, y, z.\ x\ z(y\ z) \\
&\quad = \mathbf{S}
\end{aligned}$$

Case: $\mathbf{K} :: \sigma \to \tau \to \sigma$

$$\begin{aligned}
(\mathsf{a} \to \mathsf{b} \to \mathsf{a}')\ \mathbf{K} &= (\mathsf{a} \to \mathsf{b} \to \mathsf{a}')\ \lambda x, y.\ x \\
&= \lambda x, y.\ \mathsf{a}'.\ \mathsf{a}\ x \\
&= \lambda x, y.\ \mathsf{a} \circ \mathsf{a}'\ x \\
&= \lambda x, y.\ x \\
&= \mathbf{K}
\end{aligned}$$

Case: $\top :: \tau$ by induction on the structure of the type τ:

Subcase: For $\tau = \mathsf{a}$ a variable, consider the atomic retracts $\mathsf{a}'\ \ \top$

$$\begin{aligned}
&\mathsf{lower}\ \ \top = \top\ \ \top = \top \\
&\mathsf{push}\ \ \top = \lambda y.\top \mid \mathsf{div}\ \ y = \lambda y.\top = \top
\end{aligned}$$

Subcase: For $\tau = \rho \to \sigma$ for types ρ, σ, consider the action of any retract b' and any section a.

$$\begin{aligned}
(\mathsf{a} \to \mathsf{b}')\ \top &= \lambda x.\ \mathsf{b}'\ (\top\ (\mathsf{a}\ x)) \\
&= \lambda x.\ \mathsf{b}'\ \top \\
&= \lambda x.\ \top && \textit{by hypothesis on } \mathsf{b}' \\
&= \top
\end{aligned}$$

Subcase: For $\tau = \mathsf{any}\ \mathbf{I}\ \ \top = \top$.

Case: $\mathsf{m}\ \ \mathsf{n} :: \tau$, assuming $\mathsf{m} :: \sigma \to \tau$ and $\mathsf{n} :: \sigma$. By hypothesis, $\mathsf{m} : [\sigma \to \tau]$ and $\mathsf{n} : [\sigma]$, so

$$\begin{aligned}
\mathsf{m}\ \ \mathsf{n} &\sqsubseteq [\tau]\ \ (\mathsf{m}\ \ \mathsf{n}) && \textit{since } [\tau] \textit{ is a closure} \\
&= (\mathsf{any} \to [\tau])\ \ \mathsf{m}\ \ \mathsf{n} \\
&\sqsubseteq ([\sigma] \to [\tau])\ \ \mathsf{m}\ \ \mathsf{n} && \textit{since } [\sigma] \textit{ is a closure} \\
&\sqsubseteq [\sigma \to \tau]\ \ \mathsf{m}\ \ \mathsf{n} && \textit{by respect for exponentials} \\
&= \mathsf{m}\ \ \mathsf{n} && \textit{by hypothesis on } \mathsf{m}
\end{aligned}$$

Case: $\bigsqcup \mathcal{M} :: \tau$. Observe that each operation of Simple acts pointwise on joins (say by induction over type structure), i.e. $\bigsqcup$and Simple commute. Hence,

$$\begin{aligned}
[\tau]\ \ \bigsqcup \mathcal{M} &= \bigsqcup_{\mathsf{m} \in \mathcal{M}} [\tau]\ \ \mathsf{m} && \textit{by commutativity} \\
&= \bigsqcup_{\mathsf{m} \in \mathcal{M}} \mathsf{m} && \textit{by hypothesis} \\
&= \bigsqcup \mathcal{M}
\end{aligned}$$

Case: $\mathsf{m} :: \mathsf{any}$. Since $[\mathsf{any}] = \mathbf{I}$, anything is fixed by $[\mathsf{any}]$.

$\square$

Completeness is much more difficult, and makes crucial use of the Böhm Tree theorem of 3.1. In particular, Simple does not work in full D_∞ models, where the BT theorem fails. The proof than any fixedpoint $\mathsf{q} : [\tau]$ is typable $\mathsf{q} :: \tau$ considers by way of contradiction any $\mathsf{q}! :: \tau$ not typable by τ, showing that $[\tau]$ raises q to something strictly larger. The idea is to

(1) consider WLOG any badly-typed finite SK -definable BT $\mathsf{q}' \sqsupseteq \mathsf{q}$ extending q, so also $\mathsf{q}'! :: \tau$.

(2) *adapt* the term q' to the type τ by specializing the type (via moves $\mathsf{a} :> \mathsf{b} \to \mathsf{c}$) to the term, and putting the term in β-long form down to the point of typing error.
(3) consider the BT node where the typing error occurs, a head normal form; the error is one of: too few arguments, too many arguments, or an atomic type clash.
(4) show that various sequences of $\mathsf{section} \to \mathsf{retract}$ operations raise each error to $\top$.
(5) bubble-up the error to the top of q' BT and generalize the type (via moves $\mathsf{b} \to \mathsf{c} <: \mathsf{a}$) back to its original form.

So let us consider a badly-typed finite sequential BT $\mathsf{q}'! :: \tau$. There are two problems we face in finding a type error: ill-typed terms might look like terms of lower type, or they might look like terms of higher type. To deal with higher-type-looking terms, we adapt their types by specializing moves $\mathsf{a} :> \mathsf{b} \to \mathsf{c}$. For example when checking

$$\lambda \mathsf{f}, \mathsf{x}.\mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{f})\ ?:\ \mathsf{a} \to \mathsf{a}$$

we specialize $\mathsf{a} \to \mathsf{a}\ \ :>\ \ (\mathsf{b} \to \mathsf{c}) \to \mathsf{b} \to \mathsf{c}\ \ :>\ \ ((\mathsf{d} \to \mathsf{e}) \to \mathsf{c}) \to (\mathsf{d} \to \mathsf{e}) \to \mathsf{c}$ where we can annotate

$$\lambda \mathsf{f} : (\mathsf{d} \to \mathsf{e}) \to \mathsf{c}, \mathsf{x} : \mathsf{d} \to \mathsf{c}.\ \mathsf{f}\ (\lambda \mathsf{y} : \mathsf{d}.\ \mathsf{x}\ \mathsf{f}\ : \mathsf{e})\ : \mathsf{c}$$

and observe a problem typing $\mathsf{f} : (\mathsf{d} \to \mathsf{e}) \to \mathsf{c}, \mathsf{x} : \mathsf{d} \to \mathsf{c} \vdash \mathsf{x}\ \mathsf{f} : \mathsf{e}$. Specialization is built into the Simple type constructor via conjugation

$$\mathsf{Simple}\ \mathsf{f} = \mathbf{V}\ (\ \ldots\ \mid\ \mathsf{Simple} \lambda \mathsf{a}, \mathsf{a}'.\ \mathsf{Simple} \lambda \mathsf{b}, \mathsf{b}'.\ \mathsf{f}\ (\mathsf{a}' \to \mathsf{b})\ (\mathsf{a} \to \mathsf{b}'))$$

Lemma 3.6.10. *(specialization) Let σ, τ be pure simple types, i.e., free of type constants and intersection operations. If $\sigma <: \tau$ then $[\sigma] \sqsubseteq [\tau]$.*

Proof. Suppose $\sigma <: \tau$. Then there is a sequence of types $\sigma_0 <: \sigma_1 <: \cdots <: \sigma_\mathsf{n}$ such that each $\sigma_\mathsf{i} <: \sigma_{\mathsf{i}+1}$ is the result of an elementary specializing substutition $\mathsf{a_i} <: \mathsf{b_i} \to \mathsf{c_i}$ of type variables (pf: by induction on structure of subtyping derivations). At each substitution, the conjugation operation in Simple shows that $\sigma_\mathsf{i} \sqsupseteq \sigma_{\mathsf{i}+1}$. □

To deal with lower-type looking terms, we next adapt the term by putting it in long β-η form ([Hin97]) with respect to the adapted type. For example when checking

$$\lambda \mathsf{x}, \mathsf{y}.\mathsf{x}\ ?:\ \mathsf{a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{a}$$

we lengthen the term by η-expanding and annotate to

$$\lambda \mathsf{x} : \mathsf{a}, \mathsf{y} : \mathsf{a}, \mathsf{z} : \mathsf{a}.\ \mathsf{x}\ \mathsf{z}\ : \mathsf{a}$$

to find a problem typing $\mathsf{x} : \mathsf{a}, \mathsf{z} : \mathsf{a} \vdash \mathsf{x}\ \mathsf{z} : \mathsf{a}$.

After specializing a term, there are three possibile kinds of typing problem: variable-arrow, arrow-variable, and variable-variable type clashes. When we find such a typing problem we create an error by raising the conflicting subterm to $\top$, using one of the following $\mathsf{section} \to \mathsf{retract}$ operations. Below we formally verify unary forms of each lemma, and informally prove the general n-ary forms.

Lemma 3.6.11. *(one too few args)* $\top = [\mathsf{a} \to \mathsf{a}]\ \lambda \mathsf{f}, -.\mathsf{f}$.

Proof. $(\mathsf{pull} \to \mathsf{push}); (\mathsf{raise} \to \mathsf{lower})$ raises $\lambda \mathsf{f}, -.\mathsf{f}$ as

$$\begin{aligned}
&!\mathsf{check}\ (\mathsf{pull} \to \mathsf{push})\ (\lambda \mathsf{f}, -.\mathsf{f})\\
&\qquad = (\lambda \mathsf{f}.\ (\lambda -, \mathsf{x}.\mathsf{f} \mid \mathsf{div}\ \mathsf{x})\ \bot)\\
&\qquad = (\lambda \mathsf{f}, \mathsf{x}.\mathsf{f} \mid \mathsf{div}\ \mathsf{x}).\\
&!\mathsf{check}\ (\mathsf{raise} \to \mathsf{lower})\ (\lambda \mathsf{f}, \mathsf{x}.\mathsf{f} \mid \mathsf{div}\ \mathsf{x})\\
&\qquad = (\lambda \mathsf{f}.\top)\\
&\qquad = \top.
\end{aligned}$$

□

Lemma 3.6.12. *(too few args)* $\top = [a \to a]\ \lambda f, -^{\sim n}.f$.

Proof. $(\text{pull} \to \text{push});\ n(\text{raise} \to \text{lower})$ raises

$$\begin{aligned}
\lambda f, -^{\sim n}.f &\mapsto \lambda f.\ (\lambda -^{\sim n}, x.\ f \mid \text{div}\ x)\ \bot && \textit{via}\ \text{pull} \to \text{push} \\
&= \lambda f, -^{\sim n-1}, x.\ f \mid \text{div}\ x \\
&\mapsto \lambda f, x.(\lambda -^{\sim n-1}.f) \mid \text{div}\ x && \textit{via}\ (n-1)(\text{raise} \to \text{lower}) \\
&\mapsto \lambda f.(\lambda -^{\sim n}.f) \mid \text{div}\ \top && \textit{via}\ \text{raise} \to \text{lower} \\
&= \lambda f.\text{div}\ \top \\
&= \top
\end{aligned}$$

□

Lemma 3.6.13. *(one too many args)* $\top = [a \to a]\ \lambda f.f\ \bot$.

Proof. $(\text{raise} \to \text{lower}); (\text{pull} \to \text{push})$ raises $\lambda f.f\ \bot$ as

$$\begin{aligned}
&!\text{check}\ (\text{raise} \to \text{lower})\ (\lambda f.f\ \bot) \\
&\qquad = (\lambda f.\ (\lambda - .f)\ \bot\ \top) \\
&\qquad = (\lambda f.f\ \top). \\
&!\text{check}\ (\text{pull} \to \text{push})\ (\lambda f.f\ \top) \\
&\qquad = (\lambda f.\ (\lambda x.f \mid \text{div}\ x)\ \top) \\
&\qquad = (\lambda f.\ f \mid \text{div}\ \top) \\
&\qquad = \top.
\end{aligned}$$

□

Lemma 3.6.14. *(too many args)* $\top = [a \to a]\ \lambda f.f\ \bot^{\sim n}$.

Proof. $n(\text{raise} \to \text{lower}); (\text{pull} \to \text{push})$ raises

$$\begin{aligned}
\lambda f.f\ \bot^{\sim n} &\mapsto \lambda f.\ (\lambda -^{\sim n}.\ f)\ \bot^{\sim n}\ \top^{\sim n} && \textit{via}\ n(\text{raise} \to \text{lower}) \\
&= \lambda f.\ f\ \top^{\sim n} \\
&\mapsto \lambda f.\ (\lambda x.f \mid \text{div}\ x)\ \top^{\sim n} && \textit{via}\ \text{pull} \to \text{push} \\
&= \lambda f.\ (f \mid \text{div}\ \top)\ \top^{\sim n-1} \\
&= \top\ \top^{\sim n-1} \\
&= \top
\end{aligned}$$

□

Lemma 3.6.15. *(clashing) If* a, b *are distinct type variables then in the interpreted type,*

$$\top = \text{Simple}\lambda a, a'.\ \text{Simple}\lambda b, b'.\ b' \circ a$$

Proof. We can simply mismatch two section-retract pairs

$$\begin{aligned}
&!\text{check}\ (\text{Simple}\lambda a, a'.\ \text{Simple}\lambda b, b'.\ b' \circ a) \\
&\qquad \sqsupseteq \text{lower} \circ \text{pull} \\
&\qquad = \langle \top \rangle \circ (\lambda x, y.\text{div}\ y \mid x) \\
&\qquad = (\lambda y.\text{div}\ y \mid \top) \\
&\qquad = \top.
\end{aligned}$$

□

Now after finding a type error at a specialized type, we want to be able to "bubble up" the error to the most general type.

Lemma 3.6.16. *(close bubbling)* $\top = [a \to a]\ \lambda f, -.f\ \top$.

Proof. $\mathsf{pull}\to\mathsf{push}$ raises $\lambda f,-.f\ \top$ as

$$\begin{aligned}
&!\mathsf{check}\ (\mathsf{pull}\to\mathsf{push})\ (\lambda f,-.f\ \top)\\
&\quad = (\lambda f.\ (\lambda -.\ (\lambda x.f \mid \mathsf{div}\ x)\ \top)\ \bot)\\
&\quad = (\lambda f.\ (\lambda x.\ f \mid \mathsf{div}\ x)\ \top)\\
&\quad = (\lambda f.\ f \mid \mathsf{div}\ \top)\\
&\quad = \top.
\end{aligned}$$

□

Lemma 3.6.17. *(bubbling)* $\top = [a\to a]\ \lambda f,-^{\sim n},-.f\ \bot^{\sim n}\ \top$.

Proof. $n(\mathsf{raise}\to\mathsf{lower});(\mathsf{pull}\to\mathsf{push})$ raises

$$\begin{array}{lll}
\lambda f,-^{\sim n},-.f\ \bot^{\sim n}\ \top & \mapsto\ \lambda f,-.\ f\ \top & \textit{via}\ n(\mathsf{raise}\to\mathsf{lower})\\
& \mapsto\ \top & \textit{via}\ \mathsf{pull}\to\mathsf{push}
\end{array}$$

□

Example 3.6.18. We can show that $\top = [a\to a]\ \lambda f,x.\ f\ f$ by

- first specializing to $a\to a\ :>\ (b\to c)\to b\to c$;
- then finding an error with the too-few-args lemma, raising $[(b\to c)\to b\to c]\ \lambda f,x.f\ f\ \sqsupseteq\ \lambda f,x.f\ \top$;
- and finally bubbling the error up to the more general type, raising $[a\to a]\ \lambda f,x.f\ \top = \top$.

Next we need to show that any kind of typing error can be detected with our tools for $a\to a$. For more complex types, we might have typing errors deep in a Böhm tree. We can use $a\to a$ arguments to detect any single type error in a Böhm tree, since we just need top-down and bottom-up types to agree. But we also need to be able to simultaneously raise errors in different branches of a Böhm Tree.

Example 3.6.19. Consider the type $((a\to a)\to(a\to a)\to b)\to b$ of pairs of semibooleans, and the non-inhabitant $\lambda f.\ f\ (\lambda x,-.\ x)\ (\lambda x.\ x\ \bot)$, where the first component has too few arguments, and the second too many. The too-few-args lemma raises the first component, and the too-many-args raises the second comonent, but can we raise both?

We gerneralize this with a sort of simultaneous raising lemma.

Lemma 3.6.20. *(coherence): If* Simple *raises each of a set of errors in a Böhm tree, then it simultaneously raises them all.*

Proof. By compactness we need only argue about finite sets of errors. Suppose thus we are given a finite number of arguments, each detecting a particular type error in a Böhm tree. It is sufficient to weave them together into a single argument detecting all errors.

Note that each sequence of operations in the detection of the errors is one of: (too-few), (too-many), (clashing), and (bubbling). These are composed of elementary operations $\mathsf{raise}\to\mathsf{lower}$, $\mathsf{pull}\to\mathsf{push}$, and their specializations $a :> b\to c$.

In any of these cases, we can safely apply the specialized cases first, then the $\mathsf{pull}\to\mathsf{push}$ moves, then the $\mathsf{raise}\to\mathsf{lower}$ cases. Hence there is a sequence of moves containing all each of the given raising sequences as a subsequence. □

Finally we start back at step (1) and assemble the type checking lemmas.

Theorem 3.6.21. *(completeness): If* $q:[\tau]$ *then* $q::\tau$.

Proof. Suppose q extends a finite Böhm tree with typing errors, i.e. for various subterms the top-down and bottom-up typings disagree, or some subterms cannot even be typed. Each of the typing errors is raised to $\top$ by one of the (too-few), (too-many) or (clashing) lemmas. The coherence lemma allows us to raise all the errors simultaneously. The resulting raised q then has proper type. □

Corollary 3.6.22. (Simple *works)* $q::\tau$ *iff* $q:[\tau]$.

Proof. The soundness and completeness theorems each prove one direction. □

Extending to recursive types, intersection types, and type constants

Using intersection types we can construct infinite (e.g. recursive) types by intersecting over infinite chains of approximations. Then using the subtyping partial order, we can prove equations among recursive types.

Example 3.6.23. We can construct a type μ a. b$\to$(a$\to$b)$\to$b of ambiguous numerals by intersecting all the types

$$\begin{array}{rcl} & \mathsf{any} & \\ \mathsf{b}\to(& \mathsf{any} & \to\mathsf{b})\to\mathsf{b} \\ \mathsf{b}\to((\mathsf{b}\to(& \mathsf{any} & \to\mathsf{b})\to\mathsf{b})\to\mathsf{b})\to\mathsf{b} \\ & \dots & \end{array}$$

This can be defined as a least fixed point, either outside or inside the type iterpretation

```
!check (
  pre_num  :=  (Y λa. Simpleλb, b'. b→(a→b)→b').
  pre_num = (Simpleλb, b'. V  (λa. b→(a→b)→b')  any)
).
```

We can also extend our simple type interpretation by arbitrary closures without much work. Consider extending the type language and interpretation by

extended types

$$\frac{\mathsf{m}\ \ \mathsf{tm} \qquad \mathsf{m}:\mathbf{V}}{\Gamma\vdash\mathsf{m}\ \ \mathsf{tp}} \qquad\qquad \frac{\mathsf{m}\ \ \mathsf{tm} \qquad \mathsf{m}:\mathbf{V}}{\mid\Gamma\vdash\mathsf{m}\rhd\mathsf{m}/\mathsf{m}}\ (\mathsf{close})$$

Lemma 3.6.24. *For ant* a, b : $\mathbf{V}$ *closures,* a$\to$b $=$ $\mathbf{P}$ (a$\to$any) (any$\to$b).

Proof. Simple reduction. ☐

Lemma 3.6.25. *Let τ be a simple type with both type variables and type constants (closures) as leaves, and let ρ be τ with all* variables *replaced by* any *and σ be τ with all* constants *replaced by* any*, so that $[\sigma]=\sigma$. Then $[\tau]=\mathbf{P}[\rho][\sigma]=\mathbf{P}[\rho]\sigma$.*

Example 3.6.26. (semisets) Semisets of a : $\mathbf{V}$ are joins of terms λf : a$\to$any. f x. We will discuss them more at the end of 3.7

```
sset  :=  V (Simpleλb, b'. (any→b)→b').          =  [(any→b)→b]
Sset  :=  V→V (λa. Simpleλb, b'. (a→b)→b').      =  λa : V. [(a→b)→b]
!check Sset a = P sset ((V a→any) → any).
```

Proof. By distributivity of right-application over join: each term in the simple interpretation $\mathbb{S}_{\mathsf{a},\mathsf{a}'}$ ... of τ is a conjugation of terms in ρ and closures in σ. For arbitrary terms x and closures c,

$$\mathbf{V}\ (\mathsf{x}\to\mathsf{c}) = \mathbf{P}\ (\mathsf{x}\to\mathbf{I})\ (\mathbf{I}\to\mathsf{c})$$
$$\mathbf{V}\ (\mathsf{c}\to\mathsf{x}) = \mathbf{P}\ (\mathsf{c}\to\mathbf{I})\ (\mathbf{I}\to\mathsf{x})$$

☐

3.7 Types as closures

In this section we define a variety of types and type constructors, whose inhabitants are exactly the partial SK -terms we would expect –ambiguous SKJ inhabitants are eliminated. We construct these types from Simple-definable types by raising ambiguous terms to $\top$, or "disambiguating".

However, this eschewing of ambiguous terms comes at the cost of a clean categorical characterization of our type constructors: our products and sums are not categorical products and sums. The situation becomes clearer later when we add randomness in 4 and our present disambiguation tricks fail. Forced to work with ambiguous terms, we will develop a monadic type system where type constructors satisfy categorical properties, and simple types are likely definable (this is future work). For now we accept the loss of nice categorical properties.

Atomic types for truth-values

We first construct four atomic types div, semi, unit, bool that will be used for various logics in 3.9.

We can now give a Simple definition of the type of divergent computations, introduced in section 3.8. This two-point closure will be used as the space of truth values for *probing logic*.

| !check div $= \mathbf{V}\ \mathbf{K} = \mathbf{V}\ \langle\top\rangle = \mathbf{V}$ raise $= \mathbf{V}$ lower.
| !check div $=$ (Simpleλa, a$'$. a$'$).

Theorem 3.7.1. $\mathrm{inhab}(\mathsf{div}) = \{\bot, \top\}$.

Proof. We already know that

| !check $\bot$: div.

Any other q : div converges and hence is sent to $\top$ by sufficiently many arguments of $\top$. So q $= \top$. □

Semibooleans, or Sierpinski space will be used as the space of truth values for *testing logic*.

| !define semi := (Simpleλa, a$'$. a $\to$ a$'$).

| !check ($\mathbf{I}, \bot$: $\mathbf{I} \to \mathbf{I}$).
| !check ($\mathbf{I}, \bot$: raise $\to$ lower).
| !check ($\mathbf{I}, \bot$: pull $\to$ push).
| !check ($\mathbf{I}, \bot$: semi).

| !check div <: semi.

Theorem 3.7.2. $\mathrm{inhab}(\mathsf{semi}) = \{\bot, \mathbf{I}, \top\}$.

Proof. By the Simple theorem, since these are exactly the inhabitants of the simple type a $\to$ a, they are the fixedpoints of the type's interpretation Simpleλa, a$'$. a $\to$ a$'$. □

The canonical unit type will have inhabitants $\{\mathbf{I}, \top\}$, with $\mathbf{I}$ the unique consistent inhabitant. This space will be used as the space of truth values for *checking logic*.

| !define unit := $\mathbf{P}$ semi(Above $\mathbf{I}$).

| !check unit $= \mathbf{P}$ semi($\mathbf{K}\ \mathbf{I}$).
| !check $\mathbf{I}$: unit.
| !check unit $\bot = \mathbf{I}$ **AND** $\bot$!: unit.
| !check unit <: semi.

Theorem 3.7.3. $\mathrm{inhab}(\mathsf{unit}) = \{\mathbf{I}, \top\}$.

Proof. $\mathrm{inhab}(\mathsf{unit}) \subseteq \mathrm{inhab}(\mathsf{semi})$; individual containment was just checked. □

Remark. We now know the entire semilattice of subtypes of semi:

$$\begin{tikzcd}[column sep=small]
 & \{\bot,\mathbf{I},\top\} \arrow[dl, no head] \arrow[dr, no head] & & & & \mathsf{semi} \arrow[dl, no head] \arrow[dr, no head] & \\
\{\bot,\top\} \arrow[dr, no head] & & \{\mathbf{I},\top\} \arrow[dl, no head] & : & \mathsf{div} \arrow[dr, no head] & & \mathsf{unit} \arrow[dl, no head] \\
 & \{\top\} & & & & \mathsf{nil} &
\end{tikzcd}$$

Finally we need boolean values for *predicate logic*. The boolean type is like a widened semi, inhabited by $\{\bot, \mathbf{K}, \mathbf{F}, \top\}$. Such a bool is not Simple-definable, but we can Simple-define a wider type boool inhabited by $\{\bot, \mathbf{K}, \mathbf{F}, \mathbf{J}, \top\}$

```
boool := (Simple λa, a′. a→a→a′).
```

```
!check ( ⊥, K, F, J : I→I→I ).
!check ( ⊥, K, F, J : raise→raise→lower ).
!check ( ⊥, K, F, J : pull→pull→push ).
!check ( ⊥, K, F, J : boool ).
```

```
!check (∀x:boool. x I I : semi).
```

and then manually raise $\mathbf{J}$ to $\top$.

```
!define bool := P boool (λq. q ⊥ (q ⊤ ⊥)).
```

```
!check ( ⊥, K, F : (λq. q ⊥ (q ⊤ ⊥)) ).
!check ( ⊥, K, F : bool ).
!check ⊤ = (λq. q ⊥ (q ⊤ ⊥)) J = bool J.
```

We will often employ join-eschewing terms like $(\lambda \mathsf{q}.\ \mathsf{q}\ \bot\ (\mathsf{q}\ \top\ \bot)$. Because we regard join as ambiguity, we call these *disambiguating* terms.

Theorem 3.7.4. $\mathrm{inhab}(\mathsf{bool}) = \{\bot, \mathbf{K}, \mathbf{F}, \top\}$.

Proof. By the simple theorem, we may restrict attention to the inhabitants of $\mathsf{a}\to\mathsf{a}\to\mathsf{a}$, namely $\bot$, $\top$, $\mathbf{K} = \lambda\mathsf{x}, -.\mathsf{x}$, $\mathbf{F} = \lambda -, \mathsf{y}.\mathsf{y}$, and $\mathbf{J} = \mathbf{K} \mid \mathbf{F}$. All of $\bot, \top, \mathbf{K}, \mathbf{F}$ are fixed by the term
$\mathsf{disambiguate} := \lambda\mathsf{q}.\mathsf{q}\ \bot\ (\mathsf{q}\ \top\ \bot)$, and hence inhabit bool, but disambiguate $\mathbf{J} = \top$. Hence $\bot, \top, \mathbf{K}, \mathbf{F}$ are the only inhabitants. □

Polymorphic type constructors

Various exponential types

```
Exp := (V→V→V) (λa, b. a→b).        or written a→b
Endo := (V→V) (λa. a→a).            endomorphisms
Bin_op := (V→V) (λa. a→a→a).        binary operations
From := (V→V) (λa. a→any).
To := (V→V) (λa. any→a).
```

```
!check Endo a = Exp a a.
!check Bin_op a = Exp a (Exp a a).
!check From a = Exp a ⊥.
!check To a = Exp ⊥ a.
!check Exp a b = P (From a) (To b).
```

Recall that $x = y \mod a$ iff $a\ x = a\ y$.

Theorem 3.7.5. *Let* $a, b : \mathbf{V}$. *Then* $f : a \to b$ *iff both*

(a) f *is constant on* a *equivalence classes, and*

(b) $\mathrm{rng}(f) \subseteq \mathrm{inhab}(b)$.

Proof. If $f : a \to b$, then $f = b \circ f \circ a$, so $f = f \circ a$ is constant on a equivalence classes, and

$$\mathrm{rng}(f) = \mathrm{rng}(b \circ f) \subseteq \mathrm{rng}(b) = \mathrm{inhab}(b)$$

Conversely, if (a) and (b) hold,

$$\begin{aligned} f &= f \circ a && \textit{by (a)} \\ &= b \circ f \circ a && \textit{by (b)} \\ &= (a \to b)\ f \\ &= \mathbf{V}(a \to b)\ f && \textit{since } a, b : \mathbf{V} \end{aligned}$$

whence $f : a \to b$. □

A special type of endomorphisms are the idempotents

```
Idem := (V → V) (λa. P U. Endo a).
!check U = Idem ⊥.
```

For products, we want something like system F's $\forall c.\ (a \to b \to c) \to c$, but need to disambiguate to prevent e.g. $\langle \mathbf{K}, \mathbf{F} \rangle \mid \langle \mathbf{F}, \mathbf{K} \rangle$.

```
Prod := (V → V → V) (
    λa, b. P (Simpleλc, c′. (a→b→c)→c′) (λq. (q K, q F))
).
!define prod := P (Simpleλc, c′. (any→any→c)→c′) (λq. (q K, q F)).
!check prod = Prod ⊥ ⊥.
!check Prod a b = P prod ((V a → V b → any) → any).
!check (x, y) : prod.
!check (∀a:V, b:V. (a x, b y) : Prod a b).
```

Products are actually dropped products, since $\langle \top, \top \rangle \neq \top = \top_{\mathsf{prod}}$. (Note however that the least pair is indeed $(\bot, \bot)$).

```
!check ( (⊤, ⊤), ⊤ : prod ).
!check prod ⊥ = (⊥, ⊥).
```

Remark. We will often use this style a polymorphic type Prod by a simpler type prod = Prod ⊥ ⊥. This denotational equivalent of type inference allows us, under the Curry-Howard correspondence, to insert untyped proof *sketches* into larger well-typed contexts. See 3.14 for some simple examples, and 6.1 for extended example of how this works.

The intro and elim forms are

```
Pair  :=  (∀a:V, b:V. a → b → Prod a b) (λ−, −, x, y, f. f x y).
pair  :=  (any → any → prod) (λx, y, f. f x y).
!check pair = Pair ⊥ ⊥.
!check Pair = (λa:V, b:V, x:a, y:b. (x, y)).
!check pair = (λx, y. (x, y)).                                  verifying notation
```

```
Pi1  :=  (∀a:V. Prod a nil → a) (λ(x, −). x).
Pi2  :=  (∀b:V. Prod nil b → b) (λ(−, y). y).
π1  :=  (prod→any) ⟨K⟩.
π2  :=  (prod→any) ⟨F⟩.
!check π1 = Pi1 ⊥.
!check π2 = Pi2 ⊥.
!check (λp:prod. (π1 p, π2 p)) ⊑ prod.            typing error prevents equality
```

Theorem 3.7.6. $\mathrm{inhab}(\mathsf{Prod\ a\ b}) = \{\top\} \cup \{(\mathsf{x}, \mathsf{y}) \mid \mathsf{x}{:}\mathsf{a},\ \mathsf{y}{:}\mathsf{b}\}$.

Proof. Any h.n.f. below $\mathsf{q}{:}\mathsf{Prod\ a\ b}$ must be $(\mathsf{a\ x}, \mathsf{b\ y})$ or $\top$. We construct the maximum such approximation: Its first component is $\mathsf{q}\ \mathbf{K}$. Its second component is $\mathsf{q}\ \mathbf{F}$. So disambiguate $:= \lambda \mathsf{q}.\ (\mathsf{q}\ \mathbf{K}, \mathsf{q}\ \mathbf{F})$ ensures sequentiality. □

Remark. Tuples with nil elements needn't be nil (but this can be checked):

```
!check (⊤, ⊤) : prod.
!check (⊤, ⊤) ⋢ ⊤.
```

thus this is really a "dropped" product. The usual inhabitation relation does not hold:

```
!check Prod nil nil ⋢ nil.
```

however, we will see in Section 3.10 that the theorem does hold for types as *checkable closures*.

For sums, we want something like System F's $\forall \mathsf{a}, \mathsf{b}.\ (\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{c}) \to \mathsf{c}$, but need to disambiguate to prevent e.g. $\mathsf{inl}\ \mathbf{K} \mid \mathsf{inr}\ \mathbf{F}$.

```
Sum  :=  (V→V→V) (
    λa, b. (Simpleλc, c′. (a→c) → (b→c) → c′)
         | λq. q ⊥ (q ⊤ ⊥)
             | q (q ⊥ ⊤) ⊥
             | λf, g. q (K I) ⊥ (q f ⊤)
                    | q ⊥ (K I) (q ⊤ g)
).
!define sum  :=  Sum ⊥ ⊥.
```

The intro forms are

```
Inl  :=  (∀a:V. a → Sum a ⊥) (λ−, x, f, −. f x).
Inr  :=  (∀b:V. b → Sum ⊥ b) (λ−, y, −, g. g y).
!define inl  :=  (λx, f, −. f x).
!define inr  :=  (λy, −, g. g y).
!check ( inl, inr : any→sum ).
!check inl = Inl ⊥.
!check inr = Inr ⊥.
!check (inl, inr)∘sum = sum.
```

Sums are actually dropped, lifted coproducts since $\mathsf{inl}\ \top \neq \top \neq \mathsf{inr}\ \top$ and $\mathsf{inl}\ \bot \neq \bot \neq \mathsf{inr}\ \bot$ modulo sum.

```
!check ( ⊤, inl ⊤, inr ⊤ : sum ).
!check ( ⊥, inl ⊥, inr ⊥ : sum ).
```

Theorem 3.7.7. $\mathrm{inhab}(\mathsf{Sum\ a\ b}) = \{\top,\bot\} \cup \{\mathsf{inl\ x} \mid \mathsf{x:a}\} \cup \{\mathsf{inr\ y} \mid \mathsf{y:b}\}$.

Proof. Combine the proofs of bool and Prod. □

For the successor type $1 + \mathsf{a}$ we define an operation like System F's $\forall \mathsf{a}.\ \mathsf{b} \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{b}$, disambiguating as with sums.

```
Maybe  :=  (V → V) (
    λa. (Simpleλb, b′. b → (a→b) → b′)
      | (λn, −. n,  λx, −, s. s x)              descend,  =  ⟨none,  some⟩
      | λq. q ⊥ (q ⊤ ⊥)
          | q (q ⊥ ⊤) ⊥
).
!define maybe  :=  Maybe ⊥.
```

The intro forms are

```
!define none  :=  maybe (λn, −. n).
Some  :=  (∀a:V. Maybe a) (λa, x, −, s. s x).
!define some  :=  maybe (λx, −, s. s x).
!check some = Some ⊥.
!check none : Maybe a.
```

Theorem 3.7.8. $\mathrm{inhab}(\mathsf{Maybe\ a}) = \{\top,\bot,\mathsf{none}\} \cup \{\mathsf{some\ x} \mid \mathsf{x:a}\}$.

Proof. Similar to proof of Sum. □

The Maybe together with its mapping functions form a functor.

```
map_Maybe  :=  (∀a:V, b:V. (a→b) → Maybe a → Maybe b) (
    λ−, −, f. (none,  some∘f)
).
map_maybe  :=  (any → maybe → maybe) (λf. (none, some∘f)).
!check map_maybe = map_Maybe ⊥ ⊥.
```

The naming of Maybe already hints at the Curry-Howard correspondence, whereby we interpret type constructors as logical connectives. When defining types of logical statements, we will often use the additional terminology

```
And  :=  Prod.
Or  :=  Sum.
If  :=  Exp.
```

Recursive types

We can define recursive types as fixedpoints of type constructors, e.g. the dual types num and Stream a of algebraic numerals and streams. Numerals have introduction forms none and some

```
num  :=  V (Yλa. Maybe a).
!check none : num.                       zero
!check (∀n:num. some n:num).             successor
```

To avoid streams that typecheck for a while but eventually err, we disambiguate with $\lambda \mathsf{q}.\ \mathsf{q}\lambda -, \mathsf{q}'.\ \mathsf{q}'\lambda -, -.\ \bot$, pulling errors to the head.

```
Stream  :=  (V → V) (λa. Yλb. Prod a b | λq. qλ−, q′. q′λ−, −. ⊥).
stream  :=  V (Yλa. Prod any a | λq. qλ−, q′. q′λ−, −. ⊥).
!check stream = Stream ⊥.
!check ⊤ = stream (⊥, ⊤) = stream (⊥, ⊥, ⊤) = stream (⊥, ⊥, ⊥, ⊤).
```

Now duality $\mathsf{Stream\ a} <: \mathsf{num} \to \mathsf{a}$ is expressed as

```
| !check (∀a:V, s:Stream a, n:num. s n:a).
```

Intersection types

Intersection types are definable for closures (but not general idempotents). Recall that

```
| !check P = (V → V → V) (λa, b. a | b).
```

Theorem 3.7.9. $\mathrm{inhab}(\mathbf{P}\ \mathsf{a}\ \mathsf{b}) = \mathrm{inhab}(\mathsf{a}) \cap \mathrm{inhab}(\mathsf{b})$.

Proof. Let $\mathsf{a}, \mathsf{b} : \mathbf{V}$ be types.

- $\subseteq$: If $\mathsf{x} : \mathbf{P}\ \mathsf{a}\ \mathsf{b}$ then $\mathsf{x} = \mathbf{P}\ \mathsf{a}\ \mathsf{b}\ \mathsf{x} \sqsupseteq \mathsf{a}\ \mathsf{x} \sqsupseteq \mathsf{x}$ and similarly $\mathsf{x} : \mathsf{b}$.
- $\supseteq$: If $\mathsf{x} : \mathsf{a}$ and $\mathsf{x} : \mathsf{b}$ then $(\mathsf{a} \mid \mathsf{b})\mathsf{x} = \mathsf{a}\ \mathsf{x} \mid \mathsf{b}\ \mathsf{x} = \mathsf{x} \mid \mathsf{x} = \mathsf{x}$, so $\mathbf{P}\ \mathsf{a}\ \mathsf{b}\ \mathsf{x} = \mathbf{V}(\mathsf{a} \mid \mathsf{b})\mathsf{x} = \mathsf{x}$.

$\square$

Intersection are also defined over enumerably-many types, say in a Stream as defined below:

$$\mathbf{P}_{\mathsf{stream}} := (\mathsf{Stream}\ \mathbf{V} \to \mathbf{V})\ (\mathbf{Y}\lambda \mathsf{a} : \mathsf{Stream}\ \mathbf{V} \to \mathbf{V}, (\mathsf{h}, \mathsf{t}).\ \mathbf{P}\ \mathsf{h}(\mathsf{a}\ \mathsf{t})).$$

Lower-powerdomains support singletons, unions, mapping, and restriction by a semipredicate; so we call them *semisets*. We represent semisets as ambiguous terms of simple type $(\mathsf{any} \to \mathsf{a}) \to \mathsf{a}$, e.g.

$$\{\} = \bot$$
$$\{0\} = \langle 0 \rangle$$
$$\{0,1,2\} = \langle 0 \rangle \mid \langle 1 \rangle \mid \langle 2 \rangle$$
$$\{0,1,2,\dots\} = (\mathbf{Y}\lambda \mathsf{n}.\ \langle \mathsf{n} \rangle \mid \mathsf{y}(\mathsf{succ\ n}))0$$

```
| Sset := (V → P (Above ⟨⊥⟩)) (λa. Simpleλb, b'. (a→b)→b').
| !define sset := P (Above ⟨⊥⟩) (Simpleλa, a'. (any→a)→a').
| !check sset = Sset ⊥.
| !check sset ⊥ = ⟨⊥⟩.
| !check ( ⊤, ⟨⊤⟩ : sset ).
```

```
| map_Sset := (∀a:V, b:V. (a→b) → Sset a → Sset b) (λ−, −, f, ⟨x⟩. ⟨f x⟩).
| map_sset := (any → sset → sset) (λf, ⟨x⟩. ⟨f x⟩).
| !check map_sset = map_Sset ⊥.
```

Theorem 3.7.10. $\mathrm{inhab}(\mathsf{Sset}) = \{\top\} \cup \{\bigsqcup_{\mathsf{x} \in \mathcal{M}} \langle \mathsf{x} \rangle \mid \mathcal{M} \subseteq \mathrm{inhab}(\mathsf{a})\}$.

Proof. By the Simple type theorem, where $\bot$ is the only term raised by $\langle \bot \rangle$ $\square$

Note that Sset, map_Sset are the components of the functor of a monad[2] for ambiguity, with components

$$\begin{array}{rl}
(\mathbf{V} \to \mathbf{V})\ \mathsf{Sset} & \textit{action on objects} \\
(\forall \mathsf{a} : \mathbf{V}, \mathsf{b} : \mathbf{V}.\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{Sset\ a} \to \mathsf{Sset\ b})\ \mathsf{map\_Sset} & \textit{action on arrows} \\
(\forall \mathsf{a} : \mathbf{V}.\ \mathsf{a} \to \mathsf{Sset\ a})\ (\lambda -, \mathsf{x}.\ \langle \mathsf{x} \rangle) & \textit{unit} \\
(\forall \mathsf{a} : \mathbf{V}.\ 2\ \mathsf{Sset\ a} \to \mathsf{Sset\ a})\ (\lambda -, \langle \mathsf{x} \rangle.\ \mathsf{x}) & \textit{join}
\end{array}$$

[2]See [Wad90] and [Mog91] for definition.

Dependent types

We have already been using quantified types in the notation

$$\forall \mathsf{a}:\mathbf{V}.\mathsf{M} \quad := \quad \lambda \mathsf{a}:\mathbf{V},\mathsf{x}.\ \mathsf{M}(\mathsf{x}\ \mathsf{a})$$

Now we build concrete universal and existential type constructors, defined by the typing rules for indexed products and sums

$$\frac{\mathsf{x}:\Pi\ \mathsf{a}\ \mathsf{f} \qquad \mathsf{i}:\mathsf{a}}{\mathsf{x}\ \mathsf{i}:\mathsf{f}\ \mathsf{i}}\ (\Pi) \qquad\qquad \frac{(\mathsf{i},\mathsf{x}):\Sigma\ \mathsf{a}\ \mathsf{f}}{\mathsf{i}:\mathsf{a}}\ (\Sigma.1) \qquad\qquad \frac{(\mathsf{i},\mathsf{x}):\Sigma\ \mathsf{a}\ \mathsf{f}}{\mathsf{x}:\mathsf{f}\ \mathsf{i}}\ (\Sigma.2)$$

We represent the universally quantified type $\Pi\ \mathsf{a}\ \mathsf{f} = \forall \mathsf{i}:\mathsf{a}.\ \mathsf{f}\ \mathsf{i}$ by

```
Π  :=  (∀a:V. (a→V)  →  V) (λ−, f, x, i. f i(x i)).
!check (∀a:V, f:a→V. Π a f = (∀i:a. f i)).           verifying notation
!check (∀a:V, f:a→V, x:Π a f, i:a. x i:f i).         rule (Π)
```

and the existentially quantified type $\Sigma\ \mathsf{a}\ \mathsf{f} = \exists \mathsf{x}:\mathsf{a}.\ \mathsf{f}\ \mathsf{x}$ by

```
Σ  :=  (∀a:V. (a→V)  →  P prod) (λa, f, (i, x). (a i, f i x)).
!check (∀a:V, f:a→V. Σ a f = (∃i:a. f i)).                   verifying notation
!check (∀a:V, f:a→V, (i, x):Σ a f.
    i:a   AND   x:f i                                        rules (Σ.1) and (Σ.2)
).
```

Because there is a type-of-types $\mathbf{V}$, these dependent types encompass kinds of polymorphic types. For example

```
!check Exp : (∀a:V. V → P (From a)).
!check Exp : V → (∀b:V. P (To b)).
```

Using dependent types, we can even define a type of definable functor

```
functor  :=  (∃f:V→V. ∀a:V, b:V. (a → b) → f a → f b).
```

Which types, e.g., the semiset monad and the Maybe functor.

```
!check (Sset, map_Sset) : functor.
!check (Maybe, map_Maybe) : functor.
```

Symmetry types

An interesting feature of the types-as-closures paradigm is the definability of quotient types for objects invariant under a monoid action. For example the two-element group $\{\mathbf{I},\mathbf{C}\} = \{\lambda \mathsf{f},\mathsf{x},\mathsf{y}.\mathsf{f}\ \mathsf{x}\ \mathsf{y},\ \ \lambda \mathsf{f},\mathsf{x},\mathsf{y}.\mathsf{f}\ \mathsf{y}\ \mathsf{x}\}$ (acting as always by left-application), permutes the first two arguments of a function. This action induces the type of symmetric binary functions

```
Sym_bin_fun  :=  (V → V → P C) (λa, b. a→a→b).
sym_bin_fun  :=  V C.
!check sym_bin_fun = Sym_bin_fun ⊥ ⊥.
!check sym_bin_fun <: (λf, x, y. f x y).
!check sym_bin_fun <: (λf, x, y. f y x).
```

```
!check ( J, P, 2 K x : sym_bin_fun ).
!check J = sym_bin_fun K = sym_bin_fun F.
```

As a special case, we will use commutative binary operations

```
Comm_op := (V → V) (λa. P sym_bin_fun. Bin_op a).
!check Comm_op a = Sym_bin_fun a a.
!check sym_bin_fun = Comm_op ⊥.
```

Now we can conveniently define some logical types

```
Pred := (V → V) (λa. a→bool).                       predicate
Bin_rel := (V → V) (λa. a→a→bool).                  binary relations
Sym_bin_rel := (λa. P sym_bin_fun. Bin_rel a).       symmetric binary relation
```

An advantage of types-as-closures over types-as-projections is the definability of quotients by *infinite* monoids. Consider for example functions of tails of sequences, which are invariant under shifts in either direction:

```
Fun_of_tail := (V → V → V) (
    λa:V, b:V.
    λf:Stream a → b.
    fwd := (λs:Stream a. (⊥, s)).
    bwd := (λ(−, t):Stream a. t).
    f | f∘fwd | f∘bwd
).
```

An example of a function symmetric under an uncountable symmetry monoid (in fact group) is the Join_stream function on streams, invariant under the group of permutations of ω.

More generally, we can define the quotient of semisets

```
Mod := (∀a:V. Sset (a→a) → P (Sset a)) (
    λ−, ms, xs. msλm. xsλx. ⟨m x⟩
).
mod := Mod ⊥.
!check mod ⟨C⟩ ⟨K⟩ = ⟨K⟩ | ⟨F⟩.
```

If we further have an apartness assertion assert_neq : a → a → unit for some algebra, we can construct true quotients.

```
Quotient := (∀a:V. Sset (a→a) → P C (a→a→unit) → P (Sset a)) (
    λa, sym, assert_neq.   λxs.
    (xsλx. xsλx′. assert_neq x x′).
    symλs. xsλx. ⟨s x⟩
).
```

Open Questions

Question 3.7.11. *Are intro-elim forms definable uniformly from a simple type code?*

Question 3.7.12. *Does* SKJ *have all singleton closures, i.e. is there for every* SKJ *-term* m *a closure* unit_m *with inhabitants only* $\{\mathsf{m}, \top\}$*? If so, is this uniformly definable from codes for* m*?*

Definition 3.7.13. Let M be a term and define

$$\mathsf{unit}_\mathsf{M} \;:=\; \bigsqcup \{\mathsf{a}:\mathbf{V} \;\mid\; \mathsf{M}:\mathsf{a}\}$$

to be the *principle type* of M.

Lemma 3.7.14. $\mathrm{inhab}(\mathsf{unit}_\mathsf{M}) = \{\mathsf{M}, \top\}$

Proof. Clearly $\mathsf{M}, \top : \mathsf{unit}_\mathsf{M}$. Moreover, $\mathsf{M} : \mathbf{J}\ \mathsf{M}$, so $\mathbf{J}\ \mathsf{M} :> \mathsf{unit}_\mathsf{M}$ and every inhabitant of unit_M extends M. Let $\mathsf{M}' \not\sqsupseteq \mathsf{M}$ be such an inhabitant. Then by definition of $\mathcal{H}^*$, there is a term N with $\mathsf{N}\ \mathsf{M} = \bot$ and $\mathsf{N}\ \mathsf{M}' = \top$. But $\mathsf{M} : \mathbf{V}\ \mathsf{N}$ so $\mathbf{V}\ \mathsf{N} :> \mathsf{unit}_\mathsf{M}$. Hence $\mathsf{M}' \;=\; \mathsf{unit}_\mathsf{M}\ \mathsf{M}' \;\sqsupseteq\; \mathbf{V}\ \mathsf{N}\ \mathsf{M}' \;=\; \top$. □

Question 3.7.15. *For which* M *is* unit_M *definable?*

For example we already have principle types for $\mathbf{I}, \top, \bot, \mathbf{K}, \mathbf{F}$, namely $\mathsf{unit}_\mathbf{I} = \mathsf{unit}$, $\mathsf{unit}_\top = \mathsf{nil}$, $\mathsf{unit\_Bot} = \mathsf{div}$, $\mathbf{P}\ \mathsf{bool}\ (\mathbf{J}\ \mathbf{K}) = \mathsf{unit}_\mathbf{K}$, and $\mathbf{P}\ \mathsf{bool}\ (\mathbf{J}\ \mathbf{F}) = \mathsf{unit}_\mathbf{F}$.

Conjecture 3.7.16. unit_M *is not uniformly definable.*

Later in 5.7 we show that principle types of SKJ terms are uniformly definable in SKJO .

3.8 Axioms for a few simple types

Atomic types

The type unit is inhabited by $\{\mathbf{I}, \top\}$.

```
!assume ( I,⊤ : unit ).
!assume unit x = (unit x)∘(unit x) = (unit x)(unit x).
```

Within the domain unit, application is the same as composition.

```
!assume (unit x)(unit y) = (unit x)∘(unit y).
```

The following axiom shemata are enforced for unit

$$\frac{\mathsf{x}\ \not\sqsubseteq\ \mathbf{I}}{\mathsf{unit}\ \mathsf{x} = \top} \qquad\qquad \frac{\mathsf{x}\ \mathbf{I}\ \sqsubseteq\ \mathbf{I}}{\mathsf{unit}\ \sqsubseteq\ \mathsf{x}}$$

The latter holds since unit is the largest function $\mathbf{I} \mapsto \mathbf{I}$.

The type semi is inhabited by $\{\bot, \mathbf{I}, \top\}$.

```
!assume ( ⊥,I,⊤ : semi ).
!assume semi x = (semi x)∘(semi x) = (semi x)(semi x)
               = (semi x)∘(unit x) = (semi x)(unit x).
```

As with unit, application is the same as composition.

```
!assume (semi x)(semi y) = (semi x)∘(semi y).
```

The following inference rules are enforced for semi

$$\frac{\mathsf{x}\ \not\sqsubseteq\ \bot}{\mathsf{semi}\ \mathsf{x}\ \sqsupseteq\ \mathbf{I}} \qquad\qquad \frac{\mathsf{x}\ \not\sqsubseteq\ \mathbf{I}}{\mathsf{semi}\ \mathsf{x} = \top} \qquad\qquad \frac{\mathsf{x}\ \bot = \bot \qquad \mathsf{x}\ \mathbf{I}\ \sqsubseteq\ \mathbf{I}}{\mathsf{semi}\ \sqsubseteq\ \mathsf{x}}$$

The boolean type bool is inhabited by $\{\bot, \mathbf{K}, \mathbf{F}, \top\}$.

```
!assume ( ⊥, K, F, ⊤ : bool ).
```

We can thus check equations pointwise, e.g.,

```
!assume bool x = bool x K F
               = bool x I I (bool x)
               = bool x (bool x) (bool x)
               = bool x K ⊥ | bool x ⊥ F.
```

Type constructors

We will also add the four most basic type constructors to the basis.

```
!using prod sum maybe sset.
```

The polymorphic versions relate to the simple versions via $\mathbf{P}$.

```
!assume Prod a b = P prod ((a→b→any) → any).
!assume Sum a b = P sum ((a→any) → (b→any) → any).
!assume Maybe a = P maybe (any → (a→any) → any).
!assume Sset a = P sset ((a→any)→any).
```

Conversely, the simple versions are sketches of the polymorphic versions.

```
!assume prod = Prod ⊥ ⊥.
!assume sum = Sum ⊥ ⊥.
!assume maybe = Maybe ⊥.
!assume sset = Sset ⊥.
```

We also characterize typical inhabitants

```
!assume (x, y) : prod.
!assume ( ⊥, inl x, inr y : sum ).
!assume ( ⊥, none, some x : maybe ).
```

Semisets are inhabited by singletons $\langle x\rangle$ and nonempty finitary joins, which we achieve through binary joins sset x | sset y.

```
!assume ( ⟨x⟩, sset x | sset y : sset ).
```

3.9 Various-valued logics

In this section we develop logical calculi with four different truth values: boolean truth values for (two-sided) decidability, semiboolean truth values for positive semidecidability (e.g. convergence testing), and unit and divergent truth values for negative decidability (e.g. error checking). The four types turn out to have pracitcal utility for "programming" in SKJ , but there is also motivation from descriptive complexity theory.

In the theory $\mathcal{H}^*$, raw equality and information ordering between SKJ terms is Π^0_2-complete. But some verification problems are naturally of lower complexity, e.g. converge testing (a result can be found), or divergence testing (no errors can be found). The logical types div, unit, semi, bool provide restricted domains over which questions of lower complexity class, respectively $\Pi^0_1, \Pi^0_1, \Delta^0_2, \Delta^0_1$. For example determining whether $x = \mathbf{I}$ is Π^0_2-complete, but restricted to the type unit, the question $x{:}\mathsf{unit} \vdash x{:}\mathbf{I}$ is Π^0_1-complete.

In stating and proving the following motivational theorems, use some material from later sections, e.g. $\mathbf{P}_{\mathsf{test}}$ below and Join_nat and test_nat from 3.12.

Theorem 3.9.1. *The following complexity characterizations hold for raw problems in* SKJ .

	Problem	*Complexity*
(a)	$x = \bot$	Π^0_1*-complete*
(b)	$x = \mathbf{I}$	Π^0_2*-complete*
(c)	$x = \top$	Π^0_2*-complete*

Proof.
Upper Bounds:
(a) $x = \bot$ iff div $x = \bot$ iff no β-η reduction shows div $x = \top$.
The remaining cases reduce to information ordering, which is Π^0_2.

Hardness: Let $\phi{:}\mathsf{nat}\to\mathsf{bool}$ be a total recursive predicate.
(a) $\forall n.\ \phi(n)$ iff (Join_nat $\lambda n.\ \phi\ n\ \bot\ \mathbf{I}) = \bot$.
(b) By negating (a), $\exists m.\ \phi(m)$ can be posed as a problem $x = \mathbf{I}$, where we further know $x \in \{\bot, \mathbf{I}\}$. Now consider an infinite η-expansion of

$$\mathbf{I} = \lambda x, y_0.\ x\lambda y_1.\ y_0\lambda y_2.\ y_1\lambda y_3.\ y_2\lambda y_4.\ \ldots$$

Let $\psi(m)$ be a Σ^0_1 predicate where for each m,
$u_m = \mathbf{I} \iff \psi(m)$ and $u_m \in \{\bot, \mathbf{I}\}$. Then inserting u_m's into the expansion of $\mathbf{I}$,

$$\forall m.\psi(m) \iff \mathbf{I} = \lambda x, y_0.\ x\lambda y_1.\ u_0\ y_0\lambda y_2.\ u_1\ y_1\lambda y_3.\ u_2\ y_2\lambda y_4.\ \ldots$$

(c) Consider a definition of $\top$ as

$$\top = \bigsqcup_n \mathbf{K}^n$$

Note that any finite such join is not $\top$. Let $\psi(m)$ be a Σ^0_1 predicate where for each m, $u_m = \mathbf{I} \iff \psi(m)$ and $u_m \in \{\bot, \mathbf{I}\}$. Note that

$$\forall m < 1 + n.\ \psi(m) \iff \mathbf{I} = u_0 \circ u_1 \circ \ldots \circ u_n$$

so let us define $v_n = u_0 \circ u_1 \circ \ldots \circ u_n$, so that if $\psi(m)$ fails, then $v_n = \bot$ for all $n \geq m$. Now

$$\forall m.\ \psi(m) \iff \top = \bigsqcup_m v_n\ \mathbf{K}^n$$

□

The situation is much nicer in typed or typed-and-tested contexts. Our semantics for contexts $\Gamma \vdash M = N$ is the same as that for universal quantifers and hypothetical tests. Letting $a{:}\mathbf{V}$ and $t{:}\mathsf{test}$, we localize via

$$\begin{aligned}(x{:}a \ \vdash\ f\ x = g\ x) &\iff f{\circ}a\ x = g{\circ}a\ x\\ (x{::}t \ \vdash\ f\ x = g\ x) &\iff (t\ x)(f\ x) = (t\ x)(g\ x)\\ (x{:}a{::}t \ \vdash\ f\ x = g\ x) &\iff (t{\circ}a\ x)(f{\circ}a\ x) = (t{\circ}a\ x)(f{\circ}a\ x)\end{aligned}$$

Theorem 3.9.2. *The following complexity characterizations hold for local problems in* SKJ .

	Context $\vdash$ *Problem*	*Complexity*
(a)	$\mathsf{x}\!:\!\mathsf{div} \vdash \mathsf{x} = \bot$	Π^0_1*-complete*
(b)	$\mathsf{x}\!:\!\mathsf{div} \vdash \mathsf{x} = \top$	Σ^0_1*-complete*
(c)	$\mathsf{x}\!:\!\mathsf{unit} \vdash \mathsf{x} = \mathbf{I}$	Π^0_1*-complete*
(d)	$\mathsf{x}\!:\!\mathsf{semi} \vdash \mathsf{x} = \mathbf{I}$	*complete for differences between* Σ^0_1 *sets*
(e)	$\mathsf{x}\!:\!\mathsf{semi}::\mathsf{unit} \vdash \mathsf{x} = \mathbf{I}$	Σ^0_1*-complete*
(f)	$\mathsf{x}\!:\!\mathsf{bool}::\mathsf{test\_bool} \vdash \mathsf{x} = \mathbf{K}$	Δ^0_1*-complete*
(g)	$\vdash \mathsf{s} <:: \mathsf{t}$	Π^0_2*-complete*

Note that testing $\mathsf{x}::\mathsf{t}$ is equivalent to $\mathsf{semi}(\mathsf{t\ x}) = \mathbf{I}$, corresponding to case (c).

Proof. We can WLOG restrict terms $\mathsf{x}\!:\!\mathsf{a}$ for closures a, by raising incomplete x via $\mathsf{x} \mapsto \mathsf{a\ x}$.

(a) The problem $\mathsf{x} = \bot$ does not depend on $\mathsf{x}\!:\!\mathsf{div}$, so apply the previous theorem.

(b) Suppose $\mathsf{x}\!:\!\mathsf{div}$, i.e., $\mathsf{x} \in \{\bot, \top\}$. Then $\mathsf{div\ x} = \top$ iff $\mathsf{x} \neq \bot$.

(c) Suppose $\mathsf{x}\!:\!\mathsf{unit}$, i.e., $\mathsf{x} \in \{\bot, \mathbf{I}, \top\}$. Then $\mathsf{x} = \mathbf{I}$ iff $\mathsf{div\ x} = \bot$.

(d) Suppose $\mathsf{x}\!:\!\mathsf{semi}$, i.e., $\mathsf{x} \in \{\bot, \mathbf{I}, \top\}$. Then $\mathsf{x} \sqsubseteq \mathbf{I}$ iff $\mathsf{x}\ \bot = \bot$, which is Π^0_1-complete. Also, $\mathsf{x} \sqsupseteq \mathbf{I}$ iff $\mathsf{x}\ \top \neq \bot$, which is Σ^0_1-complete. Conjoining these two problems gives problem complete for differences between Σ^0_1 sets.

(e) Suppose $\mathsf{x}::\mathsf{semi}::\mathsf{unit}$, i.e., $\mathsf{x} \in \{\bot, \mathbf{I}\}$. Then $\mathsf{x} = \mathbf{I}$ iff $\mathsf{div}(\mathsf{x}\ \bot) \neq \bot$, which is Π^0_1-complete.

(f) Suppose $\mathsf{x}\!:\!\mathsf{bool}::\mathsf{test\_bool}$, i.e., $\mathsf{x} \in \{\mathbf{K}, \mathbf{F}\}$. Then $\mathsf{x} = \mathbf{K}$ iff $\mathsf{div}(\mathsf{x}\ \bot\ \top) = \bot$ iff $\mathsf{div}(\mathsf{x}\ \top\ \bot) \neq \bot$. The former is Π^0_1-complete and the latter is Σ^0_1-complete.

(g) First we can reduce the subtest problem to $\mathsf{s} <:: \mathsf{t}$ iff $\forall \mathsf{x}::\mathsf{s}.\ \mathsf{x}::\mathsf{t}$. Now the inner testing problem is equivalent to $\mathsf{x}::\mathsf{t}$, which is Σ^0_1-complete. Letting $\mathsf{s} = \mathsf{test\_nat}$,

□

Later in 5.6 we will extend these to hyperarithmetic analogs.

Boolean Logic

We begin with Boolean logic.

```
!check true = K = bool (λx, −. x).
!check false = F = bool (λ−, y. y).
```

```
pred := any→bool.
!check Pred = (V → V) (λa. a→bool).
```

we also have unit types inhabited by the truth values

```
unit_K := P bool (Above K).
unit_F := P bool (Above F).
```

The logical operations are all SKJ -definable.

```
not := (bool → bool) C.
!check not = (F, K)∘bool.
!check not∘not = bool.
```

Following Plotkin, we define parallel versions of and and or, initially using symmetry types

```
and := Comm_op bool (λx, y. x y F).
or := Comm_op bool (λx, y. x K y).
iff := Comm_op bool (λx, y. x y. not y).
```

The parallel implication requires a little more work. For this we first define Dijkstras guarded commands

```
if := (bool → semi) (I, ⊥).
```

which can be joined together, as in

```
implies := (bool → bool → bool) (
    λx, y. if y K
         | if (not x) K
         | if (and x (not y)) F
).
```

and the converse

```
impliedby := C implies.
```

Note the relation to the sequential implication

```
!check implies ⊒ (λx:bool, y:bool. x y K).
!check implies ⋢ (λx:bool, y:bool. x y K).
```

Using this technique we can redefine and and or via

```
!check or = Bin_op bool (
    λx, y. if x K
         | if y K
         | if (not x). if (not y) F
).
!check and = Bin_op bool (
    λx, y. if (not x) F
         | if (not y) F
         | if x. if y K
).
```

Weak Logics

Weaker than boolean logic is *testing logic* with truth values : semi.

```
test := any → semi.
Test := (V → V) (λa. a → semi).
!check test = Test ⊥.
!check ( K I, K ⊥, K ⊤ : test ).        succeed/fail/err everywhere
```

Weaker still are *probing logic* with truth values : div

```
probe := any → div.
Probe := (V → V) (λa. a → div).
!check probe = Probe ⊥.
!check ( K ⊥, K ⊤ : probe ).        the minimum and maximum probes
```

and *checking logic* with truth values : unit

```
check := any → unit.
Check := (V → V) (λa. a → unit).
!check check = Check ⊥.
!check ( K I, K ⊤ : check ).        the minimum and maximum checks
```

Note that checks can easily be made into closures:

```
!check W∘check : check→V.
Checked := (∀a. Check a → P a) (λ−, c, x. c x x).
!check Checked ⊥ ⊥ = I.
!check Checked a ⊥ = V a.
!check Checked ⊥ c x = check c x x.
!check Checked a c <: V a.
!check Check (Checked a c) <: Check a.
!check Check a c : Check (Checked a c).
```

The logics have meet and join operations, but no negation.

```
or_semi  := Comm_op semi (λx, y. x).    i.e., J
or_unit  := Comm_op unit (λx, y. x).
or_div   := Comm_op div (λx, y. x).
```

```
and_semi  := Comm_op semi (λx, y. x y).
and_unit  := Comm_op unit (λx, y. x y).
and_div   := Comm_op div (λx, y. x y).
```

```
!check or_semi ⊤ = ⊤.
!check or_semi ⊥ = semi.
!check or_semi I I = I.
!check and_semi ⊤ = ⊤.
!check and_semi ⊥ ⊥ = ⊥.
!check and_semi ⊥ I = ⊥.
!check and_semi I I = I.
```

Note that checking logic simply checks for errors, so

```
!check and_unit = or_unit.
!check and_div = or_div.
```

Checking logic is used in assertions and subtyping:

```
assert  := (bool → unit) (I, ⊤).
P_where := (∀a:V. (a→unit) → P a) (λ−, w, x. w x x).
```

Definition 3.9.3. We refer to bool−, semi−, unit− and div− valued functions as *predicates*, *tests*, *checks*, and *probes*, respectively.

The motivation is that a term passes a check if no error is found (as in static typechecking); a term passes a test if no error is found and in addition it performs some computation (as in a unit test); and a term passes a probe if it doesn't blow up when mucked with. We can weaken any test by forcing convergence

```
test2check := (test → check) (λt, x. I | t x).
```

and checks are naturally tests

```
!check check <: test.
```

For each weak predicate type we can define implication / subobjecthood as follows:

Definition 3.9.4. (weak implication)

- Let $\mathsf{c}, \mathsf{c}' : \mathsf{check}$ be checks. Then c is a *subcheck* of c' iff $\mathsf{c} \sqsupseteq \mathsf{c}'$.
- Let $\mathsf{p}, \mathsf{p}' : \mathsf{probe}$ be probes. Then p is a *subprobe* of p' iff $\mathsf{p} \sqsupseteq \mathsf{p}'$.
- Let $\mathsf{t}, \mathsf{t}' : \mathsf{test}$ be tests. Then t is a *subtest* of t' iff $\mathsf{t} = \mathbf{P}_{\mathsf{test}}\ \mathsf{t}\ \mathsf{t}'$ where

$$\mathbf{P}_{\mathsf{test}} = \mathsf{Sym\_bin\_op\ \ test}\ \ (\lambda \mathsf{s}, \mathsf{t}, \mathsf{x}.\ \ \mathsf{and\_test}\ \ (\mathsf{s\ \ x})\ \ (\mathsf{t\ \ x})).$$

We use the notation $\mathsf{x} :: \mathsf{t} \iff \mathsf{test\ \ t\ \ x} = \mathbf{I}$ to say that x has passed test t, and the notation $\mathsf{s} <:: \mathsf{t}$ to say that s is a subtest of t.

Thus to intersect / conjoin checks and probes we need only join them. but to combine tests, we consider errors are more important than failures.

$$\begin{array}{|lll}
\mathsf{!define}\ \mathbf{P}_{\mathsf{test}} & := & \mathsf{Comm\_op\ \ test}\ \ (\lambda \mathsf{p}, \mathsf{q}, \mathsf{x}.\ \ \mathsf{and\_semi}(\mathsf{p\ \ x})(\mathsf{q\ \ x})). \\
\mathbf{P}_{\mathsf{check}} & := & \mathsf{Comm\_op\ \ check}\ \ (\lambda \mathsf{p}, \mathsf{q}, \mathsf{x}.\ \ \mathsf{and\_unit}(\mathsf{p\ \ x})(\mathsf{q\ \ x})). \\
\mathbf{P}_{\mathsf{probe}} & := & \mathsf{Comm\_op\ \ probe}\ \ (\lambda \mathsf{p}, \mathsf{q}, \mathsf{x}.\ \ \mathsf{and\_div}(\mathsf{p\ \ x})(\mathsf{q\ \ x})).
\end{array}$$

3.10 Tests for totality and correctness

Types as tests

Following the interpretation of types-as-closures of 3.3, we will pursue an alternate interpretation of types-as-tests, with a similar convenient notation. Moreover, the two interpretations of types cohabit gracefully.

Definition 3.10.1. (types as tests)

$$\begin{array}{rcll}
(\lambda \mathsf{x} :: \mathsf{t}.\mathsf{x}) & = & (\lambda \mathsf{x}.\mathsf{semi}(\mathsf{t\ \ x})\mathsf{x}) & \textit{tested abstraction} \\
\mathsf{x} :: \mathsf{t} & \iff & \mathsf{semi}(\mathsf{t\ \ x}) = \mathbf{I} & \textit{passing a test} \\
\forall \mathsf{x} :: \mathsf{t}.\ \ \mathsf{M} = \mathsf{N} & \iff & (\lambda \mathsf{x} :: \mathsf{t}.\mathsf{M}) = (\lambda \mathsf{x} :: \mathsf{t}.\mathsf{N}) & \textit{universal quantifiers} \\
\mathsf{s} <:: \mathsf{t} & \iff & \mathsf{test\ \ s} = \mathbf{S}(\mathsf{test\ \ t})(\mathsf{test\ \ s}) & \textit{sub-testing} \\
(\mathsf{s} :: \mathsf{t} \implies \mathsf{M} \sim \mathsf{N}) & \iff & \mathsf{test\ \ t\ \ s\ \ M} \sim \mathsf{test\ \ t\ \ s\ \ N} & \textit{tests as hypotheses}
\end{array}$$

where in the conclusion $\mathsf{M} \sim \mathsf{N}$, the relation $\sim$ is one of $=, \sqsubseteq, \sqsupseteq$.

Lemma 3.10.2. *The following are always true.*

$$\begin{array}{|l}
\mathsf{!check}\ \ (\ \ \forall \mathsf{x} :: \mathsf{t}.\ \ \mathsf{x} :: \mathsf{t}\ \ ). \\
\mathsf{!check}\ \ \mathsf{x} :: \mathsf{t} \implies \mathsf{x} :: \mathsf{t}. \\
\mathsf{!check}\ \ \mathsf{x} :: \mathsf{t} \implies \mathsf{m} = \mathsf{m}. \\
\mathsf{!check}\ \ \top :: \mathbf{I} \implies \bot = \top.
\end{array}$$

Proof. Letting $\mathsf{i} = \mathsf{test\ \ t\ \ x}$ below,

$$\begin{array}{rcl}
\forall \mathsf{x} :: \mathsf{t}.\ \ \mathsf{x} :: \mathsf{t} & \iff & \forall \mathsf{x} :: \mathsf{t}.\ \ \mathsf{test\ \ t\ \ x}) = \mathbf{I} \\
 & \iff & (\lambda \mathsf{x} :: \mathsf{t}.\ \ \mathsf{test\ \ t\ \ x}) = (\lambda \mathsf{x} :: \mathsf{t}.\mathbf{I}) \\
 & \iff & (\lambda \mathsf{x}.\mathsf{test\ \ t\ \ x}.\mathsf{test\ \ t\ \ x}) = (\lambda \mathsf{x}.\mathsf{test\ \ t\ \ x}\ \ \mathbf{I}) \\
 & \iff & (\lambda \mathsf{x}.\ \ \mathsf{i\ \ i}) = (\lambda \mathsf{x}.\ \ \mathsf{i}\ \ \mathbf{I}) \\
(\mathsf{x} :: \mathsf{t} \implies \mathsf{x} :: \mathsf{t}) & \iff & (\mathsf{x} :: \mathsf{t} \implies \mathsf{test\ \ t\ \ x} = \mathbf{I}) \\
 & \iff & \mathsf{test\ \ t\ \ x}\ \ (\mathsf{test\ \ t\ \ x}) = \mathsf{test\ \ t\ \ x}\ \ \mathbf{I} \\
 & \iff & \mathsf{i\ \ i} = \mathsf{i}\ \ \mathbf{I}
\end{array}$$

In either case, we can we can check i pointwise over $\mathrm{inhab}(\mathsf{semi}) = \{\bot, \mathbf{I}, \top\}$:

$$\begin{array}{l}
\bot\ \ \bot = \bot = \bot\ \ \mathbf{I} \\
\mathbf{I}\ \ \mathbf{I} = \mathbf{I} = \mathbf{I}\ \ \mathbf{I} \\
\top\ \ \top = \top = \top\ \ \mathbf{I}
\end{array}$$

The last two implications are trivial. □

Lemma 3.10.3. *Subtesting is a preorder.*

(a) $\mathsf{t} <:: \mathsf{t}$.

(b) *If* $\mathsf{r} <:: \mathsf{s}$ *and* $\mathsf{s} <:: \mathsf{t}$ *then* $\mathsf{r} <:: \mathsf{t}$.

Proof. Assume WLOG $\mathsf{r}, \mathsf{s}, \mathsf{t} : \mathsf{test}$

(a) $\mathsf{t} <:: \mathsf{t}$ iff $\mathsf{t} = \mathbf{S}\ \mathsf{t}\ \mathsf{t}$, which holds pointwise since $\mathbf{I}\ \mathbf{I} = \mathbf{I}$, $\bot\ \bot = \bot$, and $\top\ \top = \top$.

(b) Noting that application and composition are the same for semibooleans

```
| !check (semi x)(semi y) = (semi x)∘(semi y).
```

we can again look at the pointwise situation.

$$\begin{aligned} \mathsf{r\ x} &= (\mathsf{r\ x})\circ(\mathsf{s\ x}) && \textit{since } \mathsf{r} <:: \mathsf{s} \\ &= (\mathsf{r\ x})\circ(\mathsf{s\ x})\circ(\mathsf{t\ x}) && \textit{since } \mathsf{s} <:: \mathsf{t} \\ &= (\mathsf{r\ x})\circ(\mathsf{t\ x}) && \textit{since } \mathsf{r} <:: \mathsf{t} \end{aligned}$$

whence $\mathsf{r} <:: \mathsf{t}$.

□

However, subtesting is not a partial order, even on test, since antisymmetry fails, e.g. $\bot <:: \top$ and $\top <:: \bot$. The problem is that tests can fail in two directions ($\bot$ and $\top$), but subtesting only ask whether, not how tests fail.

Atomic tests and checks.

We define atomic tests for a few tyes

```
| test_any  :=  Test any (K I).        maximal test: everything passes
| test_nil  :=  Test nil ⟨⟩.           minimal test: nothing passes
| test_div  :=  Test div ⟨⟩.
| test_unit  :=  Test unit ⟨I⟩.
| test_semi  :=  Test semi ⟨I⟩.
| test_bool  :=  Test bool ⟨I, I⟩.
| !check test_any = K I.
```

Note that the subtypes of semi are their own tests:

```
| !check test_nil = nil.
| !check test_div = div.
| !check test_unit = unit.
| !check test_semi = semi.
```

Checks can now be defined in terms of tests.

```
| check_any  :=  test2check test_any.
| check_nil  :=  test2check test_nil.
| check_unit  :=  test2check test_unit.
| check_semi  :=  test2check test_semi.
| check_bool  :=  test2check test_bool.
```

The subtypes of unit are their own checks:

```
| !check check_nil = nil.
| !check check_unit = unit.
```

Compound Tests

In defining compound tests, we first extend to the categories of testable types and checkable types

```
testable  :=  (∃a:V. test a).
checkable  :=  (∃a:V. check a).
Testable  :=  (P V → testable) (λt. ∃a:t. test a).
Checkable  :=  (P V → checkable) (λt. ∃a:t. check a).
!check testable = Testable ⊥.
!check checkable = Checkable ⊥.
```

For example we can define

```
tb_unit  :=  ⟨unit, test_unit⟩.
tb_semi  :=  ⟨semi, test_semi⟩.
tb_bool  :=  ⟨bool, test_bool⟩.
```

Remark. Testing exponentials requires enumerating the compact elements of the domain, which is not always easy (or possible). Hence these categories are not closed.

Now we can define constructors for testables

```
test_prod  :=  ⟨2 K I⟩∘prod.
tb_Prod  :=  (testable → testable → testable) (
    λ⟨a, t_a⟩, ⟨b, t_b⟩. (
        Prod a b,
        λp. and_semi (test_prod p). and_semi (t_a (p K)) (t_b (p F))
    )
).
!check (∀aa:testable, bb:testable.
    let (−, a)  :=  aa.
    let (−, b)  :=  bb.
    let (−, ab)  :=  tb_Prod aa bb.
    ∀x::a, y::b. (x, y) :: ab
).
```

```
tb_Sum  :=  (testable → testable → testable) (
    λ⟨a, t_a⟩, ⟨b, t_b⟩. (
        Sum a b,
        λp. and_semi (p (K I) (K I)) (p t_a t_b)
    )
).
!check (∀aa:testable, bb:testable.
    let (−, a)  :=  aa.
    let (−, b)  :=  bb.
    let (−, ab)  :=  tb_Sum aa bb.
    (∀x::a. inl x :: ab) AND (∀y::b. inr y :: ab)
).
```

```
tb_Maybe  :=  (testable  →  testable) (
    λ⟨a, t_a⟩. (
        Maybe  a,
        λp. and_semi (p I (K I)) (p I t_a)
    )
).
!check (∀aa : testable.
    let (−, a)  :=  aa.
    let (−, ma)  :=  tb_Maybe aa.
    (∀x :: a. some x :: ma)  AND  (none :: ma)
).
```

and more weakly-typed versions for tests and checks only

```
test_Prod  :=  (test → test → Test prod) (
    λs, t, (x, y). and_semi (s x) (t y)
).
test_Sum  :=  (test → test → Test sum) (λs, t. (s, t)).
test_Maybe  :=  (test → Test maybe) (λt. (I, t)).
```

```
check_Prod  :=  (check → check → Check prod) (λs, t, (x, y). s x | t y).
check_Sum  :=  (check → check → Check sum) (λs, t. (s, t)).
check_Maybe  :=  (check → Check maybe) (λt. (I, t)).
```

Semisets pass a test if at least one element converges, and no diverge.

```
test_sset  :=  Test sset (λx. x (K I)).
test_Sset  :=  P_test test_sset (Test sset) (λt. ⟨t⟩).
!check test_sset = test_Sset (K I).
```

3.11 Axioms for a few simple tests

The test intersection operator

The test intersection operator is

$$\mathbf{P}_{\mathsf{test}} = (\mathsf{test} \to \mathsf{test} \to \mathsf{test})\ (\lambda t, t', x.\ \mathsf{and\_semi}\ (t\ x)\ (t'\ x)).$$

The following axiom shemata will be enforced for the atom $\mathbf{P}_{\mathsf{test}}$

$$\frac{\mathsf{check}\ p\ \sqsupseteq\ \mathsf{check}\ q \qquad \mathsf{probe}\ p\ \sqsubseteq\ \mathsf{probe}\ q}{\mathsf{test}\ p\ <::\ \mathsf{test}\ q}$$

where the subtest relation <:: is defined as

$$p <:: q \quad \Longleftrightarrow \quad p = \mathbf{P}_{\mathsf{test}}\ q\ p$$

We also assume the ACI axioms, as mentioned in 3.9

```
!assume P_test p p = test p.                      idempotence
!assume P_test p q = P_test q p.                  commutativity
!assume P_test p(P_test q r) = P_test(P_test p q)r.   associativity
```

3.12 Church numerals

The type of Church numerals

```
nat  :=  V (
    Yλa. V. (Simple λb, b′. (b′ → b) → b → b′)
            | ⟨J, I, ⊥⟩
            | ⟨λn : a, f : a → a, x : a.f(n f x),  λ−, x : a.x⟩
            | λq. q ⊥ (q ⊤ ⊥)
                | q (q ⊥ ⊤) ⊥
).
check_nat  :=  Check nat ⟨λx. I | x,  I⟩.
test_nat  :=  Test nat ⟨λx. x,  I⟩.
```

with intro forms

```
zero  :=  nat (λf, x.x).
succ  :=  (nat → nat) (λn, f, x.f.n f x).
```

Theorem 3.12.1. $\mathrm{inhab}(\mathsf{nats}) = \{\top\} \cup \{\mathsf{succ}^{\mathsf{n}}\ \mathsf{z} \mid \mathsf{n} \in \mathsf{Nats},\ \mathsf{z} \in \{\bot,\ \mathsf{zero}\}\}$.

Proof. By the Simple types theorem, all inhabitants of the raw simple type can be written as

$$\frac{}{\top : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}} \qquad \frac{}{\bot : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}} \qquad \frac{}{\mathsf{zero} : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}} \qquad \frac{\mathsf{n} : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}}{\mathsf{succ}\ \mathsf{n} : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}}$$

$$\frac{\forall \mathsf{m} \in \mathsf{M}.\ \mathsf{m} : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}}{\bigsqcup \mathsf{M} : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}}$$

Of these, we want to eliminate incompatible joins and successors of $\top$ via disambiguation. The first disambiguation term $\langle \mathbf{J}, \mathbf{I}, \bot \rangle$ eliminates successors of $\top$ by raising them to the head, as in

```
!check ⟨J, I, ⊥⟩ (succ ⊤)
    =  succ ⊤ J I ⊥
    =  J (⊤ J I) ⊥
    =  J ⊤ ⊥
    =  ⊤.
```

and similarly for n succ $\top$. The third disambiguation term q. q $\bot$ (q $\top$ $\bot$) | q (q $\bot$ $\top$) $\bot$ prevents incompatible joins at the the top of the Böhm tree, as in

```
!check (
    zs  :=  zero | succ ⊥.
    zs ⊥ ⊤ = ⊤     AND
    zs ⊤ ⊥ = ⊤     AND

(λq. q ⊥ (q ⊤ ⊥) | q (q ⊥ ⊤) ⊥) zs
    =  zs ⊥ (zs ⊤ ⊥) | zs (zs ⊥ ⊤) ⊥
    =  zs ⊥ ⊤ | zs ⊤ ⊥
    =  ⊤
).
```

Finally, the third disambiguation term ⟨λn : nat, f : nat → nat, x : nat.f(n f x), λ−, x : nat.x⟩ applies the second disambiguator deeper in the Böhm tree, preventing successors of incompatible joins. □

Dual to the type nat is the type of sequences.

$$\begin{array}{l} \mathsf{Seq} \;:=\; (\mathbf{V} \to \mathbf{V})\ (\lambda \mathsf{a}.\ \mathsf{nat} \to \mathsf{a}). \\ \mathsf{seq} \;:=\; \mathsf{Seq}\ \bot. \end{array}$$

Som extra intro forms for nats are convenient.

$$\begin{array}{l} \mathsf{one} \;:=\; \mathsf{nat}\ (\lambda \mathsf{f}, \mathsf{x}.\mathsf{f}\ \mathsf{x}). \\ \mathsf{two} \;:=\; \mathsf{nat}\ (\lambda \mathsf{f}, \mathsf{x}.\mathsf{f}.\mathsf{f}\ \mathsf{x}). \\ !\mathsf{check}\ \omega = \mathsf{nat}\ (\lambda \mathsf{f}, \mathsf{x}.\mathbf{Y}\ \mathsf{f}). \\ !\mathsf{check}\ \mathsf{succ}\ \mathsf{zero} = \mathsf{one}. \\ !\mathsf{check}\ \mathsf{succ}\ \mathsf{one} = \mathsf{two}. \\ !\mathsf{check}\ \mathsf{succ}\ \omega = \omega. \\ !\mathsf{check}\ \mathbf{Y}\ \mathsf{succ} = \omega. \end{array}$$

We also have a two-sided test for zero

$$\begin{array}{l} \mathsf{is\_zero} \;:=\; (\mathsf{nat} \to \mathsf{bool})\ \langle \mathbf{K}\ \mathsf{false},\ \mathsf{true} \rangle. \\ \mathsf{nonzero} \;:=\; \mathsf{not} \circ \mathsf{is\_zero}. \\ !\mathsf{check}\ \mathsf{is\_zero}\ \mathsf{zero} = \mathsf{true}. \\ !\mathsf{check}\ \mathsf{is\_zero}(\mathsf{succ}\ \bot) = \mathsf{false}. \end{array}$$

and a one-sided test for infinity, recalling the definition of ω,

$$\begin{array}{l} !\mathsf{check}\ \omega = \mathbf{Y}\ \mathsf{succ}. \\ !\mathsf{check}\ \omega\ :\ \mathsf{nat}. \\ \mathsf{if\_finite} \;:=\; (\mathsf{nat} \to \mathsf{semi})\ \langle \mathbf{I}, \mathbf{I} \rangle. \\ !\mathsf{check}\ \mathsf{if\_finite}\ \mathsf{zero} = \mathbf{I}. \\ !\mathsf{check}\ \mathsf{if\_finite}\ \mathsf{x} = \mathsf{if\_finite}\ (\mathsf{succ}\ \mathsf{x}). \\ !\mathsf{check}\ \mathsf{if\_finite}\ \omega = \bot. \end{array}$$

We can partially eliminate with Kleene's predecessor function, assuming nonzero input

$$\begin{array}{ll} \mathsf{prec} \;:=\; (\mathsf{nat} \to \mathsf{nat})\ \langle & \\ \quad \lambda(\mathsf{n}, -).\ (\mathsf{succ}\ \mathsf{n},\ \mathsf{n}), & \textit{step} \\ \quad (\mathsf{zero}, \top), & \textit{initial value} \\ \quad \mathbf{F} & \textit{extract rhs of result} \\ \rangle. & \\ !\mathsf{check}\ \mathsf{prec}\ \mathsf{zero} = \top. & \\ !\mathsf{check}\ \mathsf{prec}\ \mathsf{one} = \mathsf{zero}. & \\ !\mathsf{check}\ \mathsf{prec}\ \mathsf{two}\ = \mathsf{one}. & \\ !\mathsf{check}\ \mathsf{prec}\ \omega = \omega. & \\ !\mathsf{check}\ \mathsf{prec} \circ \mathsf{succ} = \mathsf{nat}. & \\ !\mathsf{check}\ \mathsf{succ} \circ \mathsf{prec} = \mathbf{P}\ \mathsf{nat}\ (\lambda \mathsf{n}.\ \mathsf{if}\ (\mathsf{is\_zero}\ \mathsf{n})\ \top). & \\ !\mathsf{check}\ \mathsf{prec} \circ \mathsf{succ} \circ \mathsf{succ}\ \bot = \mathsf{succ}\ \bot. & \end{array}$$

Remark. Note the difference between the predecessor function in Gödel's T and in system F ([GTL89] end of sec. 7.3.2).

Case analysis gives a more complete elimination form.

$$\begin{array}{l} \mathsf{case\_nat} \;:=\; (\mathsf{nat} \to \mathsf{Maybe}\ \mathsf{nat})\ (\lambda \mathsf{n}.\ \mathsf{is\_zero}\ \mathsf{n}\ \mathsf{none}\ (\mathsf{some}.\ \mathsf{prec}\ \mathsf{n})). \\ !\mathsf{check}\ \mathsf{case\_nat} \circ (\mathsf{succ}\ \mathsf{n}) = \mathsf{test\_nat}\ \mathsf{n}\ (\mathsf{some}\ \mathsf{n}). \\ !\mathsf{check}\ \langle \mathsf{zero}, \mathsf{succ} \rangle \circ \mathsf{case\_nat} = \mathsf{nat}. \end{array}$$

Note that Church numerals are particularly poor representatives of counting, since the predecessor or case analysis is so difficult to construct.

Gödel's recursor can be seen as a non-homogeneous time-evolution operator, polymorphic with type (nat→a→a)→nat→a→a,

```
Rec_nat  :=  (∀a:V. Endo. Seq. Endo a) (
    λ− . Yλr, fs, n, x. case_nat n x λn′. r fs∘succ n′ (fs 0 x)
).
rec_nat  :=  Rec_nat ⊥.
```

The recursor is perhaps more natural on streams and coalgebraic numerals (see 3.13).

Remark. Girard considers Gödel's system a step backward from the logical standpoint (see [GTL89] ch. 7), since its terms "do not correspond to proofs in an extended logical system". On the contrary, Gödel's T is a half-step forward, the remaining half being dependent types, where the recursor becomes the induction scheme for second-order Peano arithmetic.

A dependent recursor for second-order dependently typed Peano arithmetic is

```
Ind_nat  :=  (
    ∀a:nat→V. a zero → (∀n:nat. a n → a∘succ n) → ∀n:nat. a n
) (
    λ− . Yλi. λz, s, n. case_nat n z λn′. i (s 0 z) s∘succ n′
).
ind_nat  :=  Ind_nat ⊥.
```

Alternatively, without using the fixed-point combinator, and so typable in system-F,

```
!check Ind_nat = (
    ∀a:nat→V. a zero → (∀n:nat. a n → a∘succ n) → ∀n:nat. a n
) (
    λ− . λz, s, n. n (λ(z, s). (s zero z, s∘succ)) (z, s) K
).
```

We can now define arithmetic operations

```
add  :=  P C (nat → nat → nat) (λm, n, f. (m f)∘(n f)).
!check add zero = nat.                                  zero is an additive unit
!check add one = succ.
!check add two = succ∘succ.
!check add ω = (λn. check_nat n ω).
```

```
mul  :=  P C (nat → nat → nat) (λm, n, f. m(n f)).
!check mul one = nat.                                 one is a multiplicative unit
!check mul zero = (λn. check_nat n zero).
!check mul one = nat.
!check mul ω x = is_zero x zero ω.
```

```
pow  :=  (nat → nat → nat) (λm, n. n m).
!check pow zero = (λn. check_nat zero).
!check pow one = (λn. if_finite n one).
```

```
sub  :=  (λn, m. m prec n).
!check (λn:nat. sub n (succ n)) = (λn. if_finite n ⊤).
```

3.13 Coalgebraic numerals

The type of coalgebraic numerals is μ a. $1+$a, or simply

```
!check num = V (Y Maybe).
test_num := Test num (Y test_Maybe).
```

Theorem 3.13.1. num *is an adequate numeral system.*

Proof. By definition, we need only the following terms:

```
zero_num := num none.
succ_num := (num→num) some.
prec_num := (num→num) (error, I).
!check num n = prec_num∘succ_num n.
!check succ_num n = succ_num∘prec_num∘succ_num n.
```

□

Equality is thus bool-decidable

```
eq_num := (num → bool) (Yλe. ((K, λn. F), λm. (F, λn. e m n))).
if_eq_num := (num → bool) (Yλe. ((I, λn. ⊥), λm. (⊥, λn. e m n))).
!check if_eq_num = if∘eq_num.
```

As mentioned in 3.7, numerals are indices of streams.

```
!check (∀a:V, s:Stream a, n:num. s n:a).
!check (∀a:V. (Stream a)∘num <: num→a).
!check (x, xs) zero_num = x.
!check (x, xs) (succ_num n) = xs n.
```

Indeed primitive recursion with numerals and streams is very simple: given an initial state x and a sequence of state transitions fs, construct the stream of evolving states Rec_num x fs.

```
Rec_num := (∀a:V. a → Stream (a→a) → Stream a) (
    λ− . Yλr, x, (f, fs). (x, r (f x) fs)
).
rec_num := Rec_num ⊥.
```

We will use a similar encoding for recursion in Gödel's T in 6.2.

Numerals are also useful as a uniform type for large finite sets.

```
num_below := (nat → P num) ⟨λs. (⊥, s), ⊤⟩.
nat2num := (nat → num) ⟨some, none⟩.
!check num_below 0 zero_num = ⊤.
!check zero_num : num_below 1.
!check (∀n::test_nat. (num_below n) (nat2num n) = ⊤).
!check (∀n::test_nat. nat2num n : num_below(succ n)).
```

3.14 Dependent types for reduction proofs

Under the Curry-Howard correspondence, we can encode theorems as types, and construct proofs as total inhabitants of the types. More precisely, we can encode a theorem as a closure,test pair $(a, t) \ : \ \exists a{:}\mathbf{V}.\mathsf{Test}\ a$, and prove that the theorem is true by providing a proof $p{:}a{::}t$, i.e. an SKJ term $p{:}a$ for which $t\ p = \mathbf{I}$. As mentionted before, the join operation provides no new theorems under Curry-Howard, but it does provide a mechanism for ambiguity in proofs, e.g., a convenient without-loss-of-generality construct. But the join operation offers a much more powerful tool, a semantic equivalent of type-inference.

To illustrate how type-inference of proof sketches works, let us consider a simple example, say a theorem that there is no greatest natural number

$$\mathsf{thm} \ := \ \forall n{:}\mathsf{nat}.\ \exists m{:}\mathsf{nat}.\ \mathsf{assert\_greater}\ m\ n.\ \mathsf{unit}$$

We can test a proof candidate by checking whether

$$\forall n{::}\mathsf{test\_nat}. \quad \mathbf{let}\ (-, u) := p\ n. \quad u{::}\mathsf{unit}$$

An obvious proof would be to provide the successor function

$$\mathsf{pf} \ := \ \mathsf{thm}\ (\lambda n{:}\mathsf{nat}.\ (\mathsf{succ}\ m,\ \mathsf{assert\_greater}\ m\ n\ \mathbf{I}))$$

but since assert_greater is already built into thm, we can achieve the same result by simply sketching

$$\mathsf{pf}' \ := \ \mathsf{thm}\ (\lambda n.\ (\mathsf{succ}\ n,\ \bot))$$

where the bottom element $\bot$ is a formalization of ellipses in human proofs.

For our main example of real theorems, we will prove properties of the system SKJ itself, working with β-reduction, convergence, and later in 6.1 the Scott ordering. The two-category of β-reductions has objects SKJ -terms, morphisms reductions between terms, and two-cells sets of reductions between terms. We need to deal with two-categories since composition of reduction sequences is only associative up to equivalence. Identifying all reductions between terms yields a true category, in fact a preorder. The skeleton of this preorder is a poset, in fact the lattice SKJ .

Question 3.14.1. *What is the higher homology of this higer-category? Eg, how do transformations between reductions look?*

A type for SKJ terms

Our indices for reduction proofs will be terms in the language of untyped SKJ -terms with an explicit $\top$

$$\frac{x\ \mathsf{term} \qquad y\ \mathsf{term}}{x\ y\ \mathsf{term}}\,(\mathsf{ap}) \qquad \frac{}{\mathbf{S}\ \mathsf{term}}\,(\underline{\mathbf{S}}) \qquad \frac{}{\mathbf{K}\ \mathsf{term}}\,(\underline{\mathbf{K}}) \qquad \frac{}{\mathbf{J}\ \mathsf{term}}\,(\underline{\mathbf{J}}) \qquad \frac{}{\top\ \mathsf{term}}\,(\underline{\top})$$

For these SKJ terms, we define a type, test, and check

```
pre_term := V (Yλa. Sum (Prod a a). 2 Maybe bool).
check_term := Check pre_term (
    Yλc. check_Sum (check_Prod c c). 2 check_Maybe check_bool
).
!define term := Checked pre_term check_term.
!check check_term : Check term.
```

```
!define test_term := Test term (
    Yλt. test_Sum (test_Prod t t). 2 test_Maybe test_bool
).
```

where 2 Maybe bool has three values, for $\mathbf{S}, \mathbf{K}, \mathbf{J}, \top$. The atomic introduction forms are thus

```
$\underline{\mathbf{S}}$ := term (inr none).
$\underline{\mathbf{K}}$ := term (inr. some none).
$\underline{\mathbf{J}}$ := term (inr. 2 some true).
$\underline{\top}$ := term (inr. 2 some false).
!check ( $\underline{\mathbf{S}}, \underline{\mathbf{K}}, \underline{\mathbf{J}}, \underline{\top}$ :: test_term ).
```

For convenience, we define two compound introduction forms

```
ap := (term $\to$ term $\to$ term) ($\lambda$x, y. inl (x, y)).
join := (term $\to$ term $\to$ term) ($\lambda$x, y. ap(ap $\underline{\mathbf{J}}$ x)y).
!check  test_term (ap x y) = and_semi (test_term x) (test_term y).
!check  test_term (join x y) = and_semi (test_term x) (test_term y).
```

One form of elimination is by evaluating a term.

```
eval_term := (term $\to$ any) (
    $\lambda$x::test_term. $\langle$x$\rangle$.
    $\mathbf{Y}\lambda$e. ($\lambda$(x, y). (e x)(e y), $\mathbf{S}, \mathbf{K}, \mathbf{J}, \top$)
).
!check eval_term $\underline{\mathbf{S}} = \mathbf{S}$.
!check eval_term $\underline{\mathbf{K}} = \mathbf{K}$.
!check eval_term $\underline{\mathbf{J}} = \mathbf{J}$.
!check eval_term $\underline{\top} = \top$.
!check eval_term (ap x y)
     = and_semi (test_term x) (test_term y) (eval_term x) (eval_term y).
!check eval_term (join x y)
     = and_semi (test_term x) (test_term y) (eval_term x | eval_term y).
```

We can also discriminate total terms, with truth values either bool, semi, or unit

```
eq_term := $\mathbf{P}$ $\mathbf{C}$ (term $\to$ term $\to$ bool) (
    $\lambda$x, y. and_semi (test_term x) (test_term y) $\langle$x, y$\rangle$.
    $\mathbf{Y}\lambda$e. (
        $\lambda$(l, r). ($\lambda$(l', r'). and(e l l')(e r r'), $\mathbf{F}, \mathbf{F}, \mathbf{F}, \mathbf{F}$),
        ($\lambda - .\mathbf{F}$, $\mathbf{K}, \mathbf{F}, \mathbf{F}, \mathbf{F}$),
        ($\lambda - .\mathbf{F}$, $\mathbf{F}, \mathbf{K}, \mathbf{F}, \mathbf{F}$),
        ($\lambda - .\mathbf{F}$, $\mathbf{F}, \mathbf{F}, \mathbf{K}, \mathbf{F}$),
        ($\lambda - .\mathbf{F}$, $\mathbf{F}, \mathbf{F}, \mathbf{F}, \mathbf{K}$)
    )
).
!check ( $\underline{\mathbf{S}}, \underline{\mathbf{K}}, \underline{\mathbf{J}}, \underline{\top}$ :: test_bool$\circ$($\mathbf{W}$ eq_term) ).
!check $\mathbf{W}$ if_eq_term (ap x y) = and_semi (test_term x) (test_term y) true.
!check eq_term x x = test_term x true.
```

```
assert_eq_term := P C (term → term → unit) (
    λx, y. (check_term x | check_term y) ⟨x, y⟩.
    Yλe. (
        λ(l, r). (λ(l′, r′). e l l′ | e r r′, ⊤, ⊤, ⊤, ⊤),
        (⊤, I, ⊤, ⊤, ⊤),
        (⊤, ⊤, I, ⊤, ⊤),
        (⊤, ⊤, ⊤, I, ⊤),
        (⊤, ⊤, ⊤, ⊤, I)
    )
).
!check ( S̲, K̲, J̲, ⊤̲ :: W assert_eq_term ).
!check W assert_eq_term (ap x y) = and_unit (check_term x) (check_term y).
!check assert_eq_term x x = check_term x.
!check assert_eq_term x y = assert(eq_term x y).
```

```
if_eq_term := P C (term → term → semi) (
    λx, y. and_semi (test_term x) (test_term y) ⟨x, y⟩.
    Yλe. (
        λ(l, r). (λ(l′, r′). and_semi (e l l′) (e r r′), ⊥, ⊥, ⊥, ⊥),
        (⊥, I, ⊥, ⊥, ⊥),
        (⊥, ⊥, I, ⊥, ⊥),
        (⊥, ⊥, ⊥, I, ⊥),
        (⊥, ⊥, ⊥, ⊥, I)
    )
).
!check ( S̲, K̲, J̲, ⊤̲ :: W if_eq_term ).
!check W if_eq_term (ap x y) = and_semi (test_term x) (test_term y).
!check if_eq_term x x = test_term x.
!check if_eq_term x y = if(eq_term x y).
```

As with all our algebraic datatypes, these terms are their own basic elimination forms. However, we will often need compound elimination forms for various shapes of terms, e.g., $\mathbf{S}$ x y z and $\mathbf{K}$ x y. For these we define various partial elimination forms, each either asserting, or testing that the particular case holds. For example, when handling the case of applications ap x y, we eliminate with one of

```
case_ap := (term → (term→term→any) → any) (I, ⊤).
if_case_ap := (term → (term→term→any) → any) (I, ⊥).
```

If we are in the right case, we can get the terms x, y back.

```
!check (∀x:term::test_term, y:term::test_term.
    x = (case_ap (ap x y) λu, −. u) = (if_case_ap (ap x y) λu, −. u)
).
!check (∀x:term::test_term, y:term::test_term.
    y = (case_ap (ap x y) λ−, u. u) = (if_case_ap (ap x y) λ−, u. u)
).
```

But in the wrong case (here, an atom) the elim forms either err of diverge.

```
!check ⊤ = case_ap S̲ = case_ap K̲ = case_ap J̲ = case_ap ⊤̲.
!check ⊥ = if_case_ap S̲ = if_case_ap K̲ = if_case_ap J̲ = if_case_ap ⊤̲.
```

We also need elim forms for redexes $\mathbf{S}$ x y z, $\mathbf{K}$ x y, $\mathbf{J}$ x y, and $\top$ x.

```
case_S  :=  (term → (term → term → term → any) → any) (
     λsxyz. case_ap sxyz λsxy, z.
              case_ap sxy λsx, y.
              case_ap sx λs, x.
              assert_eq_term S s. ⟨x, y, z⟩
).
if_case_S  :=  (term → (term → term → term → any) → any) (
     λsxyz.  if_case_ap sxyz λsxy, z.
               if_case_ap sxy λsx, y.
               if_case_ap sx λs, x.
               if_eq_term S s.          ⟨x, y, z⟩
).
!check (∀x : term, y : term, z : term.   Sxyz := ap(ap(ap S x)y)z.
     x  =  (case_S Sxyz λu, −, −. u)  =  (if_case_S Sxyz λu, −, −. u)   AND
     y  =  (case_S Sxyz λ−, u, −. u)  =  (if_case_S Sxyz λ−, u, −. u)   AND
     z  =  (case_S Sxyz λ−, −, u. u)  =  (if_case_S Sxyz λ−, −, u. u)
).
```

```
case_K  :=  (term → (term → term → any) → any) (
     λkxy. case_ap kxy λkx, y.
             case_ap kx λk, x.
             assert_eq_term K k. ⟨x, y⟩
).
if_case_K  :=  (term → (term → term → any) → any) (
     λkxy. if_case_ap kxy λkx, y.
             if_case_ap kx λk, x.
             if_eq_term K k.      ⟨x, y⟩
).
!check (∀x : term, y : term.   Kxy := ap(ap K x)y.
     x  =  (case_K Kxy λu, −. u)  =  (if_case_K Kxy λu, −. u)   AND
     y  =  (case_K Kxy λ−, u. u)  =  (if_case_K Kxy λ−, u. u)
).
```

```
case_J  :=  (term → (term → term → any) → any) (
     λjxy. case_ap jxy λjx, y.
             case_ap jx λj, x.
             assert_eq_term J j. ⟨x, y⟩
).
if_case_J  :=  (term → (term → term → any) → any) (
     λjxy. if_case_ap jxy λjx, y.
             if_case_ap jx λj, x.
             if_eq_term J j.       ⟨x, y⟩
).
!check (∀x : term, y : term.   Jxy := ap(ap J x)y.
     x  =  (case_J Jxy λu, −. u)  =  (if_case_J Jxy λu, −. u)   AND
     y  =  (case_J Jxy λ−, u. u)  =  (if_case_J Jxy λ−, u. u)
).
```

```
case⊤  :=  (term → (term→any) → any) (
    λtx. case_ap tx λt, x.
         assert_eq_term ⊤ t. ⟨x⟩
).
if_case⊤  :=  (term → (term→any) → any) (
    λtx. if_case_ap tx λt, x.
         if_eq_term ⊤ t. ⟨x⟩
).
!check (∀x:term.  Tx := ap ⊤ x.
    x = (case⊤ Tx λu. u) = (if_case⊤ Tx λu. u)
).
```

Finally, we will often need to join over all total terms.

```
Join_term  :=  Sset term (Yλs. (⟨S⟩ | ⟨K⟩ | ⟨J⟩ | ⟨⊤⟩  |  sλx. sλy. ⟨ap x y⟩)).
!check Join_term test_term = I.
```

A dependent type for reduction paths

Next we define a type for reduction paths, dependent on the start and end terms. Our reduction paths will capture the following inference rules, notably omitting reflexivity

$$\frac{x \twoheadrightarrow y \qquad y \twoheadrightarrow z}{x \twoheadrightarrow z}\,(\text{trans}) \qquad \frac{f \twoheadrightarrow g \qquad x{:}\text{term}}{f\ x \twoheadrightarrow g\ x}\,(\text{lhs}) \qquad \frac{f{:}\text{term} \qquad x \twoheadrightarrow y}{f\ x \twoheadrightarrow f\ y}\,(\text{rhs}) \qquad \frac{x\ \text{term}}{\top\ x \twoheadrightarrow \top}\,(\top)$$

$$\frac{x, y, z{:}\text{term}}{\mathbf{S}\ x\ y\ z \twoheadrightarrow x\ z(y\ z)}\,(\mathbf{S}) \qquad \frac{x, y{:}\text{term}}{\mathbf{K}\ x\ y \twoheadrightarrow x}\,(\mathbf{K}) \qquad \frac{x, y{:}\text{term}}{\mathbf{J}\ x\ y \twoheadrightarrow x}\,(\text{J1}) \qquad \frac{x, y{:}\text{term}}{\mathbf{J}\ x\ y \twoheadrightarrow y}\,(\text{J2})$$

Now the type of reductions depends on the equality of some terms in the hypotheses above. Eg, transitivity requires the y in $x \twoheadrightarrow y$ and $y \twoheadrightarrow z$ to agree. Recalling our notation of Or = Sum and And = Prod, we define a type of pre-proofs, possibly containing errors

```
pre_Red  :=  (term → term → V) (
    Yλr, u, v.
    Or (∃x:term. And (r u x) (r x v)).                transitivity
    Or (case_ap uλf, x. case_ap vλf′, x′.
        Or (r f f′) (r x x′)).                        left and right monotonicity
    3 Maybe bool                                      5 atomic reductions
).
test_pre_Red  :=  (∀u:term, v:term. Test. pre_Red u v) (
    λ−, −. Yλt.
    test_Sum (test_Prod test_term. test_Prod t t).
    test_Sum (test_Sum t t).
    3 test_Maybe test_bool
).
```

Next we check the pre-proofs for errors, raising the entire proof to $\top$ if any errors are found.

```
check_Red := (∀u:term, v:term. Check. pre_Red u v) (
    eq := assert_eq_term.
    Yλc, u, v. (check_term u | check_term v) (
        λ(x, r, r′). c u x r | c x v r′,
        case_ap uλf, x. case_ap vλf′, x′. ( eq x x′. c f f′ ,
                                            eq f f′. c x x′ ),
        case⊤ uλx. eq v ⊤̲,
        case_S uλx, y, z. eq v (ap (ap x z) (ap y z)),
        case_K uλx, y. eq v y,
        case_J uλx, y. (eq v x,  eq v y)
    )
).
```

```
!define Red := (∀x:term, y:term. Checked (pre_Red x y). check_Red x y).
!check check_Red : (∀x:term, y:term. Check. Red x y).
```

Once check_Red has eliminated possible errors, totality testing is much easier, and can in fact ignore its parameters.

```
test_Red := (∀x:term,  y:term. Test. Red x y) test_pre_Red.
```

Reduction proofs have introduction forms for each inference rule above.

```
Trans := (∀x, y, z. Red x y → Red y z → Red x z) (
    λ−, y, −, xy, yz. inl (y, xy, yz)
).
Lhs_r := (∀f, f′, x. Red f f′ → Red (ap f x) (ap f′ x)) (
    λ−, −, −. inr∘inl∘inl
).
Rhs_r := (∀f, x, x′. Red x x′ → Red (ap f x) (ap f x′)) (
    λ−, −, −. inr∘inl∘inr
).
⊤_R := (∀x. Red (ap ⊤̲ x) ⊤̲) (λ − . 2 inr none).
S_R := (∀x, y, z. Red (ap(ap(ap S̲ x)y)z) (ap(ap x z)(ap y z))) (
    λ−, −, −. 3 inr none
).
K_R := (∀x, y. Red (ap(ap K̲ x)y) x) (λ−, −. 4 inr none).
J_R1 := (∀x, y. Red (join x y) x) (λ−, −. 5 inr true).
J_R2 := (∀x, y. Red (join x y) y) (λ−, −. 5 inr false).
```

Now these introduction forms are dependent on the terms they deal with. The join operation allows us to ignore these parameters in proof bodies, simply sketching the rules we use. Wrapping a proof sketch in the theorem-as-closure it proves will raise the proof sketch up to a total well-indexed inhabitant. Thus we can use the abbreviated introduction forms

```
trans_r := Trans ⊥ ⊥ ⊥.
lhs_r := Lhs_r ⊥ ⊥ ⊥.
rhs_r := Rhs_r ⊥ ⊥ ⊥.
⊤_r := ⊤_R ⊥.
S_r := S_R ⊥ ⊥ ⊥.
K_r := K_R ⊥ ⊥.
J_r1 := J_R1 ⊥ ⊥.
J_r2 := J_R2 ⊥ ⊥.
```

The case study in 6.1 gives many examples of how to use these proof sketches.
We can also search for reduction proofs by joining

```
Join_Red := (∀x:term, y:term. Sset. Red x y) (
    eq := if_eq_term.
    Yλj, u, v. ( Join_term λx. j u xλr. j x vλr′. ⟨$\mathrm{trans}_r$ r r′⟩ )
            | ( if_case_ap u λf, x.
                if_case_ap v λf′, x′. eq x x′ (j f f′λr. ⟨$\mathrm{lhs}_r$ r⟩)
                                   | eq f f′ (j x x′λr. ⟨$\mathrm{rhs}_r$ r⟩) )
            | ( $\mathrm{if\_case}_\top$ u λ− . eq v $\underline{\top}$ ⟨$\top_r$⟩ )
            | ( $\mathrm{if\_case}_{\mathbf{S}}$ u λx, y, z. eq v (ap (ap x z) (ap y z)) ⟨$\mathbf{S}_r$⟩ )
            | ( $\mathrm{if\_case}_{\mathbf{K}}$ u λx, y. eq v x ⟨$\mathbf{K}_r$⟩ )
            | ( $\mathrm{if\_case}_{\mathbf{J}}$ u λx, y. eq v x ⟨$\mathbf{J}_{r1}$⟩
                                   | eq v y ⟨$\mathbf{J}_{r2}$⟩ )
).
!check (∀x::test_term, y::test_term. Join_Red x y (check_Red x y) = I).
```

This allows us to semidecide reduction between any two terms:

```
if_Red := (term → term → semi) (λx, y. Join_Red x y (test_pre_Red x y)).
```

Dependent types for convergence

Convergence is much easier to prove than general reduction. In $\mathcal{H}^*$, a term converges iff there is a reduction sequence to $\top$, after feeding the term some number of arguments of $\top$. Thus we formalize this idea as the two inference rules

$$\frac{\mathsf{x}\ \twoheadrightarrow\ \top}{\mathsf{x}\ \mathsf{Conv}}\ (\text{done}) \qquad\qquad \frac{\mathsf{x}\ \top\ \mathsf{Conv}}{\mathsf{x}\ \mathsf{Conv}}\ (\text{next})$$

As with the type of reductions Red, we define the type of convergence proofs by first defining a type of possibly-erroneous pre-proofs (conv), then defining a proof-checker (check_Conv), and finally restricting the pre-proofs to checked proofs (Conv).

```
conv := V (Yλa. Or a. Red ⊥ $\underline{\top}$).
test_conv := Test conv (Yλt. λx. test_Sum t (test_pre_Red ⊥ $\underline{\top}$)).
```

```
check_Conv := (term → Check conv) (
    Yλc, x. check_term x. check_Sum (c (ap x $\underline{\top}$)) (check_Red x $\underline{\top}$)
).
Conv := (∀x:term. Checked conv. check_Conv x).
!check check_Conv x : Check (Conv x).
```

```
test_Conv := (∀x:term. Test. Conv x) (λ− . test_conv).
```

Convergence proofs have the total introduction forms

```
Next := (∀x. if (Conv. ap x $\underline{\top}$). Conv x) (λ− . inr).
Done := (∀x. if (Red x $\underline{\top}$). Conv x) (λ− . inl).
```

and partial proof sketches

```
next := Next ⊥.
done := Done ⊥.
```

For example each atom in the basis provably converges.

```
!check $3$ next (done $\mathbf{S}_r$) :: test_Conv $\underline{\mathbf{S}}$.
!check $2$ next (done $\mathbf{K}_r$) :: test_Conv $\underline{\mathbf{K}}$.
!check $2$ next (done $\mathbf{J}_{r1}$) :: test_Conv $\underline{\mathbf{S}}$.
!check next (done $\top_r$) :: test_Conv $\underline{\top}$.
```

We can also semidecide convergence of terms by joining over all total proofs

```
Join_Conv := ($\forall$x:term. Sset. Conv x) (
    $\mathbf{Y}\lambda$j, x. (Join_Red x $\underline{\top}$ $\lambda$r. $\langle$done r$\rangle$)
          | (j (ap x $\underline{\top}$) $\lambda$c. $\langle$next c$\rangle$)
).
!check ($\forall$x::test_term. Join_Conv x check_conv = $\mathbf{I}$).
```

and testing whether any of them match a term in question

```
if_conv_term := (term $\to$ semi) ($\lambda$x. Join_Conv x test_conv).
!check ( $\underline{\mathbf{S}}, \underline{\mathbf{K}}, \underline{\mathbf{J}}, \underline{\top}$ :: if_conv_term ).
!check ($\forall$x, y. ap x y :: if_conv_term $\implies$ x :: if_conv_term).
```

Chapter 4

Untyped λ-calculus for convex sets of probability distributions (SKRJ)

In this chapter we examine a λ-calculus SKRJ with uncertainty, represented as convex sets of probability distributions (CSPDs), or lower-sets of probability valuations. Our design goal for this language is a definability of simple types theorem, in the manner of 3.6. At the time of the writing this thesis we have not achieved this goal; however, the attempt has shed light on the simpler language SKJ , in two respects.

First, join has traditionally been interpreted as nondeterminism or concurrency ([DCHA00],[DCL02]); however concurrency distributes over randomness, preventing definable types-as-closures. Our solution is to instead let randomness distributes over join, naturally leading to convex sets of probability distributions. In this semantics, join is seen to model *ambiguity* or indeterminacy rather than concurrency.

Second, just as the Simple type constructor in SKJ allows ambiguous inhabitants, any possible constructor definable in SKRJ would allow fuzzy inhabitants (where fuzzy means ambiguously random, as in CSPDs). In SKJ we were able to filter these out with *disambiguation* tricks (in 3.7), but these tricks generally fail in SKRJ . Forced to accommodate the fuzzy inhabitants of a possible Simple type constructor, we develop a sampling monad for arbitrary simple types (similar to [PPT05]). Although the monadic types are more difficult to hack with, they obey better categorical properties than our easier-to-work-with types from 3.7.

Our motivation for working with probabilistic languages comes from research in Bayesian networks and their generalizations. Koller et al [KMP97] describe methods for learning parameters of stochastic programs. Pfenning et al [PPT05] implement a probabilistic language λ_O by extending OCaml with a monadic probability type. Although this language does not admit a types-as-closures interpretation in SKRJ , their probability monad can be interpreted unchanged to represent convex sets of probability distributions.

Semantics of probabilistic programs has a long history, beginning in the early 1980s. Kozen [Koz81] describes a linear probabilistic semantics, but his stochastic programs cannot take random generators as parameters. That is, pmfs can only be input via their *parameters*. Plotikin [Plo82] and Plotkin and Jones [JP89] develop a probabilistic powerdomain as a domain-theoretic model of ambiguity. Jones analyses the interaction of probability and non-determinism in her thesis [Jon89]. Heckmann [Hec94] develops the probabilistic powerdomain in terms of Vickers' information systems. Edalat [Eda95] develops domain theory for stochastic processes. Jung [JT98] discusses problems relating the probabilistic powerdomain and function space operators. Desharnais and Jagadeesan [DGJP04] discuss metric structure on probabilistic spaces, where equality is too fine for many practical purposes.

Convex sets of probability distributions (CSPDs), or credal sets, are widely studied in the literature on imprecise probabilities, for example in the bi-annual International Symposium on Imprecise Probability: Theory and Applications (ISIPTA). Doan et al [DVH98] provide geometric foundations for CSPDs. Cano and Moral [CM99] survey belief propagation algorithms for CSPDs.

4.1 Axioms for randomness

| !using $\mathbf{R}$.

In this section we show that joins of mixtures generalize convex sets of probability valuations. Hence SKRJ provides a nice denotational semantics for programming languages with monadic types of imprecise probabilities. First let us introduce a syntax with infinitary joins and mixtures.

Definition 4.1.1. A *join* is a J-term (closure under J), defined by the language

$$\frac{\mathsf{x\ term}}{\mathsf{x\ join}}\ \text{(singleton)} \qquad \frac{\mathsf{x\ join} \quad \mathsf{y\ join}}{\mathsf{x} \mid \mathsf{y\ join}}\ \text{(binary)} \qquad \frac{\mathsf{X\ sset}}{\bigsqcup \mathsf{X\ join}}\ \text{(infinitary)}$$

A *mixture* is an R-pmf (closure under R), defined by the language

$$\frac{\mathsf{x\ term}}{\mathsf{x\ mix}}\ \text{(singleton)} \qquad \frac{\mathsf{x\ mix} \quad \mathsf{y\ mix}}{\mathsf{x}+\mathsf{y\ mix}}\ \text{(binary)} \qquad \frac{\mathsf{X\ pmf(mix)}}{\mathbb{E}[\mathsf{X}]\ \mathsf{mix}}\ \text{(infinitary)}$$

A *(JR-)slurry* is a join-mixture (closed under J,R).

Notation 4.1.2. We write $\mathbf{R}\ \mathsf{x}\ \mathsf{y} = \mathsf{x}+\mathsf{y}$ for the random choice operation. Beware that $+$ is commutative but not associative: we never write $\mathsf{x}+\mathsf{y}+\mathsf{z}$. We let $+$ have higher precedence than join, so that $\mathsf{x}+\mathsf{y} \mid \mathsf{z} = (\mathsf{x}+\mathsf{y}) \mid \mathsf{z}$.

Our semantics for randomness will be defined in terms of traces.

Definition 4.1.3. A *trace* is a sequence[1] $\langle \mathsf{M}_1, \ldots, \mathsf{M}_\mathsf{n} \rangle$ of n terms, for some $\mathsf{n} \geq 0$. An SKJ *-trace* is a sequence of terms from SKJ; and similarly for other fragments.

In SK and SKJ, convergence under traces provide complete information about any term.

Theorem 4.1.4. *Let* x, y *be* SK *or* SKJ *terms. Then* $\mathsf{x} \sqsubseteq \mathsf{y}$ *iff for every trace* $\mathsf{t} = \langle \mathsf{M}_1, \ldots, \mathsf{M}_\mathsf{n} \rangle$,
$\mathsf{t\ x\ conv} \implies \mathsf{t\ y\ conv}$.

In a system with randomness, convergence is probabilistic so that a given term x will converge only *with some probabilty* p. This motivates the extension of $\mathcal{H}^*$ from SK and SKJ to SKRJ:

Definition 4.1.5. In *trace probability* semantics $\mathcal{H}^*$ of the language SKRJ, the order relation $\mathsf{x} \sqsubseteq \mathsf{y}$ between terms x, y holds iff under every SKJ-trace t, y is at least as likely to converge as x, i.e., $\mathbb{P}(\mathsf{t\ x\ conv}) \leq \mathbb{P}(\mathsf{t\ y\ conv})$.

Our main theorem of this section is that mixtures distribute over joins,

Theorem 4.1.6. *(distributivity)* $(\mathsf{x} \mid \mathsf{y}) + \mathsf{z} = \mathsf{x}+\mathsf{z} \mid \mathsf{y}+\mathsf{z}$.

Proof. Consider trace probability semantics. At each trace t, we have convergence $\mathsf{c}(-) := \mathbb{P}(\mathsf{t}\ -\ \mathsf{conv})$

$$\begin{aligned} \mathsf{c}((\mathsf{x} \mid \mathsf{y}) + \mathsf{z}) &= (\max(\mathsf{c}(\mathsf{x}), \mathsf{c}(\mathsf{y})) \ + \ \mathsf{c}(\mathsf{z}))/2 \\ &= \max((\mathsf{c}(\mathsf{x}) + \mathsf{c}(\mathsf{z}))/2, \ (\mathsf{c}(\mathsf{y}) + \mathsf{c}(\mathsf{z}))/2) \\ &= \mathsf{c}(\mathsf{x}+\mathsf{z} \mid \mathsf{y}+\mathsf{z}) \end{aligned}$$

□

Naively slurries can be arbitrarily deep, as in $\mathsf{w} + (\mathsf{x} \mid (\mathsf{y}+\mathsf{z}))$, however, distributivity gives us a notion of join normal form.

[1] Recall from 1.5 that $\langle \mathsf{M}_1, \ldots, \mathsf{M}_\mathsf{n} \rangle = \lambda \mathsf{f}.\ \mathsf{f\ M}_1\ \ldots\ \mathsf{M}_\mathsf{n}$, so that we can treat sequences as argument lists.

Corollary 4.1.7. *(JR-normal form) Every slurry can be written as a join of mixtures.*

In contrast to join-as-concurrency, our join-as-ambiguity does not distribute over mixtures (**J** does not distribute over **R**). This means we have no **R**-normal form, and an operational semantics for random sampling fails.

Distributivity gives us a clean notion of Böhm trees in SKRJ.

Definition 4.1.8. A *JR-Böhm tree* (JR-BT) is the SKRJ notion of Böhm tree, defined by limits in the language

$$\frac{\mathsf{x}\ \ \mathsf{var}}{\mathsf{x}\ \ \mathsf{BT}}\,(\mathsf{var}) \qquad\qquad \frac{\underline{\mathsf{x}},\mathsf{h}\ \ \mathsf{var} \qquad \underline{\mathsf{M}}\ \ \mathsf{slurry}(\mathsf{BT})}{\lambda\underline{\mathsf{x}}.\ \mathsf{h}\ \underline{\mathsf{M}}\ \ \mathsf{BT}}\,(\mathsf{abs}-\mathsf{app})$$

A *finite* BT consists of only finite slurries and finitely many applications of rule abs-app.

Theorem 4.1.9. *Every* SKRJ *term is a join of mixtures of JR-BTs (under $\mathcal{H}^*$).*

Proof. By straight-forward extension of SKJ -theorem. □

Finite approximation also carries over, using the following

Lemma 4.1.10. *Let* X *be a probability distribution over* SKRJ *terms. Then the mixture* $\mathbb{E}[\mathsf{X}]$ *is a directed join of finite mixtures of* SKRJ *terms.*

Proof. Note that X must have countable support, since SKRJ is countable. Let $\mathsf{X_n}$ be the approximation of X where probabilities are rounded down to n-bit approximations, with the remaining mass given to $\bot$. Then $\mathbb{E}[\mathsf{X}] = \bigsqcup\limits_{\mathsf{n}>0} \mathsf{X_n}$. □

Finally, we give a semantics as convex sets of probability distributions, where join is interpreted as the *convex hull* operation, and mixture is interpreted as the *convex combination* operation.

Definition 4.1.11. (CSPD semantics) Let the *CSPD interpretation* of a term x be the set

$$[\mathsf{x}] \ := \ \{(\mathsf{t}, [0, \mathbb{P}(\mathsf{t}\ \mathsf{x}\ \mathsf{conv})) \ \mid\ \mathsf{t}\ \text{a}\ \mathsf{trace}\}$$

of trace,probability pairs such that $(\mathsf{t},\mathsf{p}) \in \mathsf{x}$ iff $\mathbb{P}(\mathsf{t}\ \mathsf{x}\ \mathsf{conv}) < \mathsf{p}$.

Theorem 4.1.12. *Joins interpret as convex hulls*

$$[\mathsf{x} \mid \mathsf{y}] = \langle [\mathsf{x}], [\mathsf{y}] \rangle := \{(t, [\mathsf{x}]t \cap [\mathsf{y}]t) | t \text{ a trace}\}$$

and mixtures interpret as convex combinations

$$[\mathsf{x}+\mathsf{y}] = \frac{[\mathsf{x}]+[\mathsf{y}]}{2} := \left\{\left(t, \frac{[\mathsf{x}]t+[\mathsf{y}]t}{2}\right) \mid t \text{ a trace}\right\}$$

Proof. By trace probability semantics, joins are maxima, and mixtures are averages. □

Distributivity has a nice visual interpretation in this semantics, shown in Figure 4.1. We will use the following lemma to characterize types-as-closures in 4.3.

Corollary 4.1.13. *Let* $\mathsf{x} = \bigsqcup\limits_{\alpha} \mathbb{E}[\mathsf{X}_\alpha]$ *be the JR-normal form of* x*, where each* X *is a head normal form. Then* x *can be interpreted as the convex hull of points* X_α *in the simplex of probability distributions over* $\bigcup\limits_{\alpha} \mathsf{support}(\mathsf{X}_\alpha)$.

Note that among types with a totality semipredicate, we can think of $\bot$ as a non-value and instead consider convex sets of *subprobability* distributions.

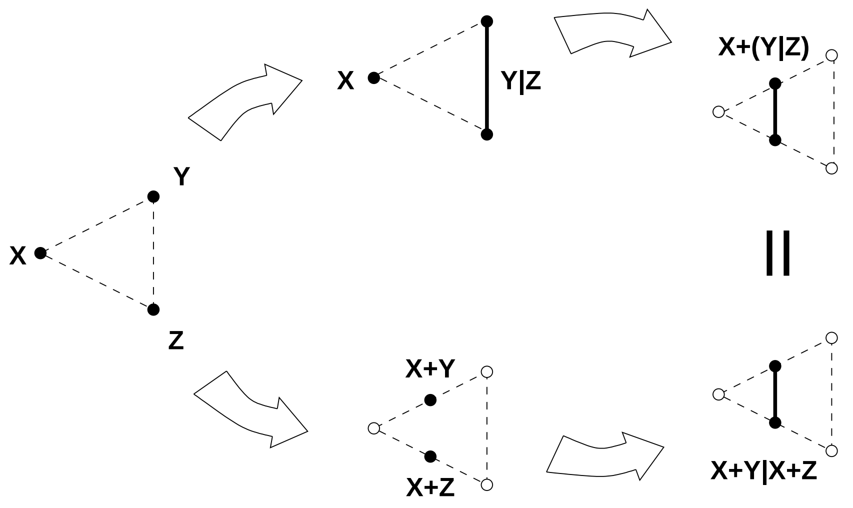

Figure 4.1: Distributivity of convex-combination + over convex-hull |.

Axioms and axiom schemata

The following axiom shemata are enforced for the atom $\mathbf{R}$:

$$\frac{}{\mathsf{x}+\mathsf{x}=\mathsf{x}}\ (\mathsf{idem}) \qquad \frac{}{\mathsf{x}+\mathsf{y}=\mathsf{y}+\mathsf{x}}\ (\mathsf{comm}) \qquad \frac{}{(\mathsf{x}+\mathsf{y})\mathsf{z}=\mathsf{x}\ \ \mathsf{z}+\mathsf{y}\ \ \mathsf{z}}\ (\mathsf{distrib}-\mathbf{R})$$

$$\frac{\mathsf{x}\sqsubseteq\mathsf{z} \qquad \mathsf{y}\sqsubseteq\mathsf{z}}{\mathsf{x}+\mathsf{y}\sqsubseteq\mathsf{z}}\ (\mathsf{subconvex}) \qquad \frac{\mathsf{x}\sqsubseteq\mathsf{y} \qquad \mathsf{x}\sqsubseteq\mathsf{z}}{\mathsf{x}\sqsubseteq\mathsf{y}+\mathsf{z}}\ (\mathsf{supconvex})$$

We begin with negative order axioms (the first since 2.1)

| !assume $\mathbf{K}+\mathbf{F}=\mathbf{R}$.
| !assume $(\ \mathbf{K},\mathbf{F}\ \ \not\sqsubseteq\ \ \mathbf{R}\ )$.
| !assume $(\ \mathbf{K},\mathbf{F}\ \ \not\sqsupseteq\ \ \mathbf{R}\ )$.
| !assume $\bot\ \sqsubsetneq\ \mathbf{R}\ \bot\ \sqsubsetneq\ \mathbf{I}\ \sqsubsetneq\ \mathbf{R}\ \top\ \sqsubsetneq\ \top$.
| !check $\mathbf{R}\ :\ \Phi$.
| !check $\mathbf{R}\ :\ \langle\mathbf{K},\mathbf{F}\rangle$.

algebraic properties

| !assume $\mathsf{x}+\mathsf{x}=\mathsf{x}$. *idempotence*
| !assume $(\mathsf{x}+\mathsf{y})\mathsf{z}=\mathsf{x}\ \ \mathsf{z}+\mathsf{y}\ \ \mathsf{z}$. *distributivity of right-application*
| !assume $\mathbf{B}(\mathsf{x}+\mathsf{y})=\mathbf{B}\ \ \mathsf{x}+\mathbf{B}\ \ \mathsf{y}$. *distributivity of right-composition*

and shallow and deep commutativity.

| !assume $\mathsf{x}+\mathsf{y}=\mathsf{y}+\mathsf{x}$.
| !assume $(\forall\mathsf{x}.\ (\mathsf{a}+\mathsf{x})+(\mathsf{b}+\mathsf{y})=(\mathsf{b}+\mathsf{x})+(\mathsf{a}+\mathsf{y}))$.
| !assume $(\forall\mathsf{y}.\ (\mathsf{a}+\mathsf{x})+(\mathsf{b}+\mathsf{y})=(\mathsf{b}+\mathsf{x})+(\mathsf{a}+\mathsf{y}))$.

Note that we manually abstract the x and y in the last two, as the cost of closing over four variables is very high (see 2.2).

We can express countable additivity as

| !assume $\mathbf{Y}(\mathbf{R}\ \ \mathsf{x})=\mathsf{x}$.

Corollary 4.1.14. $\mathbf{R}$ *is injective, with left-inverse* $\mathbf{Y}$.

```
!check  Y∘R = I.
```

The basic combinators left-distribute over mixtures

```
!assume  K(x + y) = K x + K y.
!assume  F(x + y) = F x + F y.
!assume  C(x + y) = C x + C y.
!assume  B(x + y) = B x + B y.
!assume  W(x + y) = W x + W y.
!assume  S(x + y) = S x + S y.
```

Mixture distributes over join

```
!assume  x + (y | z) = x + y | x + z.
!assume  (x | y) + z = x + z | y + z.
```

but not conversely

```
!check  J(x + y)  ≠  J x + J y.
```

Some care must be taken when joining $\mathbf{R}$-pmfs, for example one might hope to join two sketches $\mathbf{K} + \perp \mid \perp + \mathbf{F}$ to get $\mathbf{K} + \mathbf{F}$, however

```
!check  K + ⊥ | ⊥ + F = J + ⊥  ⋣  K + F.
```

4.2 A constructor for simple types of fuzzy terms

This section describes our attempt to define closures for simple types in SKRJ . At the time of writing of this thesis, we have not succeeded in defining a Simple type constructor. Later sections in this chapter rely on a Simple type constructor, and serve to motivate what can be done if Simple is indeed definable.

The main difficulty in extending our definition of Simple from 3.6 is that we cannot raise different branches of Böhm trees with different section-retract pairs; i.e., the coherence lemma fails. This same difficulty prevents us from disambiguating terms in 5.7.

Interpreting simple types as closures

We now state the simple types conjecture for SKRJ , following the simple types theorem from SKJ . The languages of terms and types in 4.2 differ from the SKJ terms and types only in that we now have mixtures.

Conjecture 4.2.1. *There is an* SKRJ *-definable term* Simple *and corresponding interpretation* $[-]$ *of simple types, following 3.6, such that for each term* q *and simple type* τ *as in Figure 4.2,* $\mathsf{q}::\tau$ *iff* $\mathsf{q}:[\tau]$.

Our current best attempt is similar to the Simple from SKJ . We first define two section-retract pairs

```
raise  :=  (λx, −. x).
lower  :=  (λx. x ⊤).
!check  raise∘lower = I.
!check  lower∘raise  ⋢  I.

join  :=  (λf, x, y. f (x | y)).
copy  :=  (λf, x. f x x).
!check  join∘copy = I.
!check  copy∘join  ⋢  I.
```

term formation

$$\frac{}{\mathbf{S}\ \mathsf{tm}} \qquad \frac{}{\mathbf{K}\ \mathsf{tm}} \qquad \frac{}{\top\ \mathsf{tm}} \qquad \frac{\mathsf{m}\ \mathsf{tm} \qquad \mathsf{n}\ \mathsf{tm}}{\mathsf{m}\ \mathsf{n}\ \mathsf{tm}} \qquad \frac{\forall \mathsf{m} \in \underline{\mathsf{M}}.\ \mathsf{m}\ \mathsf{tm}}{\bigsqcup \underline{\mathsf{M}}\ \mathsf{tm}} \qquad \frac{\forall \mathsf{m} \in \mathsf{supp}\ \underline{\mathsf{M}}.\ \mathsf{m}\ \mathsf{tm}}{\mathbb{E}[\underline{\mathsf{M}}]\ \mathsf{tm}}$$

type formation

$$\frac{\mathsf{a}\ \mathsf{var}}{\mathsf{G}, \mathsf{a}, \mathsf{G}' \vdash \mathsf{a}\ \mathsf{tp}} \qquad \frac{\mathsf{G} \vdash \sigma\ \mathsf{tp} \qquad \mathsf{G} \vdash \tau\ \mathsf{tp}}{\mathsf{G} \vdash \sigma \to \tau\ \mathsf{tp}} \qquad \frac{}{\mathsf{G} \vdash \mathsf{any}\ \mathsf{tp}}$$

typing

$$\frac{\mathsf{G} \vdash \rho\ \mathsf{tp} \qquad \mathsf{G} \vdash \sigma\ \mathsf{tp} \qquad \mathsf{G} \vdash \tau\ \mathsf{tp}}{\mathbf{S} :: (\rho \to \sigma \to \tau) \to (\rho \to \sigma) \to \rho \to \tau} \qquad \frac{\mathsf{G} \vdash \sigma\ \mathsf{tp} \qquad \mathsf{G} \vdash \tau\ \mathsf{tp}}{\mathbf{K} :: \sigma \to \tau \to \sigma} \qquad \frac{\mathsf{G} \vdash \tau\ \mathsf{tp}}{\top :: \tau} \qquad \frac{\mathsf{m} :: \sigma \to \tau \qquad \mathsf{n} : \sigma}{\mathsf{m}\ \mathsf{n} :: \tau}$$

$$\frac{\forall \mathsf{m} \in \underline{\mathsf{M}}.\ \mathsf{m} :: \tau}{\bigsqcup \underline{\mathsf{M}} :: \tau} \qquad \frac{\forall \mathsf{m} \in \mathsf{supp}\ \underline{\mathsf{M}}.\ \mathsf{m} :: \tau}{\mathbb{E}[\underline{\mathsf{M}}] :: \tau} \qquad \frac{\mathsf{m}\ \mathsf{tm}}{\mathsf{m} :: \mathsf{any}}$$

Figure 4.2: Typing and type interpretation. $\mathbb{E}[\underline{\mathsf{M}}]$ denotes expectation or integration over a probability distribution $\underline{\mathsf{M}}$ with support $\underline{\mathsf{M}}$. $\bigsqcup \underline{\mathsf{M}}$ denotes the join or convex hull generated by a set $\underline{\mathsf{M}}$ of SKRJ -terms-as-CSPDs.

and then recursively close under conjugation.

```
Simple  :=  (any→V) (
    Yλs, f. f I I
          | f raise lower
          | f join copy
          | sλa, a′. sλb, b′. f (a′→b) (a→b′)
).
```

Note that we need section-retract pairs to distribute over CSPDs of head normal forms. Each of raise, lower, join, copy and their conjugates do, however, this requirement prevents us from using pairs like

```
open  :=  (λf, ⟨x⟩. f x).
pack  :=  (λf, x. f⟨x⟩).

curry  :=  (λf, x, y. f(x, y)).
uncurry  :=  (λg, (x, y). g x y).
```

4.3 Monadic types as closures

In this section we define a monadic type Fuzzy for CSPDs, and define compatible basic types and type constructors. Our main tool is the Simple type constructor defined in 3.6. However, unlike our treatment in 3.7, we will not be able to use disambiguation tricks to prevent, e.g. $\mathbf{K} \mid \mathbf{F} : \mathsf{bool}$.

Monadic types

The Kleisli triple presentation of a monad ([Wad90], [Mog91]) is a functor Fuzzy, Map together with a pair of natural transformations Exactly, Lift:

$$\begin{aligned}
\mathsf{Fuzzy} &: \mathbf{V}\to\mathbf{V}\\
\mathsf{Map} &: \forall \mathsf{a}:\mathbf{V}, \mathsf{b}:\mathbf{V}.\ (\mathsf{a}\to\mathsf{b})\to\mathsf{Fuzzy\ a}\to\mathsf{Fuzzy\ b}\\
\mathsf{Exactly} &: \forall \mathsf{a}:\mathbf{V}.\ \mathsf{a}\to\mathsf{Fuzzy\ a}\\
\mathsf{Lift} &: \forall \mathsf{a}:\mathbf{V}, \mathsf{b}:\mathbf{V}.\ (\mathsf{a}\to\mathsf{Fuzzy\ b})\to\mathsf{Fuzzy\ a}\to\mathsf{Fuzzy\ b}
\end{aligned}$$

Specializing to SKRJ, Exactly is the identity, and Lift = Map basically sample points from CSPDs. The main discovery of this section is that we can implement Lift for the standard Church-style booleans, naturals, etc., by using a continuation passing transform.

Our Fuzzy types will be the same as the previous SKJ types-as-closures in 3.7, only we will not disambiguate (so e.g. $\mathbf{K} \mid \mathbf{F} : \mathsf{bool}$ here). Lift is not definable uniformly over types; however we will attempt to define it uniformly for type constructors (e.g. define Lift_{a+b} in terms of Lift_a and Lift_b).

$$\begin{aligned}
&\mathsf{Test} := (\mathbf{V}\to\mathbf{V})\ (\lambda a.\ a\to \mathsf{semi}). && \textit{a type of tests}\\
&\mathsf{Lift} := (\lambda a:\mathbf{V}.\ \mathbf{P}\ (\mathsf{Endo}\ (a\to\mathsf{any}))\ (\mathsf{Above}\ \lambda f,-.\ f\ \bot)). && \textit{a type of lifters}\\
&\mathsf{testable} := (\exists a:\mathbf{V}.\ \mathsf{Test}\ a).\\
&\mathsf{fuzzy} := (\exists a:\mathbf{V}.\ \mathsf{Prod}\ (\mathsf{Test}\ a)\ (\mathsf{Lift}\ a)).
\end{aligned}$$

It will be helpful in constructing types to ensure that $\mathsf{lift}\ f \sqsupseteq \lambda-.\ f\ \bot$, i.e., lifting is at least as informative as blinding $f \mapsto \lambda-.\ f\ \bot$ (however blinding can safely be ignored in the lifting theorem below).

To reason about the behavior of lifted functions by their action on unambiguous things, we will sample them at various points. The sampling process should distribute over CSPDs as follows: For this we need

Theorem Schema: (sampling) Let a be a "nice" testable type, and $f : a \to \mathsf{any}$ be a strict co-strict function, i.e., $f\ \bot = \bot$ and $f\ \top = \top$. Then lifted functions distribute over CSPDs, i.e., letting $f' := \mathsf{Lift}_a\ f$ and $x, y : a$,

$$\begin{aligned}
f'(x \mid y) &= f'\ x \mid f'\ y\\
f'(x + y) &= f'\ x + f'\ y
\end{aligned}$$

and lifting perserves the identity

$$\forall x:a::\mathsf{test}_a.\ \mathsf{lift}_a\ \mathbf{I}\ x = x$$

Proof Schema: We will typically implement lifting functions by performing a continuation passing transform, so that $\mathsf{lift}_a\ f$ is a trace $\langle M_1, \ldots, M_n\rangle$. Distributivity of $\mathsf{lift}_a\ f$ then follows from distributivity of right-application over joins and mixtures:

$$\begin{aligned}
\mathsf{lift}_a\ f\ (x \mid y) &= \langle M_1,\ldots,M_n\rangle\ (x\mid y)\\
&= (x\mid y)\ M_1\ \ldots\ M_n\\
&= x\ M_1\ \ldots\ M_n \mid y\ M_1\ \ldots\ M_n && \textit{by distributivity}\\
&= \langle M_1,\ldots,M_n\rangle\ x \mid \langle M_1,\ldots,M_n\rangle\ y\\
&= \mathsf{lift}_a\ f\ x \mid \mathsf{lift}_a\ f\ y
\end{aligned}$$

and similarly for mixtures $x + y$. Preservation of identity will depend on the implementations. □

We would like to formally verify such theorems in Johann by eliminating universal quantifiers over fuzzy types a and functions f. We can eliminate the quantifier over f using the test for a, but we cannot eliminate the quantifier over fuzzy types a, since the closure fuzzy fails to ensure the important property

$$\forall x:a.\ \mathsf{test}_a\ x = \top \implies x = \top$$

Thus we uniformize all three equations for verification, parametrized by fuzzy type:

```
sampling_thm_J := (fuzzy → prod) (
    λ(a, test_a, lift_a), w, f, x, y.
    f′ := lift_a (λx. test_a x. f x).
    w (f′(x | y)) (f′ x | f′ y)
).
sampling_thm_R := (fuzzy → prod) (
    λ(a, test_a, lift_a), w, f, x, y.
    f′ := lift_a (λx. test_a x. f x).
    w (f′(x + y)) (f′ x + f′ y)
).
sampling_thm_I := (fuzzy → prod) (
    λ(a, test_a, lift_a), w, x : a :: test_a.
    (x, lift_a I x)
).
```

To verify the theorems a specific type fuzzy_a, we show that the equations are symmetric under swapping

```
swap := (prod → prod) (λ(x, y). (y, x)).
```

as in

```
!check sampling_thm_J fuzzy_a : swap.
!check sampling_thm_R fuzzy_a : swap.
!check sampling_thm_I fuzzy_a : swap.
```

Fuzzy atomic types

Let's begin by define sampling functions for types nil, unit, bool, using their fuzzy forms.

```
nil := (Simple λa, a′. a′).
unit := (Simple λa, a′. a → a′ | ⟨I⟩).
bool := (Simple λa, a′. a → a → a′ | ⟨K, F⟩).
```

```
test_nil := Test nil ⟨⟩.
test_unit := Test unit ⟨I⟩.
test_bool := Test bool ⟨I, I⟩.
```

```
lift_nil := Lift nil (λf. ⟨⟩).
lift_unit := Lift unit (λf. ⟨f I⟩).
lift_bool := Lift bool (λf. ⟨f K, f F⟩).
```

```
fuzzy_nil := fuzzy (nil, test_nil, lift_nil).
fuzzy_unit := fuzzy (unit, test_unit, lift_unit).
fuzzy_bool := fuzzy (bool, test_bool, lift_bool).
```

Theorem 4.3.1. *(sampling* nil, unit, *and* bool*)*

```
!check sampling_thm_J fuzzy_nil : swap.
!check sampling_thm_R fuzzy_nil : swap.
!check sampling_thm_I fuzzy_nil : swap.
```

```
!check sampling_thm_J fuzzy_unit : swap.
!check sampling_thm_R fuzzy_unit : swap.
!check sampling_thm_I fuzzy_unit : swap.
```

```
!check sampling_thm_J fuzzy_bool : swap.
!check sampling_thm_R fuzzy_bool : swap.
!check sampling_thm_I fuzzy_bool : swap.
```

Proof. Left as exercise for Johann. □

Naturals are a bit trickier: we need to pay special attention to partial naturals like succ(succ ⊥). We define a type and a test

```
nat := V (
    Y λa. Simple λb, b'. (b' → b) → b → b'
                   | ⟨λn:a, f:a → a, x:a. f(n f x), λ−, x:a. x⟩
).
test_nat := Test nat ⟨K I, I⟩.
```

with intro forms

```
zero := (λ−, x. x).
succ := (λn:nat, f, x. f(n f x) | n f(f x)).
```

Note that succ joins over two ways to increment. Now to lift

```
lift_nat := Lift nat (Y λl. ⟨λn, f. f ⊥ | l f∘succ n,  λf. f zero⟩).
```

Finally we package these together as a fuzzy type.

```
fuzzy_nat := fuzzy (nat, test_nat, lift_nat).
```

Theorem 4.3.2. *(sampling* nat*)*

```
!check sampling_thm_J fuzzy_nat : swap.
!check sampling_thm_R fuzzy_nat : swap.
!check sampling_thm_I fuzzy_nat : swap.
```

Proof. Left as exercise for Johann. □

Fuzzy covariant type constructors

We define products, sums, and successor types as in 3.7 and tests as in 3.10.

```
Prod  := (V → V → V) (λa, b. Simple λc, c'. (a → b → c) → c').
Sum   := (V → V → V) (λa, b. Simple λc, c'. (a → c) → (b → c) → c').
Maybe := (V → V) (λa. Simple λc, c'. c → (a → c) → c').
```

```
test_Prod := (∀(a, −):testable, (b, −):testable. Test. Prod a b) (
    λ(−, ta), (−, tb), (x, y). and_semi (ta x) (tb y)
).
test_Sum := (∀(a, −):testable, (b, −):testable. Test. Sum a b) (
    λ(−, ta), (−, tb). ⟨λx. I | ta x, λy. I | tb y⟩
).
test_Maybe := (∀(a, −):testable. Test. Maybe a) (
    λ(−, ta). ⟨I, λx. I | ta x⟩
).
```

These have intro forms

```
π1 := (λ(x, −). x).    inl := (λx, f, −. f x).     none := (λf, −. f).
π2 := (λ(−, y). y).    inr := (λx, −, g. g x).     some := (λx, −, g. g x).
```

and lifters

```
lift_Prod  :=  (∀(a, −) : fuzzy, (b, −) : fuzzy. Lift. Prod a b) (
    λ(−, −, la), (−, −, lb), f. ⟨la λx. lb λy. f (x, y)⟩
).
lift_Sum  :=  (∀(a, −) : fuzzy, (b, −) : fuzzy. Lift. Sum a b) (
    λ(−, −, la), (−, −, lb), f. ⟨la f∘inl, lb f∘inr⟩
).
lift_Maybe  :=  (∀(a, −) : fuzzy. Lift. Maybe a) (
    λ(−, −, la), f. ⟨f none, la f∘some⟩
).
```

We package all of this information together as fuzzy types.

```
fuzzy_Prod  :=  (fuzzy → fuzzy → fuzzy) (
    λaa. (a, ta, la)  :=  aa.
    λbb. (b, tb, lb)  :=  bb.
    (Prod a b, test_Prod (a, ta) (b, tb), lift_Prod aa bb)
).
fuzzy_Sum  :=  (fuzzy → fuzzy → fuzzy) (
    λaa. (a, ta, la)  :=  aa.
    λbb. (b, tb, lb)  :=  bb.
    (Sum a b, test_Sum (a, ta) (b, tb), lift_Sum aa bb)
).
fuzzy_Maybe  :=  (fuzzy → fuzzy) (
    λaa. (a, ta, la)  :=  aa.
    (Maybe a, test_Maybe (a, ta), lift_Maybe aa)
).
```

We cannot verify general sampling theorems for algebraic these types, since the type constructors take arguments whose fuzziness cannot be guaranteed with a closure, and thus cannot be eliminated.

Theorem 4.3.3. *(sampling products, sums, and successors) If sampling works for* a, b : fuzzy*, then it also works for* fuzzy_Prod a b*, for* fuzzy_Sum a b*, and for* fuzzy_Maybe a*.*

Proof. By distributivity of traces over joins and mixtures. □

We can verify some particular cases:

```
!check sampling_thm_J (fuzzy_Prod fuzzy_bool fuzzy_bool) : swap.
!check sampling_thm_R (fuzzy_Prod fuzzy_bool fuzzy_bool) : swap.
!check sampling_thm_I (fuzzy_Prod fuzzy_bool fuzzy_bool) : swap.
```

```
!check sampling_thm_J (fuzzy_Sum fuzzy_bool fuzzy_bool) : swap.
!check sampling_thm_R (fuzzy_Sum fuzzy_bool fuzzy_bool) : swap.
!check sampling_thm_I (fuzzy_Sum fuzzy_bool fuzzy_bool) : swap.
```

```
!check sampling_thm_J (fuzzy_Maybe fuzzy_bool) : swap.
!check sampling_thm_R (fuzzy_Maybe fuzzy_bool) : swap.
!check sampling_thm_I (fuzzy_Maybe fuzzy_bool) : swap.
```

The coalgebraic numeral type is the fixed points of the successor

```
num  :=  Y Maybe.
```

```
test_num := (Y λt. Test_Maybe (num, t)).
```

```
lift_num := Lift num (λf. ⟨f zero, f∘succ⟩).
```

```
fuzzy_num := fuzzy (num, test_num, lift_num).
```

Theorem 4.3.4. *(sampling* num*)*

```
!check sampling_thm_J fuzzy_num : swap.
!check sampling_thm_R fuzzy_num : swap.
!check sampling_thm_I fuzzy_num : swap.
```

Proof. By induction, using the sampling theorem for Maybe above. □

Coinductive stream types are fixed points of products

```
Stream := (V→V) (λa. Y (Prod a)).
stream := Stream ⊥.
```

```
test_Stream := (∀(a,−):testable. Test a) (
     λ(a,ta). Y λt. Test_Prod (a,ta) (Stream a,t)
).
```

```
lift_Stream := (∀(a,−):fuzzy. Lift. Stream a) (
     λaa. (a,ta,la) := aa.
     ts := test_Stream (a,ta).
     Yλls. lift_Prod aa (Stream a, ts, ls)
).
```

```
fuzzy_Stream := (fuzzy → fuzzy) (
     λaa. (a,ta,la) := aa.
     (Stream a, test_Stream (a,ta), lift_Stream (a,ta,la))
).
```

Dialogs, e.g. $\mathbf{V}$ ($\mathbf{Y}\lambda$a. Sum a. $\mathbf{W}$ Prod a), are similarly definable.

Theorem 4.3.5. *(sampling streams) If sampling works for* a : fuzzy, *then it also works for* fuzzy_Stream a.

Proof. By coinduction, using the sampling theorem for Prod above. □

We can verify some particular cases:

```
!check sampling_thm_J (fuzzy_Stream fuzzy_bool) : swap.
!check sampling_thm_R (fuzzy_Stream fuzzy_bool) : swap.
!check sampling_thm_I (fuzzy_Stream fuzzy_bool) : swap.
```

```
!check sampling_thm_J (fuzzy_Stream fuzzy_nat) : swap.
!check sampling_thm_R (fuzzy_Stream fuzzy_nat) : swap.
!check sampling_thm_I (fuzzy_Stream fuzzy_nat) : swap.
```

Fuzzy exponentials

We can define fuzzy exponenetials for general types that support case analysis for partial terms. That is, let $(\mathsf{a}, \mathsf{test_a}, \mathsf{lift_a}) : \mathsf{fuzzy}$ be a fuzzy type with inhabitants $\{\bot, x_1, \ldots, x_n, \top\}$ such that for each i, $x_i :: \mathsf{test_a}\ x_i$. Suppose there is an elim form $\mathsf{case_a}$ implemented as a trace, and satisfying $\mathsf{case_a}\ x_i\ y_1\ \ldots\ yn = yi$ for each $i = 1, \ldots, n$. Then we can implement fuzzy exponentials from a using the test

```
test_{a→b}  :=  Test (a→b) (
    λf:a→b. test_b (f x_1) | ... | test_b (f x_n)
).
```

and lifter

```
lift_{a→b}  :=  Lift (a→b) (
    λf:a→b::test_{a→b}, x:a::test_a.  case_a x (f x_1) ... (f x_n)
).
```

Examining some special cases, consider first the domain bool → unit, with functions discriminated by response to well-checked arguments $\bot, \mathbf{K}, \mathbf{F}$

```
        T
        |
       K I
      /   \
  <I,_>   <_,I>
      \   /
        _
```

where we ensure $\top$ always maps to $\top$ (and hence ignore $\langle\top,\bot\rangle$ and $\langle\bot,\top\rangle$).

```
b2u  :=  P (bool→unit) (⟨K,⊥⟩ | ⟨F,⊥⟩).        ensure toplessness
test_b2u  :=  Test b2u (⟨K⟩ | ⟨F⟩).
```

```
lift_b2u  :=  Lift b2u (
    λ−, f, p. p ⊥ (f λ− . I)
            | p K ( f ⟨I,⊥⟩ | p F (f ⟨I,I⟩) )
            | p F ( f ⟨⊥,I⟩ | p K (f ⟨I,I⟩) )
).
```

```
fuzzy_b2u  :=  fuzzy (b2u, test_b2u, lift_b2u).
```

Theorem 4.3.6. *(sampling* b2u*) Sampling works for* b2u*:*

```
!check sampling_thm_J fuzzy_b2u : swap.
!check sampling_thm_R fuzzy_b2u : swap.
!check sampling_thm_I fuzzy_b2u : swap.
```

Proof. Left as exercise for Johann. □

For less trivial codomain, we add branching, as in bool → bool

```
b2b  :=  P (bool→bool) (⟨K,⊥,⊥⟩ | ⟨F,⊥,⊥⟩).
test_b2b  :=  Test b2b (⟨K,I,I⟩ | ⟨F,I,I⟩).
```

```
lift_b2b := Lift b2b (
    λf,p. p ⊥ (f λ− . I)
        | p K ( f ⟨K,⊥⟩ | p F (f ⟨K,K⟩) (f ⟨K,F⟩) )
              ( f ⟨F,⊥⟩ | p F (f ⟨K,K⟩) (f ⟨K,F⟩) )
        | p F ( f ⟨⊥,K⟩ | p K (f ⟨K,K⟩) (f ⟨F,K⟩) )
              ( f ⟨⊥,F⟩ | p K (f ⟨K,F⟩) (f ⟨F,F⟩) )
).
```

```
fuzzy_b2b := fuzzy (b2b, test_b2b, lift_b2b).
```

Theorem 4.3.7. *(sampling* b2u*)*

```
!check sampling_thm_J fuzzy_b2u : swap.
!check sampling_thm_R fuzzy_b2u : swap.
!check sampling_thm_I fuzzy_b2u : swap.
```

Proof. Left as exercise for Johann. □

Finally, we consider functions of infinite domains, say num→semi (witnessed by semi-streams). We can no longer implement case analysis as traces, but we can recursively sample, as in lift_Stream above.

```
n2s := P (num→semi) (Yλc,q. q none ⊥ | c q∘some).
test_n2s := Test n2s (Yλc,q. q none | c q∘some).
```

```
lift_n2s := Lift n2s (
    Yλl.
    λf,p. p ⊥ (λ− . I)
        | p none ( f (I,⊥) | l (λp′. f (I,p′)) )
        | l (λp′. f (⊥,p′)) p∘some
).
```

```
fuzzy_n2s := fuzzy (n2s, test_n2s, lift_n2s).
```

Theorem 4.3.8. *(sampling* n2s*)*

```
!check sampling_thm_J fuzzy_n2s : swap.
!check sampling_thm_R fuzzy_n2s : swap.
!check sampling_thm_I fuzzy_n2s : swap.
```

Proof. Left as exercise for Johann. □

Question 4.3.9. *What data is needed to uniformly define exponentials? Is this data itself uniformly definable, under exponentials?*

Chapter 5

Untyped λ-calculus with logical reflection (SKJO)

This chapter deals with automated reasoning in illative combinatory logic, the programme of using combinatory algebras as languages for logics. Building on the algebra SKJ and Johann's equational theorem proving, we develop two tools to achieve a powerful verification system:
(1) a code comonad à la Brookes for efficient storage of quoted terms, and
(2) an oracle for encoding Δ_1^1 statements about SKJO terms as booleans.
These two developments are independent, but in our implementation, the oracle relies on comonadic codes to achieve a reasonable density of information storage.

Our first verification tool is an coding of terms in a flat domain, so that algorithms can work with intensions of terms. Some coding of terms is needed if we are to reflect statements about the calculus into the calculus itself, i.e. to embed an intensional semantics into our extensional model. The natural way to define codes ([DP96], [WLPD98]) is with a modal type Code, functions Apply, Eval, Quote, and an operation $\{-\}$ to quote closed terms. This structure and its untyped version $\langle \mathsf{code}, \mathbf{A}, \mathbf{E}, \mathbf{Q}, \{-\}\rangle$ afford us notation for quoted pattern-matching of untyped and typed terms

$$\begin{array}{rcll}
\mathsf{M} & \mapsto & \{\mathsf{M}\} & \textit{quoting closed terms} \\
\{\mathsf{M}\ \mathsf{N}\} & = & \mathbf{A}\{\mathsf{M}\}\{\mathsf{N}\} & \textit{application} \\
(\mathbf{let}\ \{\mathsf{x}\} := \mathsf{M}.\ \mathsf{x}) & = & \mathbf{E}\ \mathsf{M} & \textit{evaluation} \\
(\mathbf{let}\ \{\mathsf{x}\} := \mathsf{M}.\ \{\mathsf{x}\}) & = & \mathsf{code}\ \mathsf{M} & \textit{typing} \\
(\mathbf{let}\ \{\mathsf{x}\} := \mathsf{M}.\ \{\{\mathsf{x}\}\}) & = & \mathbf{Q}\ \mathsf{M} & \textit{quoting quoted terms}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{M}:\mathsf{a} & \mapsto & \{\mathsf{M}\}:\mathsf{Code}\{\mathsf{a}\} \\
\mathsf{M}:\mathsf{a}\to\mathsf{b}, \mathsf{N}:\mathsf{a}\ \vdash\ \{\mathsf{M}\ \mathsf{N}\} & = & \mathsf{Apply}\{\mathsf{a}\}\{\mathsf{b}\}\{\mathsf{M}\}\{\mathsf{N}\} \\
(\mathbf{let}\ \{\mathsf{x}:\mathsf{a}\} := \mathsf{M}.\ \mathsf{x}) & = & \mathsf{Eval}\{\mathsf{a}\}\ \mathsf{M} \\
(\mathbf{let}\ \{\mathsf{x}:\mathsf{a}\} := \mathsf{M}.\ \{\mathsf{x}\}) & = & \mathsf{Code}\{\mathsf{a}\}\ \mathsf{M} \\
(\mathbf{let}\ \{\mathsf{x}:\mathsf{a}\} := \mathsf{M}.\ \{\{\mathsf{x}\}\}) & = & \mathsf{Quote}\{\mathsf{a}\}\ \mathsf{M}
\end{array}$$

Note that $\mathsf{Quote}\{\mathsf{a}\}:\mathsf{Code}\{\mathsf{a}\}\to\mathsf{Code}\{\mathsf{Code}\{\mathsf{a}\}\}$ can be applied to arbitrary terms of type code, and $\{-\}$ can be applied to closed terms of arbitrary type, but variables of arbitrary type cannot be quoted. This restriction allows quoting to be non-monotone, so that codes can provide a flat representation of an ordered domain.

The problem with a naively modal type of codes is code size: Johann remembers facts about extensional terms only once per equivalence class, but intensional terms would require overhead for each inhabitant of an equivalence class. For example, Johann knows that $\mathbf{S}\ \mathbf{K} = \mathbf{K}\ \mathbf{I}$, and so needs store only one of the equations $(\mathbf{K}\ \mathbf{I})\mathbf{I} = \mathbf{I}$ and $(\mathbf{S}\ \mathbf{K})\mathbf{I} = \mathbf{I}$. However the naively modal type of codes, the intensions are different: $\{\mathbf{S}\ \mathbf{K}\} \neq \{\mathbf{K}\ \mathbf{I}\}$, so each equation $\mathbf{E}(\mathbf{A}\{\mathbf{S}\ \mathbf{K}\}\{\mathbf{I}\}) = \mathbf{E}\{\mathbf{I}\}$ and $\mathbf{E}(\mathbf{A}\{\mathbf{K}\ \mathbf{I}\}\{\mathbf{I}\}) = \mathbf{E}\{\mathbf{I}\}$ must be stored separately.

Our solution is to define intensional codes *modulo extensionality*, to achieve a space-efficient 1-1 coding of terms, while still neutralizing the Scott order, so that, e.g., $\bot \sqsubseteq \top$ but $\{\bot\}$ and $\{\top\}$ are order-incomparable. This solution is captured by the two reasoning principles

$$\frac{\vdash \mathsf{M} = \mathsf{N}}{\vdash \{\mathsf{M}\} = \{\mathsf{N}\}}\ \text{(extensional)} \qquad\qquad \frac{\vdash \mathsf{M} \neq \mathsf{N}}{\vdash \{\mathsf{M}\} \not\sqsubseteq \{\mathsf{N}\}}\ \text{(neutral)}$$

The 1-1 coding of terms is only consistent only w.r.t. *provable* equality, but this turns out to be just as space efficient as a true 1-1 coding would have been. Adding the extensionality equations to our modal type system yields the richer structure of a *computational comonad*, as studied by Brookes and Geva [BG92]. The extensional computational comonad provides a tower of isomorphisms between the code types, where all but the first operation $\{-\}$ are SKJ -code-definable.

```
    eval          eval                  eval
   <-----        <-----                <-----
a        Code{a}        Code{Code{a}}         ...
   - - ->        ----->                ----->
    {-}          quote                 quote
```

Our second verification tool is an extension of SKJ with a Π^1_1 semi-hard (hence Δ^1_1 bool-hard) oracle $\mathbf{O}$, to achieve an elegant illative combinatory logic SKJO for hyperarithmetic functions. It is well-known that the equational logic of hyperarithmetic functions is equivalent in logical strength to predicative set theory ([Fef05]). Moreover, Feferman, Weyl, Poincare and later Friedman have shown that "most" of mainstream mathematics can be carefully formulated within predicative set theory ([Fef92]). Thus SKJO is sufficiently strong to serve as a foundation for "most" of mathematics.

Our approach is to add an oracle answering a simple (but uncomputable) problem, but allow it to recursively call itself as a subroutine. There is some freedom as to what sort of simple question the oracle solves at each stage. An obvious choice would be to have the oracle solve the halting problem, reducing a Π^0_1-complete problem down to a Δ^0_1 problem. However, since we are starting with the Π^0_2-complete equational theory $\mathcal{H}^*$ (recall from 3.9), it is more convenient to be able to reduce a Π^0_2-complete problem down to a Δ^0_2 problem. Specifically, we want to eliminate universal quantifiers over some countable domain, converting them to semiboolean values, e.g.,

$$\mathbf{O}_{\mathsf{nat}}\{\phi\} = \begin{cases} \mathbf{I} & \text{if } \forall \mathsf{n} \in \mathbb{N}.\ \phi(\mathsf{n}) = \mathbf{I} \\ \bot & \text{otherwise} \end{cases}$$

We can make the oracle even more convenient if we abstract out the domain as a parameter to $\mathbf{O}$, i.e. as a code for the totality test for that domain (so that, e.g., $\mathbf{O}_{\mathsf{nat}} = \mathbf{O}\{\mathsf{test\_nat}\}$). Thus we choose to let $\mathbf{O}$ answer the Π^0_2-complete *subtest* problem
Given: an SKJ -code $\{\mathsf{s}\}$ for $\mathsf{s}:\mathsf{test}$, and an SKJO -code $\{\mathsf{t}\}$ for $\mathsf{t}:\mathsf{test}$

Semidecide: $\mathbf{O}\{\mathsf{s}\}\{\mathsf{t}\} = \mathbf{I} \iff \mathsf{s} <:: \mathsf{t}$

where the subtest relation is defined as

$$\mathsf{s} <:: \mathsf{t} \iff (\forall \mathsf{x}::\mathsf{s}.\ \mathsf{x}::\mathsf{t}) \iff \forall \mathsf{x}.\ (\mathsf{s}\ \mathsf{x} = \mathbf{I} \implies \mathsf{t}\ \mathsf{x} = \mathbf{I})$$

Recall from 3.9 that the testing problem $\mathsf{x}::\mathsf{t} \iff \mathsf{semi}(\mathsf{t}\ \mathsf{x}) = \mathbf{I}$ is Δ^0_2, so the subtest problem is Π^0_2. We will define $\mathbf{O}$ more precisely in 5.5.

5.1 Related work

Curry developed many systems of illative combinatory logic, but his strongest systems were plagued by inconsistencies. See Curry's volume [CHS72] for an overview of his approach. [BBD93] discusses an illative combinatory logic for first-order predicate calculus; our present logic is stronger.

Harrison ([Har95]) surveys a wide variety of reflection principles in logic. Nuprl's principle of proof reflection is detailed in [ACHA90] and surveyed in [Con94]. Demers ([DM95]) surveys reflection principles in logic, functional programming and metaprogramming.

Kieffer, Avigad and Friedman [KAF08] propose an extension ZFC with partial terms as a human-motivated foundation for mathematics. They compute statistics as to the complexity of formulas on a corpus of data. The language SKJO can be motivated as a foundation of math with simplified syntax, so that such statistics are much easier to compute.

5.2 Axioms for an extensional code type ($\mathsf{code}, \mathbf{A}, \mathbf{E}, \mathbf{Q}, \{-\}$)

In this section we axiomatize a comonadic type for codes modulo provable equality. We start with a definable type for free codes in a language $\mathbb{C}$ defined by

$\mathbb{C}$ ::=	$\mathbb{C}\ \mathbb{C}$	*binary application*
	$\mid \mathbf{S} \mid \mathbf{K} \mid \mathbf{J} \mid \mathbf{O} \mid \mathsf{code}$	*and five atoms*

Next we axiomatize a quotient type code $<:$ code_0, modulo Johann's theory of equality. This quotient type is then extensional in that provably equal terms have provably equal codes. But the type is also intensional in that the order of terms is neutralized by quoting, so that we can discriminate codes with a predicate $\mathsf{equal\_code} : \mathsf{code} \to \mathsf{code} \to \mathsf{bool}$, total up to Johann's theory of equality. Thus we achieve a good balance between intensionality (which is required to discriminate between codes), and extensionality (which is desired for space efficiency).

It turns out that the quotient type code and introduction form for application $\mathbf{A}\{-\}$ together form a functor from the category of unityped terms under composition to the category of codes under quoted composition. Moreover the evaluation and quotation operators $\mathbf{E}, \mathbf{Q}$ form a comonadic pair of natural transformations between 1 and the $(\mathsf{code}, \mathbf{A}\{-\})$ functor. Finally, the quoting operation $\{-\}$ on closed terms provides a third natural transformation. This combined structure $\langle \mathsf{code}, \mathbf{A}\{-\}, \mathbf{E}, \mathbf{Q}, \{-\}\rangle$, together with a few equations, constitutes a *computational comonad* (as observed in a similar setting by Brookes and Geva [BG92]). Weaker forms of quoting, with the same signature but missing some equations, are well-studied in the programming languages literature (e.g., [DP96], [WLPD98]). However, those equations –the computational comonad conditions– are exactly what makes extensional codes attractive for our purposes, and seem only to hold in the presence of some extensionality principle.

Axioms for a quotient type of codes

We start with a closure,test pair for free codes

```
code_0 := P (Yλa. Sset. Sum (Prod a a) (num_below 5)).
test_code_0 := Test code_0 (
    Yλt. test_Sset. test_Sum (test_Prod t t) test_num
).
!check code_0 = Sset (Sum (W Prod code_0) (num_below 5)).
!check code_0 <: Sset (Sum (W Prod code_0) num).
```

The type code will be a similar but undefinable type

```
!using code test_code.
!assume code : V.
!assume code <: Sset (Sum (W Prod code) num).
!assume test_code = Test code test_code_0.
```

which is a quotient of the code_0 type

```
!assume code <: code_0.
!assume test_code <:: test_code_0.
```

For introduction forms, we need codes for combinators (application, $\mathbf{S}$, $\mathbf{K}$, and $\mathbf{J}$), a code for the oracle $\mathbf{O}$ (which will be axiomatized later in 5.5), and access to the code comonad itself (via code). Note that Johann's theory is Σ^0_1 at any time, so the atom code has no extra logical power beyond the Turing-complete language SKJ . However, to allow Johann's theory to change over time, we must refrain from statically *defining* code as an SKJ term; having access to code in the basis allows us to be agnostic about its SKJ -definition. Thus we introduce atoms

```
atom_code := (nat → code) (λn. ⟨inr. nat2num n⟩).
!using O.
!assume atom_code 0 = {S}.
!assume atom_code 1 = {K}.
!assume atom_code 2 = {J}.
!assume atom_code 3 = {O}.
!assume atom_code 4 = {code}.
!check atom_code 5 = ⊤.
!assume ( {S}, {K}, {J}, {O}, {code} :: test_code ).
```

and applications

```
!define A := Bin_op code (λx, y. ⟨inl(x, y)⟩).
!assume test_code (A x y) = and_semi (test_code x) (test_code y).
!assume A{x}{y} :: test_code.
!check {x y} = A{x}{y}.                              just checking notation
```

(We will explain the semantics for quoted quantification $\forall\{x\}.\ \phi(x, \{x\})$ shortly.)

We define evaluation and quoting operations $\mathbf{E}, \mathbf{Q}$ by simultaneous recursion, using only the atoms $\mathbf{S}, \mathbf{K}, \mathbf{J}, \mathbf{O}$, code and the quoting operation $\{-\}$ on various closed terms.

```
let (eval_code, quote_code) := (
    Yλ(e, q). (
        (code → any) ⟨(
            λ(x, y). (e x)(e y),
            S, K, J, O, code,
            error
        )⟩,
        (code → code) ⟨(
            λ(x, y). A (A {A} (q x)) (q y),
            {{S}}, {{K}}, {{J}}, {{O}}, {{code}},
            error
        )⟩
    )
).
```

```
!define E := eval_code.
!check E = (code → any) ⟨(λ(x, y). (E x)(E y), S, K, J, O, code, error)⟩.
```

```
!check E{S} = S.
!check E{K} = K.
!check E{J} = J.
!check E{O} = O.
!check E{code} = code.
!check E{x y} = (E{x})(E{y}).
```

```
!define Q := quote_code.
!check Q = (code→code) ⟨(
    λ(x,y). A (A {A} (Q x)) (Q y),
    {{S}},{{K}},{{J}},{{O}},{{code}},
    error
)⟩.
```

```
!check Q{S} = {{S}}.
!check Q{K} = {{K}}.
!check Q{J} = {{J}}.
!check Q{O} = {{O}}.
!check Q{code} = {{code}}.
!check Q{x y} = A (A {A} (Q{x})) (Q{y}).
```

Next we assume a very natural set of 15 equations that turn out to axiomatize $\langle \mathsf{code}, \mathbf{A}\{-\}, \mathbf{E}, \mathbf{Q}, \{-\}\rangle$ being a computational monad. Because the definition of a comonad is clearer in a typed theory (i.e., in a category with more than one object), we will delay the definition until we introduce an type-indexed comonad $\langle \mathsf{Code}, \mathsf{Apply}, \mathsf{Eval}, \mathsf{Quote}\rangle$ in 5.3. For now, simply observe that the following equations are all natural assumptions that are desirable for a type of extensional codes.

```
functoriality
!assume code : V.
!assume A{f} : code→code.
!assume A{I} = code.
!assume A{f} ; A{g} = A{f;g}.
```

```
naturality
!assume E : code→any.
!assume Q : code→code.
!assume A{f} ; E = E ; f.
!assume A{f} ; Q = Q ; A{A{f}}.
```

```
comonad conditions
!assume Q ; E = code.
!assume Q ; A{E} = code.
!assume Q ; Q = Q ; A{Q}.
```

```
computation conditions
!assume {x} : code.
!assume A{f}{x} = {f x}.
!assume E{x} = x.
!assume Q{x} = {{x}}.
```

Axioms and axiom schemata for extensionality

Ideally we would like a fully extensional 1-1 coding of terms, so that each $\mathcal{H}^*$ equivalence class corresponded to a unique code; that way there would be as few as possible codes taking up space in Johann's database. This ideal is captured by the reasoning principles

ideal coding

$$\frac{\mathsf{M} = \mathsf{N}}{\{\mathsf{M}\} = \{\mathsf{N}\}}\ (1-1) \qquad\qquad \frac{\mathsf{M} \neq \mathsf{N}}{\mathsf{code}(\{\mathsf{M}\} \mid \{\mathsf{N}\}) = \top}\ (\mathsf{flat})$$

$$\frac{\mathsf{c} :: \mathsf{test\_code} \qquad \mathsf{c}' :: \mathsf{test\_code} \qquad \mathbf{E}\ \mathsf{c} = \mathbf{E}\ \mathsf{c}'}{\mathsf{c} = \mathsf{c}'}\ (\mathsf{flat} - \mathsf{test})$$

where the $(1-1)$ rule guarantees that codes cost as few obs as possible, and the (flat) rules guarantee that every term $\mathsf{c} : \mathsf{code} :: \mathsf{test\_code}$ is the quotation $\{\mathsf{M}\}$ of some term M. However these rules are inconsistent. An example of terms M, N for which $\mathsf{M} = \mathsf{N}$ but $\{\mathsf{M}\} = \{\mathsf{N}\}$ is inconsistent will be provided by Gödel's second incompleteness theorem (discussed in detail later in 5.4). To achieve consistent coding principles that are still dense (nearly 1-1), we avoid self-referential terms by restricting the hypotheses in the ideal reasoning principles to *provable* equality and inequality.

The following axiom schemata will be enforced for the quoting operation $\{-\}$ and the atoms code, test_code, $\mathbf{A}$, $\mathbf{E}$. Note that since $\mathbf{E}$ is a left inverse for $\mathbf{Q}$ on codes, we do not need explicit axiom schemata regarding $\mathbf{Q}$.

quoting

$$\frac{}{\mathsf{M} = \mathbf{E}\{\mathsf{M}\}}\ (\mathbf{E} - \{-\}) \qquad \frac{}{\{\mathsf{M}\} : \mathsf{code}}\ (\{-\} - \mathsf{code}) \qquad \frac{}{\{\mathsf{M}\} :: \mathsf{test\_code}}\ (\{-\} - \mathsf{test})$$

code

$$\frac{\mathsf{x} :: \mathsf{test\_code} \qquad \mathsf{y} :: \mathsf{test\_code}}{\mathbf{E}(\mathbf{A}\ \mathsf{x}\ \mathsf{y}) = (\mathbf{E}\ \mathsf{x})(\mathbf{E}\ \mathsf{y})}\ (\mathbf{E} - \mathbf{A})$$

$$\frac{\mathsf{c} : \mathsf{code} :: \mathsf{test\_code} \qquad \mathsf{c}' : \mathsf{code} :: \mathsf{check\_code} \qquad \mathbf{E}\ \mathsf{c}' = \mathbf{E}\ \mathsf{c}}{\mathsf{c}\ \sqsupseteq\ \mathsf{c}'}\ (\mathsf{flat}-=)$$

$$\frac{\mathsf{c} : \mathsf{code} :: \mathsf{test\_code} \qquad \mathsf{c}' : \mathsf{code} \qquad \mathbf{E}\ \mathsf{c}'\ \not\sqsupseteq\ \mathbf{E}\ \mathsf{c}}{\mathsf{code}(\mathsf{c} \mid \mathsf{c}') = \top}\ (\mathsf{flat}-\not\sqsupseteq)$$

The quoting axioms $(\{-\})$ simply ensure that quoted terms are codes and evaluate to the terms they quote (so $\mathbf{E}$ is a left inverse for $\{-\}$). The $(\mathbf{E} - \mathbf{A})$ axiom maintains consistency between the code for a term and the codes for its subterms, relating them by application $\mathbf{A}$. The (flat) axioms ensure flatness of the code type, up to provable equality.

The quoting axioms describe exactly what we need to interpret universal quantification over codes, as in $\forall\{\mathsf{x}\}.\ \mathsf{x} = \mathbf{E}\{\mathsf{x}\}$ above. We generally want to interpret a sentence $\forall\{\mathsf{x}\}.\phi(\mathsf{x}, \{\mathsf{x}\})$ depending on both x and x's code[1] as a sentence $\forall\mathsf{c}.\psi(\mathsf{c})$ quantifying over codes c. The quoting axioms require $\mathsf{x} = \mathbf{E}\ \mathsf{c}$, $\mathsf{c} : \mathsf{code}$, and $\mathsf{c} :: \mathsf{test\_code}$, so it is enough to interpret

$$\forall\{\mathsf{x}\}.\ \phi(\mathsf{x}, \{\mathsf{x}\}) \qquad \Longleftrightarrow \qquad \forall \mathsf{c} : \mathsf{code} :: \mathsf{test\_code}.\ \phi(\mathbf{E}\ \mathsf{c}, \mathsf{c})$$

This provides semantics for universal quantification codes above, and allows us to universally close equations with variables that appear in quotes.

[1]higher codes can be attained by quoting.

To ensure provable extensionality, we need to reflect each of Johann's equational reasoning principles to a corresponding principle for quoted terms. We begin by explicitly identifying codes of β-equivalent terms.

!assume $\{\bot\ x\} = \{\bot\}$. *i.e.,* $\forall c{:}\mathsf{code}::\mathsf{test\_code}.\ \mathbf{A}\{\bot\}c = \{\bot\}$
!assume $\{\top\ x\} = \{\top\}$.
!assume $\{\mathbf{I}\ x\} = \{x\}$.
!assume $\{\mathbf{K}\ x\ y\} = \{x\}$.
!assume $\{\mathbf{F}\ x\ y\} = \{y\}$.
!assume $\{\mathbf{W}\ x\ y\} = \{x\ y\ y\}$.
!assume $\{\mathbf{B}\ x\ y\ z\} = \{x(y\ z)\}$.
!assume $\{\mathbf{C}\ x\ y\ z\} = \{x\ z\ y\}$.
!assume $\{\mathbf{S}\ x\ y\ z\} = \{x\ z(y\ z)\}$.

We also assume the bounded semilattice axioms for join

!assume $\{\mathbf{J}\ x\ x\} = \{x\}$.
!assume $\{\mathbf{J}\ x\ y\} = \{\mathbf{J}\ y\ x\}$.
!assume $\{\mathbf{J}\ x(\mathbf{J}\ y\ z)\} = \{\mathbf{J}(\mathbf{J}\ x\ y)z\}$.

and the defining properties of $\mathbf{Y}$ and $\mathbf{V}$.

!assume $\{\mathbf{Y}\ f\} = \{f(\mathbf{Y}\ f)\}$.
!assume $\{\mathbf{V}\ a\} = \{\mathbf{I} \mid (\mathbf{V}\ a)\circ a\}$.

Finally we need to reflect the axiom schemata hard-coded into Johann. These reflected axiom schemata are listed in Appendix B.

Characterization weakly extensional codes

What does does the code closure look like in this weaker theory of provable extensionality? By relaxing extensionality to only provable equality, we gave up true flatness, so that the ideal (flat) rules fail. To see the local effects of the relaxation, observe that for distinct M, N whose equality is unknown to Johann, the following are all distinct inhabitants of code.

$$\begin{tikzcd}
 & \top \arrow[d, dash] & \\
 & \{M\} \mid \{N\} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{M\} \arrow[dr, dash] & \{M \mid N\} \arrow[d, dash] & \{N\} \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Lemma 5.2.1. *If Johann can prove neither* $M = N$ *nor* $M \neq N$*, then*
(a) *the codes* $\{M\}$, $\{N\}$, $\{M \mid N\}$, *and* $\{M\} \mid \{N\}$ *are distinct total inhabitants of* code, *and*
(b) *they obey the order relations in the diagram above.*

Proof. In acquiring information, Johann can only join equivalence classes of code. Consider the four possibilities of acquiring information
(1) If $M = N$ then $\{M\} = \{N\} = \{M \mid N\} = \{M\} \mid \{N\}$.
(2) If $M \sqsubsetneq N$ then $\{M\}$, $\{N\} = \{M \mid N\}$ and $\{M\} \mid \{N\} = \top$ are distinct.
(3) If $M \sqsupsetneq N$ then $\{M\} = \{M \mid N\}$, $\{N\}$ and $\{M\} \mid \{N\} = \top$ are distinct.
(4) If $M \neq N$ are incomparable, then all four codes are distinct.

In all cases of complete knolwedge, $\{\mathsf{M} \mid \mathsf{N}\} \sqsubseteq \{\mathsf{M}\} \mid \{\mathsf{N}\}$, so this relation also holds under incomplete knowledge. However, since in case (4) all four codes are distinct, they must also be distinct in the partial information case. □

The minimal tested codes all arise as quotations of terms $\{\mathsf{M}\}$; thus inhabitants like $\{\mathsf{M}\} \mid \{\mathsf{N}\} : \mathsf{code} :: \mathsf{test\_code}$ are pathological.

Definition 5.2.2. A term $\mathsf{c} : \mathsf{code} :: \mathsf{test\_code}$ is *standard* iff it arises as the quotation of some term M, and *nonstandard* otherwise.

Lemma 5.2.3. $\mathsf{c} : \mathsf{code} :: \mathsf{test\_code}$ *is nonstandard iff there is another* $\mathsf{c}' : \mathsf{code} :: \mathsf{test\_code}$ *strictly below* c.

Even within standard codes, unbounded β-expansion leads to noncompact codes below every compact code, but evaluating to the same thing.

Example 5.2.4. An infinite **I**-expansion of any term

$$\mathsf{x} = \mathbf{I}\ \mathsf{x} = \mathbf{I}(\mathbf{I}\ \mathsf{x}) = \mathbf{I}(\mathbf{I}(\mathbf{I}\ \mathsf{x})) = \ldots$$

will evaluate to $\bot$, but code can't raise it to a tested semiset.

```
III = Y(A{I}).
!check III : code.
!check test_code III = ⊥.
!check (∀c::test_code. III | c :: test_code).
```

Nonstandard codes pose a potential problem to our interpretation of statements universally quantifying quoted variables

$$\forall\{\mathsf{x}\}.\ \phi\{\mathsf{x}\} \quad \Longleftrightarrow \quad \forall \mathsf{c} : \mathsf{code} :: \mathsf{test\_code}.\ \phi(\mathsf{c})$$

For example the flatness statement

$$\forall\{\mathsf{x}\}, \{\mathsf{y}\}.\ \{\mathsf{x}\} \mid \{\mathsf{y}\} :: \mathsf{test\_code} \quad \Longrightarrow \quad \mathsf{x} = \mathsf{y}$$

fails even though there are no provable witnesses of this failure. We can manually avoid such problems by never mentioning nonstandard codes like $\mathsf{code}(\{\mathsf{x}\} \mid \{\mathsf{y}\})$. In addition to allowing term discrimination, the flatness rules allow Johann to re-use the application indexing data structure to index also quoted terms. To see how this works, first consider the ideal situation. Under the ideal (flat) rules above, we could define $\{\mathsf{M}\}$ in terms of M, as the unique solution c to

$$\mathbf{E}\ \mathsf{c} = \mathsf{M} \qquad \mathsf{c} : \mathsf{code} \qquad \mathsf{c} :: \mathsf{test\_code}$$

Then Johann could avoid explicitly indexing quoting, and instead search for a solution to the above equations. In weaker extensionality with only provable equality, there will be additional nonstandard solutions c at any time. However only the standard solution will be the only *provable* solution, so we can still use the ideal indexing scheme.

Subalgebras of SKJO as subtypes of code

Later we will need to restrict some functions and quantifiers from SKJO codes to SKJ codes (e.g. the oracle inputs two codes $\mathbf{O}\{\mathsf{p}\}\{\mathsf{q}\}$, but we will require $\{\mathsf{p}\}$ to be SKJ -definable). To achieve a uniform quoting mechanism $\{-\}$ across multiple subalgebras, we will re-use the code closure, and vary only the test semipredicate for definability. Abstractly, we implement a definability test parametric in the subset

$\mathbb{B} \subseteq \{\mathbf{S}, \mathbf{K}, \mathbf{J}, \mathbf{O}, \text{code}\}$ generating the subalgebra (so a term is $\mathbb{B}$-definable iff it can be expressed using only atoms from $\mathbb{B}$).

```
test_subcode := (Test num → Test code) (
    λbasis.
    Yλt. test_Sset. test_Sum (test_Prod t t) (P_test test_num basis)
).
```

```
!check test_subcode b <:: test_code.
!check test_code = test_subcode ⊥.
!check test_code = test_subcode (I, I, I, I, I, ⊤).
```

Thus the whole algebra SKJO is given by the closure,test pair

```
skjo := code.
test_skjo := test_subcode (I, I, I, I, ⊥, ⊤).
```

In fact at any time code is SKJ -definable, hence SKJO -definable, so we have

```
!assume test_skjo = test_code.
```

Now for SKJ codes, we test whether there is an $\mathbf{O}$-free form.

```
skj := code.
```

```
!define test_skj = test_subcode (I, I, I, ⊥, ⊥, ⊤).
!assume test_skj := test_subcode (I, I, I, ⊥, I, ⊤).
!assume ( {S}, {K}, {J}, {code} :: test_skj ).
!assume test_skj (A x y) = and_semi (test_skj x) (test_skj y).
```

To express that $\mathbf{O}$ is not SKJ -definable, we assume that

```
!assume test_skj {O} = ⊥.
```

5.3 Injective codes for reflection

In this section we extend the $\langle \text{code}, \mathbf{A}\{-\}, \mathbf{E}, \mathbf{Q}, \{-\}\rangle$ structure to a rich type-indexed modal type structure

$$\langle \text{Code}\{-\},\ \text{Apply}\{-\}\{-\}\{-\},\ \text{Eval}\{-\},\ \text{Quote}\{-\},\ \text{Code}\{-\}\{-\}\rangle$$

combining the expressive power of codes and Simple-definable types. We show that the two structures are [untyped and typed, resp.] computational monads in the sense of Brookes and Geva [BG92].

A computational comonad for typed codes

Definition 5.3.1. A (definable) *functor* on types-as-closures is a pair $F = (F_0, F_1)$ of transformations of type

$$F_0 \ : \ \mathbf{V} \to \mathbf{V}$$
$$F_1 \ : \ \forall \text{a}:\mathbf{V}, \text{b}:\mathbf{V}.\ (\text{a} \to \text{b}) \ \to \ F_0 \ \text{a} \ \to \ F_0 \ \text{b}$$

perserving identity and composition

$$F_1 \ \text{a} \ \text{b} \ 1_\text{a} = 1_\text{b}$$
$$F_1 \ \text{a} \ \text{b} \ \text{f} \ ; \ F_1 \ \text{b} \ \text{c} \ \text{g} = F_1 \ \text{a} \ \text{c} \ (\text{f};\text{g})$$

Definition 5.3.2. A (definable) *natural transformation* between functors $\eta : F \to \mathsf{G}$ is an operation

$$\eta \ : \ \forall \mathsf{a} : \mathbf{V}. \ F_0 \ \mathsf{a} \ \to \ \mathsf{G}_0 \ \mathsf{a}$$

satisfying

$$F_1 \ \mathsf{a} \ \mathsf{b} \ \mathsf{f} \ ; \ \eta \ \mathsf{b} = \eta \ \mathsf{a} \ ; \ \mathsf{G}_1 \ \mathsf{a} \ \mathsf{b} \ \mathsf{f}$$

Comonads are dual to the more familiar structure of monads, in that arrows of η and μ are reversed

Definition 5.3.3. A (definable) *comonad* is a functor F_0, F_1 together with a pair of natural transformations $\eta : F \to 1, \mu : F \to F^2$ satisfying

$$\begin{array}{ll}
\mu \ \mathsf{a} \ ; \ \eta(F_0 \ \mathsf{a}) = 1_{F_0 \ \mathsf{a}} & \textit{left-identity} \\
\mu \ \mathsf{a} \ ; \ F_1(\eta \ \mathsf{a}) = 1_{F_0 \ \mathsf{a}} & \textit{right-identity} \\
\mu \ \mathsf{a} \ ; \ \mu(F_0 \ \mathsf{a}) = \mu \ \mathsf{a} \ ; \ F_1(\mu \ \mathsf{a}) & \textit{associativity}
\end{array}$$

(which are generalizations of the comonoid axioms – just squint).

Monads ([Wad90], [Mog91]) and comonads ([BG92], [Geh95]), with a little extra structure come up frequently in functional programming, though true comonads appear only in the context of semantics.

Definition 5.3.4. A (definable) *computational comonad* is a comonad $(F_0, F_0, \eta, \mu, \gamma)$ with an additional natural transformation $\gamma : 1 \to F$ satisfying

$$\begin{array}{l}
\gamma \ \mathsf{a} \ ; \ \eta \ \mathsf{a} = 1_{\mathsf{a}} \\
\gamma \ \mathsf{a} \ ; \ \mu \ \mathsf{a} = \gamma \ \mathsf{a} \ ; \ \gamma \ (F_0 \ \mathsf{a})
\end{array}$$

Brooks and Geva [BG92] use a comonad to capture intensional semantics in an extensional system. There is also a history of comonadic type structure in staged compilation of functional languages (e.g. [DP96], [WLPD98]) In these applications, the comonad captures an idea of neutralization of computations, and is often called a code, eval, quote comonad.

Definition 5.3.5. The type-indexed *code comonad*

$$\langle \mathsf{Code}\{-\}, \ \mathsf{Apply}\{-\}\{-\}\{-\}, \ \mathsf{Eval}\{-\}, \ \mathsf{Quote}\{-\}, \ \mathsf{Code}\{-\}\{-\} \rangle$$

now has components

$$\begin{array}{|lll}
\mathsf{Code} & := & (\mathsf{code} \ \to \ \mathbf{P} \ \mathsf{code}) \ (\lambda \mathsf{a}, \mathsf{x}. \ \mathbf{A} \ (\mathbf{A} \ \{\mathbf{V}\} \ \mathsf{a}) \ \mathsf{x}). \\
\mathsf{Apply} & := & (\forall \{\mathsf{a} : \mathbf{V}\}, \{\mathsf{b} : \mathbf{V}\}. \ \mathsf{Code}\{\mathsf{a} \to \mathsf{b}\} \ \to \ \mathsf{Code}\{\mathsf{a}\} \ \to \ \mathsf{Code}\{\mathsf{b}\}) \ (\lambda -, -. \ \mathbf{A}). \\
\mathsf{Eval} & := & (\forall \{\mathsf{a} : \mathbf{V}\}. \ \mathsf{Code}\{\mathsf{a}\} \ \to \ \mathsf{a}) \ (\lambda - . \ \mathbf{E}). \\
\mathsf{Quote} & := & (\forall \{\mathsf{a} : \mathbf{V}\}. \ \mathsf{Code}\{\mathsf{a}\} \ \to \ \mathsf{Code}\{\mathsf{Code}\{\mathsf{a}\}\}) \ (\lambda - . \ \mathbf{Q}).
\end{array}$$

where Code$\{\mathsf{a}\}$ is the action of the functor on the object a, Apply$\{\mathsf{a}\}\{\mathsf{b}\}\{\mathsf{f}\}$ is the action of the functor on the arrow $\mathsf{f} : \mathsf{a} \to \mathsf{b}$, Eval$\{-\}$ and Quote$\{-\}$ form the comonadic pair of natural transformations, and the typed quoting operation Code$\{-\}\{-\}$ constructs computations.

The components of the code comonad are not definable as functions, so some of the arrows in the comonad conditions will not be realized by SKJO -terms (because of the undefinability of quoting $\{-\}$). However the relations still hold at each objects a, b and arrows f, g. This contrasts the case of, e.g., the Sset monad, where Sset a and the other structure is all SKJO -definable (see 3.7).

Having assumed in 5.2 the computational comonad conditions for the unityped, fully-definable comonad $(\mathsf{code}, \mathbf{A}\{-\}, \mathbf{E}, \mathbf{Q}, \{-\})$, we now verify the computational comonad conditions for the indexed version.[2]

[2]Keep in mind the semantics of quantified variables:

$$\forall \{\mathsf{x}\}. \ \phi(\mathsf{x}, \{\mathsf{x}\}) \quad \Longleftrightarrow \quad \forall \mathsf{c} : \mathsf{code} :: \mathsf{test\_code}. \ \phi(\mathbf{E} \ \mathsf{c}, \mathsf{c})$$

These equations were arrived at by straight-forward annotation of the 15 comonad axioms for the unityped theory, which were in turn arrived at by reversing the arrows in Wadler's very lucid exposition [Wad90] of computational monads.

```
the type-indexed identity morphism
Id  :=  (∀a:V. a→a) (λ− . I).
!check Id = V.
```

```
functoriality of (Code{−}, Apply{−}{−}{−})
!check (∀{a:V}.   Code{a}  :  V)
!check (∀{a:V}, {b:V}, {f:a→b}.   Apply{a}{b}{f}  :  Code{a}  →  Code{b}).
!check (∀{a:V}.   Apply{a}{a}{Id a} = Id(Code{a})).
!check (∀{a:V}, {b:V}, {c:V}, {f:a→b}, {g:b→c}.
      Apply{a}{b}{f} ; Apply{b}{c}{g} = Apply{a}{c}{f; g}
).
```

```
naturality of Eval{−}{−} and Quote{−}{−}
!check (∀{a:V}.   Eval{a}  :  Code{a}  →  a).
!check (∀{a:V}.   Quote{a}  :  Code{a}  →  Code{Code{a}}).
!check (∀{a:V}, {b:V}, {f:a→b}.   Apply{a}{b}{f} ; Eval{b} = Eval{a} ; f).
!check (∀{a:V}, {b:V}, {f:a→b}.
      Apply{a}{b}{f} ; Quote{b}
            =  Quote{a} ; Apply{Code{a}}{Code{b}}{Apply{a}{b}{f}}
).
```

```
comonad conditions
!check (∀{a:V}.   Quote{a} ; Eval{Code{a}} = Code{a}).
!check (∀{a:V}.   Quote{a} ; Apply{a}{Code{a}}{Eval{a}} = Code{a}).
!check (∀{a:V}.
      Quote{a} ; Quote{Code{a}} = Quote{a} ; Apply{a}{Code{a}}{Quote{a}}
).
```

```
computation conditions
!check (∀{a:V}, {x:a}.   Code{a}{x}  :  Code{a}).
!check (∀{a:V}, {b:V}, {f:a→b}, {x:a}.
      Apply{a}{b}{f}(Code{a}{x}) = Code{b}{f x}
).
!check (∀{a:V}, {x:a}.   Eval{a}(Code{a}{x}) = x).
!check (∀{a:V}, {x:a}.   Quote{a}(Code{a}{x}) = Code{Code{a}}{Code{a}{x}}).
```

Finally, as with every polymorphic type, we also define tests, checks, and joins for Code

```
test_Code  :=  (∀{a : V}. Test a → Test(Code{a})) (
     λ−, t, {x} :: test_code. t x
).
!check test_code = test_Code {any} I.

check_Code  :=  (∀{a : V}. Check a → Check(Code{a})) (
     λ−, t, {x} :: check_code. t x
).
!check check_code = check_Code {any} I.

Join_code  :=  Sset code (
     Yλj. (jλx. jλy. ⟨A x y⟩) | ⟨{S}⟩ | ⟨{K}⟩ | ⟨{J}⟩ | ⟨{O}⟩ | ⟨{code}⟩
).
Join_Code  :=  (∀{a : V}. Sset. Code{a}) (λa. Join_code λx. ⟨A a x⟩).
!check Join_code = Join_Code {I}.

Join_skj  :=  Sset skj (Join_code λc :: test_skj. ⟨c⟩).
```

Domain-specific quoting functions

As mentioned earlier, the quoting arrow $\mathsf{a}\to\mathsf{Code}\{\mathsf{a}\}$ is missing from general comonads, but is present in computational comonads for some restricted subsets of a, e.g. as $\{-\}$ for closed terms of type a. The arrow is also present for code types quote_code $= \mathbf{Q}$ and flat domains e.g. numeral systems, though only the total inhabitants are correctly quoted. We provide quoting functions for polynomial datatypes in general. Each quoter will have type

```
Quoter  :=  (code → V) (∀{a : V}. a → Code{a}).
```

For example the atomic types

```
quote_semi  :=  Quoter{semi} ⟨{I}⟩.
!check test_semi = test_code∘quote_semi;
!check quote_semi I = {I}.
```

```
quote_bool  :=  Quoter{bool} ⟨{K}, {F}⟩.
!check test_bool = test_code∘quote_bool;
!check quote_bool K = {K}.
!check quote_bool F = {F}.
```

and basic type constructors

```
quote_Maybe  :=  (∀{a : $\mathbf{V}$}. Quoter{a} → Quoter{Maybe a}) (
    λ−, q. ({none}, inr∘$q_b$)
).
quote_Sum  :=  (
    ∀{a : $\mathbf{V}$}, {b : $\mathbf{V}$}. Quoter{a} → Quoter{b} → Quoter{Sum a b}
) (
    λ−, −, $q_a$, $q_b$. ⟨inl∘$q_a$, inr∘$q_b$⟩
).
quote_Prod  :=  (
    ∀{a : $\mathbf{V}$}, {b : $\mathbf{V}$}. Quoter{a} → Quoter{b} → Quoter{Prod a b}
) (
    λ−, −, $q_a$, $q_b$, (x, y). {x} := $q_a$ x. {y} := $q_a$ y. {(x, y)}
).
```

We also provide quoters specifically for numerals, naturals and SKJ -terms.

```
quote_num  :=  Quoter{num} ($\mathbf{Y}$ quote_Maybe).
!check test_num = test_code∘quote_num;
!check quote_num none = {none}.
!check (∀{n :: test_num}. quote_num(some n) = {some n}).
```

```
quote_nat  :=  Quoter{nat} ($\mathbf{A}${succ}, {zero}).
!check test_nat = test_code∘quote_nat;
!check quote_nat 0 = {0}.
!check (∀{n :: test_nat}. quote_nat(succ n) = {succ n}).
```

```
quote_term  :=  Quoter{term} ($\mathbf{Y}$λq. (
    λ(x, y). $\mathbf{A}$ ($\mathbf{A}$ {ap} (q x)) (q y), {$\underline{\mathbf{S}}$}, {$\underline{\mathbf{K}}$}, {$\underline{\mathbf{J}}$}
).
!check test_term = test_code∘quote_term;
!check quote_term $\underline{\mathbf{S}}$ = {$\underline{\mathbf{S}}$}.
!check quote_term $\underline{\mathbf{K}}$ = {$\underline{\mathbf{K}}$}.
!check quote_term $\underline{\mathbf{J}}$ = {$\underline{\mathbf{J}}$}.
!check quote_term $\underline{\top}$ = {$\underline{\top}$}.
!check (∀{x :: test_term}, {y :: test_term}. quote_term(ap x y) = {ap x y}).
```

Since codes can be thought of as equivalence classes of terms, we can define a lifting function from term to skj

```
term_to_code  :=  (term → code) (
    $\mathbf{Y}$λt2c. (λ(x, y). $\mathbf{A}$(t2c x)(t2c $\mathbf{Y}$), {$\mathbf{S}$}, {$\mathbf{K}$}, {$\mathbf{J}$}, {⊤})
).
```

and an inverse from code→Sset term. Since code is not statically SKJ -definable, we must postulate its definition as a term.

```
!using code_as_term
!assume code_as_term : term
!assume code_as_term :: test_term.
code_to_term  :=  (skj → Sset term) (
    $\mathbf{Y}$λc2t. ⟨
        λ(x, y). c2tλx. ct2λy. ⟨ap x y⟩,
        $\underline{\mathbf{S}}$, $\underline{\mathbf{K}}$, $\underline{\mathbf{J}}$, ⊥, code_as_term, error
    ⟩
).
```

Now we can relate the quoting operation on terms to the quoting operations on codes

```
!check (λ⟨x⟩. ⟨quote_term t⟩)∘code_to_term ⊑ code_to_term∘Q.
```

Fixed-point theorems

In this section we formulate some basic results from classical recursion theory in terms of the code comonad. See [Rog67], [Odi92], for classical presentations,[3] [Bar84] section 6.5 for presentation in untyped λ-calculus, and [CS88] for presentation in a typed λ-calculus.

Kleene's first fixed-point theorem (the recursion theorem), plus uniqueness

Theorem 5.3.6. *(Kleene, Böhm, van der May)*
(a) $\forall f.\exists x.\ x = f(x)$*; and uniformly,*
(b) $\exists \mathbf{Y}.\forall f.\ \mathbf{Y}\ f = f(\mathbf{Y}\ f)$*; moreover*
(c) the $\mathbf{Y}$ *in part (b) is unique modulo* $\mathcal{H}^*$

Proof. (a) follows from (b).
(b) Letting $\mathbf{Y} = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$, the equation follows from β equivalence.
(c) Proof in [Bar84], Lemma 6.5.3.

□

Kleene's second fixed point theorem, for quoted fixedpoints of SKJO terms

Theorem 5.3.7. *(Kleene)*
(a) $\forall f.\exists\{x\}.\ x = f\{x\}$*; and uniformly,*
(b) $\exists \mathbf{Y}'.\forall\{f\}.\ \mathbf{Y}'\{f\} = f\{\mathbf{Y}'\{f\}\}$.

Proof. (a) follows from (b).
(b) We simply annotate with quotes the proof from the first fixed point theorem. Recall the syntactic sugar for quoting

$$\lambda\{x\},\{y\}.\ \{x\ y\} = \mathbf{A}$$
$$\lambda\{x\}.\ \{\{x\}\} = \mathbf{Q}$$
$$\lambda\{x\}.\ x = \mathbf{E}$$

Consider the following (purely mechanical) annotation steps

$$\begin{array}{lll}
\mathbf{Y} = \lambda f.\ (\lambda x.\ f.\ x\ x)\ (\lambda x.\ f.\ x\ x) & & \\
\quad\rightsquigarrow\ \lambda\{f\}.\ (\lambda x.\ f.\ x\ x)\ (\lambda x.\ f.\ x\ x) & & \mathbf{Y}' \textit{ inputs a code} \\
\quad\rightsquigarrow\ \lambda\{f\}.\ (\lambda x.\ f\{x\ x\})\ (\lambda x.\ f\{x\ x\}) & & f \textit{ inputs a code} \\
\quad\rightsquigarrow\ \lambda\{f\}.\ (\lambda\{x\}.\ f\{x\ x\})\ (\lambda\{x\}.\ f\{x\ x\}) & & x \textit{ is quoted} \\
\quad\rightsquigarrow\ \lambda\{f\}.\ (\lambda\{x\}.\ f\{x\ x\})\ \{\lambda\{x\}.\ f\{x\ x\}\} & & \{x\} \textit{ is a code} \\
\quad\rightsquigarrow\ \lambda\{f\}.\ (\lambda\{x\}.\ f\{x\{x\}\})\ \{\lambda\{x\}.\ f\{x\{x\}\}\} & & x \textit{ inputs a code} \\
\quad =:\ \mathbf{Y}' & &
\end{array}$$

Now checking

```
!define Y' := (λ{f}. (λ{x}. f{x{x}}) {λ{x}. f{x{x}}}).
!check Y'{f}
    = (λ{x}. f{x{x}}) {λ{x}. f{x{x}}}
    = f{(λ{x}. f{x{x}}) {λ{x}. f{x{x}}} }
    = f{Y'{f}}.
```

as required.

□

[3]Warning: our notation {M} for quoting terms is opposite to Kleene's notation $\{\phi\}$ for *evaluating* codes for partial recursive functions.

Question 5.3.8. *Does uniqueness still hold for* $\mathbf{Y}'$*?*

Following [DP96], we can also take a modal logic perspective. Letting $\Box\mathsf{a} = \mathsf{Code}\{\mathsf{a}\}$ denote the comonadic type constructor for a moment, we see that the Code comonad's terms have types corresponding to the (S4) axioms of modal logic (see 5.4). To these we can further add a typed fixedpoint Fix corresponding to the irreflexivity axiom or Löb's theorem,[4] to achieve the typings

Code{a}{x}	:	$\Box\mathsf{a}$	*necessitation*
Apply a b	:	$\Box(\mathsf{a}\to\mathsf{b})\to\Box\mathsf{a}\to\Box\mathsf{b}$	*distribution*
Eval a	:	$\Box\mathsf{a}\to\mathsf{a}$	*reflexivity*
Quote a	:	$\Box\mathsf{a}\to\Box\Box\mathsf{a}$	*transivity*
Fix a	:	$\Box(\Box\mathsf{a}\to\mathsf{a})\to\Box\mathsf{a}$	*irreflexivity = Löb's theorem*

Fix is defined in terms of $\mathbf{Y}'$ by quoting the result of $\mathbf{Y}'$.

```
Fix  :=  (∀{a:V}. Code{Code{a}→a}  →  Code{a}) (λ−, {f}. {Y′{f}}).
!check  (∀{a:V}, {f:a}. Fix{a}{f}  :  Code{a}.
!check  (∀{a:V}. Fix{a}{f} = {f{Fix{a}{f}}}.
```

Later in 5.4, Fix will be the main tool in proving Löb's theorem.

Provable equality

Our extensional code type identifies provably equivalent terms, and thus allows introspection into Johann's current theory. For example, we can define semipredicates for *provable* equality and information ordering, employing the ($\mathbf{E}-=$) rule

```
if_pr_equal  :=  P  C  (code  →  code  →  semi) (
    Yλe. ⟨(
        λx, y. ⟨(λx′, y′. and_semi  (e  x  x′)  (e  y  y′),  ⊥)⟩,
        λn. ⟨(⊥,  eq_num  n)⟩
    )⟩
).
!check  if_pr_equal  x  y  ⊑  and_semi  (test_code  x)  (test_code  y).
!check  (  {S}, {K}, {J}, {O}, {code}, {x}  ::  W  if_pr_equal  ).
```

```
if_pr_less  :=  (code  →  code  →  semi)  (λ{x}, {y}. if_pr_equal  {x | y}  {y}).
!check  if_pr_less  x  y  ⊑  and_semi  (test_code  x)  (test_code  y).
!check  (  {S}, {K}, {J}, {O}, {code}, {x}  ::  W  if_pr_less  ).
!check  (  ⟨{⊥}, {x}⟩,  ⟨{x}, {x}⟩,  ⟨{x}, {x | y}⟩,  ⟨{x}, {⊤}⟩  ::  ⟨if_pr_less⟩  ).
```

Negation is a little trickier, but we can define it in terms of code, employing the rule ($\mathbf{E}-\not\sqsubseteq$)

```
if_pr_nequal  :=  P  C  (code  →  code  →  semi)  (
    λ{x}, {y}.  if_pr_equal  {⊤}  {  code({x} | {y})  }
).
!check  if_pr_nequal  x  y  ⊑  and_semi  (test_code  x)  (test_code  y).
```

```
if_pr_nless  :=  (code  →  code  →  semi)  (
    λ{x}, {y}.  if_pr_equal  {⊤}  {  code({x | y} | {y})  }
).
!check  if_pr_nless  x  y  ⊑  and_semi  (test_code  x)  (test_code  y).
!check  I = if_pr_nless  {⊤}  {⊥}.
```

[4]Note that these axioms are inconsistent for modal logic, as reflexivity and irreflexivity together imply $\Box\mathsf{a}$ for any a.

Combining (positive,negative) pairs of tests, we can define boolean-valued introspective predicates for equality and Scott ordering.

```
pr_equal := P C (code → code → bool) (
    λx, y. if_pr_equal x y true | if_pr_nequal x y false
).
!check pr_equal x y ⊑ and_semi (test_code x) (test_code y).
!check if_pr_equal x y = if (pr_equal x y).
!check if_pr_nequal x y = if∘not (pr_equal x y).
```

```
pr_less := (code → code → bool) (
    λx, y. if_pr_less x y true | if_pr_nless x y false
).
!check pr_less x y ⊑ and_semi (test_code x) (test_code y).
!check if_pr_less x y = if (pr_less x y).
!check if_pr_nless x y = if∘not (pr_less x y).
```

At any time the value of such a predicate will be provably true if Johann has proved the result, provably false if Johann has disproved the result, and undecidable otherwise. In particular, pr_equal x y will never be proved to be bottom (for total x, y); that way Johann leaves room for future assumptions/axioms.

We will see stronger predicates later in 5.6, including a generalized provability logic.

5.4 Provability and reflection

Previously in 5.3 we saw how to decide provable equality between quoted SKJO terms. At the term {true} this gives us a decision procedure for provability of general boolean statements.

```
pr := (Code{bool} → bool) (λ{x::test_bool}. pr_equal {true} {x}).
!check pr ⊑ pr_equal{true}.
!check (∀{φ::test_bool}. pr{φ} :: test_bool).
```

Th provability predicate pr{−} is defined by the bi-directional reasoning principle

$$\frac{\mathsf{pr}\{\phi\} = \mathsf{true}}{\{\phi\} = \{\mathsf{true}\}}\ (\mathsf{pr} - \mathsf{def})$$

Soundness of pr{−} (as Johann's theory) can be expressed by the two reasoning principles $\frac{\mathsf{pr}\{\phi\} = \mathsf{true}}{\phi = \mathsf{true}}$ and $\frac{\phi = \mathsf{false}}{\mathsf{pr}\{\phi\} = \mathsf{false}}$. While the former holds by definition

```
!check pr{true} = true.
```

the latter pr{false} = false cannot be proven or even !assumed. In fact this is Gödel's second incompleteness theorem.

The provability predicate pr{−} models the provability logic GL (see [Ver08], [JJ98]). GL is the modal logic satisfying Löb's theorem (as above), the distribution rule

```
!check (∀{φ:bool::test_bool}, {ψ:bool::test_bool}. true =
    implies ( pr{implies φ ψ} ).
    implies ( pr{φ} ).
    pr{ψ}
).
```

and the inference rule $\dfrac{\phi = \mathsf{true}}{\mathsf{pr}\{\phi\} = \mathsf{true}}$ (but not its converse). This rule follows from the definition of pr and the extensionality principle for code $\dfrac{\vdash \phi = \psi}{\vdash \{\phi\} = \{\psi\}}$.

Transitivity is a theorem of GL; hence GL extends K4.

```
| !check (∀{φ:bool::test_bool}. true = implies (pr{φ}) (pr{pr{φ}})).
```

Having reflected Johann's hard-coded reasoning principles in B, we can also assume the converse of transitivity

```
| !check (∀{φ:bool::test_bool}. pr{φ} = pr{pr{φ}}).
```

and justly refer to pr{−} as Johann's theory.

Using the provability predicate pr{−}, we can relativize some classical incompleteness theorems from PA to code (see [Smo77] and [BBJ07] for statements of the theorems in these forms).

We begin with the diagonalization lemma, which in our context is a special case of Kleene's second fixed point theorem (from 5.3). Let $\vdash \phi$ denote $\mathsf{pr}\{\phi\} = \mathsf{true}$, and consider predicates of quoted booleans

```
| pred_qbool := V (Code{bool} → bool).
| test_pred_qbool := Test pred_qbool (
|     λφ. and_semi (φ{true}) (φ{false})
| ).
```

Theorem 5.4.1. *(Diagonalization lemma)*

(a) For every total predicate $\phi\{-\}$ on codes of sentences, there is a total sentence ψ satisfying $\vdash \psi \leftrightarrow \phi\{\psi\}$ (i.e., $\mathsf{pr}\{\mathsf{iff}\ \psi\ (\phi\{\psi\})\} = \mathsf{true}$).

(b) The ψ in part (a) is uniformly definable.

Proof. (a) follows from (b).

(b) We apply $\mathbf{Y}'$ from Kleene's second fixed-point theorem

$$\psi := \mathbf{Y}'\{\phi\} \qquad \Longrightarrow \qquad \mathbf{Y}'\{\phi\} = \phi\{\mathbf{Y}'\{\phi\}\}$$

And since Johann knows this, we get the stronger pr{−} version

```
| !check (
|     ∀{φ:pred_qbool::test_pred_qbool}.  let {ψ} := {Y'{φ}}.
|     ψ = φ{ψ}     AND     pr{iff ψ (φ{ψ})} = true
| ).
```

□

An easy consequence of the diagonalization lemma is Gödel's first incompleteness theorem for Johann's theory.

Theorem 5.4.2. *(Gödel's first incompleteness theorem) There is a sentence ϕ such that neither $\vdash \phi$ nor $\vdash \mathsf{not}\ \phi$.*

In terms of decision,

```
| decides := (Code{bool} → semi) (if∘pr | if∘pr∘(A{not})).
| !check decides ⊑ test_Code {bool} test_bool.
| !check decides <:: test_Code {bool} test_bool.
```

The theorem states that Johann does not decide everything

```
decides ⋣ test_Code {bool} test_bool.
decides !::> test_Code {bool} test_bool.
```

(but Johann cannot prove these exactly; this is Gödel's second incompleteness theorem).

Proof. Applying the diagonalization lemma, let $\phi := \mathbf{Y}'\{\mathsf{not}\circ\mathsf{pr}\}$ so that $\phi = \mathsf{not}\circ\mathsf{pr}\{\phi\}$. Then assuming either $\vdash \phi$ or $\vdash$ not ϕ leads to contradiction

$$\begin{array}{rcll} \vdash \phi & \iff & \vdash \mathsf{pr}\{\phi\} & \textit{by strong transitivity} \\ & \iff & \vdash \mathsf{not}\ \ \phi & \textit{by definition} \end{array}$$

Formally checking,

```
!check (φ := Y′{not∘pr}.  pr{φ} = pr{pr{φ}} = pr{not φ}).
```

and assuming consistency, we can prove the contradiction

```
inconsistent  :=  pr{false}.
consistent  :=  not∘pr{false}.
!check (φ := Y′{not∘pr}.  pr{φ} = inconsistent = pr{not φ}).
```

□

Note that it is inconsistent for Johann to !assume consistent = true. Also, the witness ϕ cannot be statically SKJO -definable; since every SKJO -definable sentence can eventually be decided (by !assuming additional equations, e.g. the sentence itself). Thus ϕ must be theory- or time-dependent, by mentioning code.

Gödel's second incompleteness theorem observes that the final !check above gives us a simpler form for the witness ϕ.

Theorem 5.4.3. *(Gödel's second incompleteness theorem) If Johann's theory is consistent, then it cannot prove its own consistency (and also the trivial converse):* consistent = not∘pr{consistent}

I.e., consistent = not∘pr{false} is the fixed point of not∘pr{−}.

Proof. Continuing from above,

```
!check (φ  :=  Y′{not∘pr}.
     inconsistent = pr{φ}    AND               by above
     consistent = not∘pr{φ}                    by negation
                 =  not∘pr{not∘pr{φ}}          by definition of φ
                 =  not∘pr{consistent}         by above
).
```

□

To state Löb's theorem, we briefly use modal logic notation, with $\Box$ denoting pr{−}.

Theorem 5.4.4. *(Löb's theorem) For any sentence ϕ, if* $\vdash$ pr{ϕ} $\implies$ ϕ *then* $\vdash \phi$*; or in modal notation* $\Box(\Box\phi \to \phi) \to \Box\phi$. *Relying on the trivial converse allows a simpler assumption*

```
!check (∀{φ : bool :: test_bool}. pr{implies (pr{φ}) φ} = pr{φ}).
```

I.e., $\Box\phi$ is the fixed point of $\Box(- \longrightarrow \phi)$.

Proof. Applying the diagonalization lemma, let $\psi = \mathbf{Y}'\{\mathsf{implies}\ (\mathsf{pr}\{\psi\})\ \phi\}$, so that $\psi = \mathsf{imples}\ (\mathsf{pr}\{\psi\})\ \phi$.

1.	$\vdash \psi \leftrightarrow (\Box\psi \to \phi)$	*diagonalization lemma*
2.	$\vdash \psi \to \Box\psi \to \phi$	*1*
3.	$\vdash \Box(\psi \to \Box\psi \to \phi)$	*2: informal transitivity*
4.	$\vdash \Box\psi \to \Box(\Box\psi \to \phi)$	*3: distributivity*
5.	$\vdash \Box\psi \to \Box\Box\psi \to \Box\phi$	*4: distributivity*
6.	$\vdash \Box\psi \to \Box\Box\psi$	*formal transitivity*
7.	$\vdash \Box\psi \to \Box\phi$	*5,6*
8.	$\vdash (\Box\phi \to \phi) \to \Box\psi \to \phi$	*7*
9.	$\vdash (\Box\phi \to \phi) \to \psi$	*8,1*
10.	$\vdash \Box(\Box\phi \to \phi) \to \Box\psi$	*9: distributivity*
11.	$\vdash \Box(\Box\phi \to \phi) \to \Box\phi$	*7,10*

□

Gödel's second theorem and Löb's theorems are instances of a more general fixed-point principle for $\mathsf{pr}\{-\}$-guarded predicates.

Theorem 5.4.5. *(de Jongh, Sambin 1975) Let* $\mathsf{a}:\mathsf{Code}\{\mathsf{bool}\}\to\mathsf{bool}$ *be a predicate expression such that in* $\mathsf{a}\{\phi\}$, ϕ *only occurs inside* $\mathsf{pr}\{-\}$ *(i.e. is "guarded" by* $\mathsf{pr}\{-\}$*). Then there is a* $\mathsf{b}:\mathsf{Code}\{\mathsf{bool}\}\to\mathsf{bool}$ *with*

$$\forall\{\psi:\mathsf{bool}::\mathsf{test\_bool}\}.\ \mathsf{pr}\{\ \mathsf{iff}\ \ (\mathsf{b}\{\psi\})\ \ (\mathsf{a}\{\mathsf{b}\{\psi\}\}\{\psi\})\ \ \}.$$

Proof. See [BBJ07].

Note that we can achieve the general fixed-point theorem already with $\mathbf{Y}'$, so that Sambin's and de Jongh's result guarantees definability in the guarded case.

5.5 Axioms for a hyperarithmetic oracle (O)

In this section we implicitly define an oracle **O** that answers hyperarithmetic questions with boolean values. The purpose of adding such an oracle to Johann's language is to gain expressiveness. The language SKJ can express computable functions, and equations in SKJ essentially express totality of computable functions. By allowing access to a hyperarithmetic oracle, we can express essentially arbitrary statements in predicative set theory ([Fef05]). With a language complete for implicit definability, the task of verifying more complex problems is limited to the question of what inference rules to add.

The study of computability w.r.t. oracles has a long history (e.g. Rogers' comprehensive [Rog67]) but is often confused by the definition of oracles as sets with two-sided membership decidability (the exception being enumeration degrees). This approach leads to various hierarchy theorems, which obfuscate the purpose of adding an oracle: to find a system closed under stronger notions of computability.

For example, it is classical theorem that the boolean implicit-definability operation has not fixedpoint

Theorem 5.5.1. *The operation* $\mathsf{jump}:(\mathsf{Pred\ \ skj}\ \ \to\ \ \mathsf{Pred\ \ skj})$ *defined*

```
jump  o = (Pred  skj  →  Pred  skj)  (
    λ{p:(Pred  skj  →  Pred  skj)  →  pred}::test_skj.
    pₒ  :=  p  o.                                   let p use the oracle
    (Join_skj  λx. check_bool(pₒ  x)).              check p for errors
    [∀{x}:test_skj.  pₒ  {x}]                        quantify over codes
)
```

has no total well-tested fixedpoint.

Proof. Consider the term

$$\mathsf{p}\ \ :=\ \ (\mathbf{Y}'\lambda\{\mathsf{q}\},\mathsf{o},\{\mathsf{x}\}.\ \mathsf{implies}\ \ (\mathsf{eq\_skj}\{\mathsf{x}\}\{\mathsf{q}\}).\ \mathsf{not}(\mathsf{o}\ \ \{\mathsf{q}\})).$$

Then $\mathsf{jump\ \ o}\ \ \{\mathsf{p}\} = \mathsf{not}(\mathsf{o}\ \ \{\mathsf{p}\})$. □

However, our order-conscious approach embraces a variety of truth values bool, semi, unit, div. We show in this section that, formulating the same problem with semiboolean truth values, the implicit-definability operation does have a fixedpoint, a language closed under implicit semidefinability. This language is SKJO
.

In posing the implicit definability problem, we used a flat domain of codes. The definition of **O** works with any old flat type of codes, e.g. Gödel numerals. But since **O** respects equivalence of codes modulo their evaluation, it would waste space for codes to be perfectly intensional. This observation is the motivation for our extensional flat codes.

Implicit definition of the hyperarithmetic oracle

Statements in brackets $[\phi]$ are taken at meta-mathematical value, with bool truth values: true = $\mathbf{K}$, false = $\mathbf{F}$. Later in section 5.6, we will show how to compile a fragment of this informal logic to SKJO terms.

We define an oracle $\mathbf{O}$ as the least fixed point of the meta-equation

```
O = (skj → code → semi) (
        λ{s:test}::test_skj.          contravariant, so SKJ-definable
        λ{t:test}::test_skjo.         covariant, so allow self-reference
          if [∃x::s. t x ⊒ ⊤] ⊤
        | if [∀x::s. t x ⊒ I] I
    )
```

where $\mathbf{O}$ appears implicitly on the right hand side in evaluating the SKJO-code $\mathsf{t}:\mathsf{test}$, and the meta-level statements $[\exists \mathsf{x}::\mathsf{s}.\ \mathsf{t}\ \mathsf{x} \sqsupseteq \top]$ and $[\forall \mathsf{x}::\mathsf{s}.\ \mathsf{t}\ \mathsf{x} \sqsupseteq \mathbf{I}]$ are interpreted as a booleans. Note since the first argument $\mathsf{s}:\mathsf{test}$ is $\mathbf{O}$-free, the right hand side is increasing in $\mathbf{O}$, so by the Tarski-Knaster fixed-point theorem, $\mathbf{O}$ is well-defined. Moreover, since $\mathbf{O}$ operates on codes, it is a join of SKJ-terms. Hence the Böhm-tree approximation theorem from 5 carries over, and with it the correctness proofs of Simple and all closures from SKJ.

What is the logical strength of $\mathbf{O}$? At each step, $\mathbf{O}$ transforms a Π^0_2-complete problem $\mathsf{s} <:: \mathsf{t}$ to a Δ^0_2 problem $\mathbf{O}\{\mathsf{s}\}\{\mathsf{t}\} = \mathbf{I}$ (see 3.9), but since $\mathbf{O}$ may occur in t, this step can be iterated arbitrarily often.

Definition 5.5.2. A predicate $\phi:\mathsf{nat}\to\mathsf{bool}$ is *bool-hard* (or just *hard*) for a problem class $\mathsf{X} \subseteq \mathbb{N}$ iff X can be bool-decided by ϕ, i.e. $\phi\ \mathsf{n} = (\mathsf{if}\ \ \mathsf{n} \in \mathsf{X}\ \ \mathsf{then}\ \ \mathbf{K}\ \ \mathsf{else}\ \ \mathbf{F})$. A semipredicate $\psi:\mathsf{nat}\to\mathsf{semi}$ *semi-hard* for a problem class X if every X can be semi-decided by ψ, i.e. $\psi\ \mathsf{n} = (\mathsf{if}\ \ \mathsf{n} \in \mathsf{X}\ \ \mathsf{then}\ \ \mathbf{I}\ \ \mathsf{else}\ \ \bot)$.

Theorem 5.5.3. *$\mathbf{O}$ is Π^1_1-semi-complete.*

Proof. (upper-bound) Let $\phi(-)$ be the Π^0_2 predicate for satisfaction of the above fixed-point equation. Then $\mathbf{O}$ is the unique solution to the Π^1_1 predicate defining the *least* solution of $\phi(-)$, w.r.t. the Π^0_2 ordering problem $\mathsf{o} \sqsubseteq \mathsf{o}'$

$$\phi(\mathsf{o})\quad \mathbf{AND}\quad \forall \mathsf{o}'.\ \phi(\mathsf{o}') \quad\Longrightarrow\quad \mathsf{o} \sqsubseteq \mathsf{o}'$$

(hardness) It is enough to semidecide the well-foundedness of recursive (SKJ -definable) countably-branching trees:

```
if_wfdd_tree := (skj → semi) (
    λ{φ:Test(Test num)}. [           given an SKJ-code φ,
        ∀s:Test num.                 for each semipredicate s,
        ∀n::test_num. n::s   ⟹       if s is total
        s::φ                         then s::φ
    ]
).
```

In particular, it suffices to show that the meta-statement $[\ldots]$ can be interpreted as an SKJO term. Thus, working with SKJ-codes,

```
if_wfdd_tree := (skj → semi) (
    Y'λ{t}. {
        λ{φ:(num→semi)→semi}::test_skj.
        or_semi (φ ⊥).
        O {test_num} {t {(I, φ∘succ)}}
    }
).
```

□

Question 5.5.4. *How hard is equality in* SKJO *?*

Lemma 5.5.5. SKJO *equality is* Δ^1_2.

Proof. By simply expanding the definitions of $\sqsubseteq$ and $\mathbf{O}$, we can put each relation $x \sqsubseteq y$ between SKJO terms in the form

$$\forall_1 z.\ \left(\exists_2 o.\ \phi_x(o,z)\right) \implies \left(\exists_2 o.\ \phi_y(o,z)\right)$$

where the second-order existentials quantifiers range over Böhm trees o approximating the oracle $\mathbf{O}$, and the ϕ are arithmetic relations encoding statements "o satisfies the fixed point eqation for $\mathbf{O}$, and particular $\langle \mathbf{S}, \mathbf{K}, \mathbf{J}, o\rangle$-term converges". Putting this statement in Kleene normal form yields a Δ^1_2 formula. □

Axioms and axiom schemata

We start with some typing and testing axioms

```
!using O.
!assume O : skj → code → semi.
!assume O : (∀p:Code{test}. Code{P_test p} → semi).
!assume O p q = and_semi (test_skj p) (test_code q) (O p q).
```

and some properties

```
!assume O p p = test_skj p.               regardless of whether E p:test
!assume O{p}{q} = O{p}{P_test p q}.
```

Following 5.3, we consider as special cases the most commonly used flat domains: num, nat, term.

Example 5.5.6. Subtests of coalgebraic numerals.

```
O_num := O {test_num}.
!assume O_num {test_num} = I.
!assume (
    div_at := (λm, n. if∘not (equal_num m n)).     converges except at m
    ∀{n:nat}. O_num {div_at n} = ⊥
).
```

Example 5.5.7. Subtests of Church numerals.

```
O_nat := O {test_nat}.
!assume O_nat {test_nat} = I.
```

Example 5.5.8. Subtests of SKJ terms.

```
O_term := O {test_term}.
!assume O_term {test_term} = I.
```

Example 5.5.9. Subtests of codes.

```
O_code := O {test_code}.
!assume O_code {test_code} = I.
```

Example 5.5.10. Subtests of indexed Codes.

```
O_Code := (λ{a:V}, {t:Test a}. O {test_Code{a}t}).
!assume (∀{a:V}, {t:Test a}. O_Code {test_code{a}t} = I).
```

Example 5.5.11. Subtests of SKJ -codes.

```
O_skj := O {test_skj}.
!assume O_skj {test_skj} = I.
!check O_skj ::> O_code.
```

To prove properties of general tests, we need more general reasoning principles. Thus the following axiom schemata will be enforced for the atom $\mathbf{O}$:[5]

pass/fail/error

$$\frac{\{p\} :: \mathsf{test\_skj} \qquad p <:: q}{\mathbf{O}\{p\}\{q\} = \mathbf{I}}\ (\mathbf{O} - \mathbf{I}) \qquad \frac{\{p\} :: \mathsf{test\_skj} \qquad x :: p \qquad q\ x = \bot}{\mathbf{O}\{p\}\{q\}\ :\ \mathsf{div}}\ (\mathbf{O} - \bot)$$

$$\frac{\{p\} :: \mathsf{test\_skj} \qquad x :: p \qquad q\ x \not\sqsubseteq \mathbf{I}}{\mathbf{O}\{p\}\{q\} = \top}\ (\mathbf{O} - \top)$$

variance

$$\frac{\{p\} :: \mathsf{test\_skj} \qquad p <:: p'}{\mathbf{O}\{p\} ::> \mathbf{O}\{p'\}}\ (\mathbf{O}-\ ::>) \qquad \frac{q <:: q'}{\mathbf{C}\ \mathbf{O}\{q\} <:: \mathbf{C}\ \mathbf{O}\{q'\}}\ (\mathbf{C}\ \mathbf{O}-\ <::) \qquad \frac{p \sqsubseteq p'}{\mathbf{C}\ \mathbf{O}\{p\} \sqsupseteq \mathbf{C}\ \mathbf{O}\{p'\}}\ (\mathbf{O}-\ \sqsupseteq)$$

$$\frac{q \sqsubseteq q'}{\mathbf{C}\ \mathbf{O}\{q\} \sqsubseteq \mathbf{C}\ \mathbf{O}\{q'\}}\ (\mathbf{O}-\ \sqsubseteq)$$

The pass/fail/error rules reflect Johann's knowledge of subtesting $p <:: q$ to the oracle, as a quoted truth values: $\mathbf{O}\{p\}\{q\} = \mathbf{I}$ for subtests, either $\bot$ or $\top$ (i.e. some value: div) for failed subtests. and $\top$ for subtesting errors. The variance rules ($\mathbf{O}-\ ::>$) and ($\mathbf{O}-\ <::$) rules state that $\mathbf{O}$ is contravariant and covariant in its first and second arguments, both with respect to the subtest ordering $<::$ and the information ordering $\sqsubseteq$.

Are these axioms and schemata enough for successful verification? How much do we need to assume to "reasonably" approximate $\mathbf{O}$? It turns out that we can axiomatize Δ^1_1-much of $\mathbf{O}$ using only induction (via the above schemata) and $\mathcal{H}^*$.

Theorem 5.5.12. *(hardness) The partial axiomatization above, together with the locally Π^0_2-complete $\mathcal{H}^*$ is Δ^1_1-bool-hard.*

Proof. By classic theorem of Kleene ([Kle55]), the Δ^1_1 predicates are exactly the hyperaritmetic functions from a countable domain to booleans. The hyperarithmetic functions are exactly those functions definable by induction over recursively well-orderings. If we can locally decide $\mathcal{H}^*$, then each stage of the oracle $\mathbf{O}$'s definition can be decided by the $\mathbf{O}$-axiom schemata. By induction over computable well-orderings, this allows us to compute arbitrary hyperarithmetic functions. □

Knowing this strength result of our $\mathbf{O}$ axioms allows us to focus scientific effort (finding new axioms) on $\mathcal{H}^*$.

In fact this is all of $\mathbf{O}$ that is meaningfully observable, in the sense of refutation in the limit. The following adapt the work of Kelly and Schulte [KS95] to our setting of equational theories of SKJO .

Definition 5.5.13. Let a *hypothesis* be a set of equational theories (of say SKJO). A singleton hypothesis is called *emprically complete*. A hypothesis is *refutable in the limit* iff, as a set of sets of equations, it is Π^0_2.

[5] These schemata leverage the schemata for $-$: div in 3.8, and the schemata for $\mathbf{P}_{\mathsf{test}}$ and the subtest relation $<::$ in 3.11.

A problem is reducible to a hypothesis iff it is uniformly reducible to the theories in the hypothesis; thus we can speak of the logical strength of an hypothesis as if it were a single partial theory.

Theorem 5.5.14. *(limit of observability) No refutable-in-the-limit hypothesis with Π^1_1-hard conclusions can be empirically complete.*

As a corollary of the completeness of axiomatization, we also have

Corollary 5.5.15. *There is a refutable-in-the-limit hypothesis that has Δ^1_1-hard predictions.*

Question 5.5.16. *Is* SKJO *the full Böhm-tree model of* SKJ *in predicative set theory? (I.e, the model of Böhm trees whose nodes are arbitrary joins of head normal forms)*

5.6 The logic of SKJ terms

In this section we develop first-order logic fo SKJ -terms with basic relations for equality and Scott's information ordering. This builds on the boolean and semiboolean propositional logics we developed in 3.9. Now in SKJO , with a Δ^1_1-bool-hard oracle in the language, we can even evaluate the truth-value of all such sentences that are meaningful.[6]

[6]Meaningful in the generalized Popperian sense of [KS95]: predicted by hypotheses that are refutable in the limit.

Recall from 3.10 that each type of conditional truth value unit, unit $\vdash$ semi, semi, semi $\vdash$ bool corresponds to a complete complexity class, respectively Π^0_1, Σ^0_1, Δ^0_2, Δ^0_1. By adding the Π^1_1-complete oracle $\mathbf{O}$ in the way we did, with truth-values of semi, the complexity theorem extends precisely to a hyperarithmetic analog.

Theorem 5.6.1. *The following complexity characterizations hold for local problems in* SKJO .[7]

	Context $\vdash$ *Problem*	*Complexity*
(a)	$x{:}\mathsf{div} \vdash x = \bot$	Σ^1_1*-complete*
(b)	$x{:}\mathsf{div} \vdash x = \top$	Π^1_1*-complete*
(c)	$x{:}\mathsf{unit} \vdash x = \mathbf{I}$	Π^1_1*-complete*
(d)	$x{:}\mathsf{semi} \vdash x = \mathbf{I}$	*complete for differences between* Π^1_1 *problems*
(e)	$x{:}\mathsf{semi}{::}\mathsf{unit} \vdash x = \mathbf{I}$	Π^1_1*-complete*
(f)	$x{:}\mathsf{bool}{::}\mathsf{test\_bool} \vdash x = \mathbf{K}$	Δ^1_1*-hard and* Π^1_1

Note that in contrast to the case of SKJ where s <:: t was Π^0_2-complete while x::t was only Δ^0_2, in SKJO the oracle reduces exactly these problems, so that s <:: t $\iff$ $\mathbf{O}\{s\}\{t\}::\mathbf{I}$, as in case (c).

Proof. We begin with cases (a) and (c), since they are closest to the definition of $\mathbf{O}$. The functions $\lambda x.x \mid \mathbf{I}:\mathsf{div}\to\mathsf{unit}$ and $\lambda x.x\ \bot:\mathsf{unit}\to\mathsf{div}$ provide translations between (a) and (c), so we show hardness for (c) and prove the upper bound for (a)

(c) We showed in 5.5 that $\mathbf{O}$ is hard for Π^1_1 questions, using truth values $\{\bot, \mathbf{I}\}$.

(a) Note that the definition of $\mathbf{O}$ is Π^1_1, so convergence of SKJO terms can be at most Π^1_1.

Now all other cases variously reduce to case (c).

(b) This is just the negation of part (a).

(d) Suppose $x{:}\mathsf{semi}$. Then $x = \mathbf{I}$ iff unit $x = \mathbf{I}$ and div $x = \top$. Conjoining these is a differenc of Π^1_1-complete problems.

(e) Suppose $x{:}\mathsf{semi}{::}\mathsf{unit}$. Then $x \in \{\bot, \mathbf{I}\}$. The function $\lambda x.\mathbf{I} \mid x\ \top{:}\mathsf{semi}\to\mathsf{unit}$ reduces this to part (b). We already showed hardness in 5.5.

(f) For hardness, suppose ϕ is a Δ^1_1 statement. Then it and its negation can both be posed as Π^1_1 problems to $\mathbf{O}$, say as if_phi, if_not_phi : semi :: unit. Now define $[\phi]$ = if_phi $\mathbf{K}$ $\mid$ if_not_phi $\mathbf{F}$. Then $[\phi]$ :: test_bool since ϕ either true or false, but not both..

For , observe that for $x{:}\mathsf{bool}{::}\mathsf{test\_bool}$, we know $x \in \{\mathbf{K}, \mathbf{F}\}$. Then $x = \mathbf{K}$ iff div$(x\ \bot\ \top) = \bot$.

$\square$

Basic relations

We can tests for basic equality and ordering between SKJ terms using the proof theory of $\mathcal{H}^*$ demonstrated in 6.1. First we define the semi-valued testing versions

```
| lift_term := (Test term → Test skj) (λt, x::test_code. code_to_term x t).
```

```
| if_conv := (skj → semi) (lift_term if_conv_term).
| if_div := (skj → semi) (
|     lift_term λx. {x} := quote_term x. O {test_Conv x} {⊥}
| ).
```

[7]Recall from 3.9 our notation for contexts corresponding to binding of universally quantified variables:

$$\begin{aligned}(x{:}a\ \vdash\ f\ x = g\ x) &\iff f{\circ}a\ x = g{\circ}a\ x\\ (x{::}t\ \vdash\ f\ x = g\ x) &\iff (t\ x)(f\ x) = (t\ x)(g\ x)\\ (x{:}a{::}t\ \vdash\ f\ x = g\ x) &\iff (t{\circ}a\ x)(f{\circ}a\ x) = (t{\circ}a\ x)(f{\circ}a\ x)\end{aligned}$$

```
if_less := (skj → skj → semi) (
    conv_at := (skj → skj → semi) (λ{z}, {f}. if_conv {f z}).
    λ{x}, {y}. O {conv_at{x}} {conv_at{y}}
).
if_equal := P C (skj → skj → semi) (
    λ{x}, {y}. and_semi (if_less{x}{y}) (if_less{y}{x})
).
if_nless := (skj → skj → semi) (
    λ{x}, {y}.
    Join_skj λ{f}. and_semi (if_conv {f x}) (if_div {f y})
).
```

Next we package (positive,negative) pairs together as boolean valued predicates.

```
conv := (skj → bool) (
    λx. if_conv x true | if_div x false
).
less := (skj → skj → bool) (
    λx, y. if_less x y true | if_nless x y false
).
equal := P C (skj → skj → bool) (
    λx, y. if_equal x y true | if_nless x y false | if_nless y x false
).
```

These relate to the semidecidable versions as expected.

```
!check if_conv x = if (conv x).
!check if_div x = if∘not (conv x).
!check if_less x y = if (less x y).
!check if_nless x y = if∘not (less x y).
!check if_equal x y = if (equal x y).
!check equal x y = and (less x y) (less y x).
```

Finally, we give an alternative definition of provable equality, exploiting the fact that code is at any time SKJ -definable, hence quoted terms are SKJ -definable, hence quoted terms are separable by a boolean predicate. That is, two terms are provably equal iff their codes are provably equal.

```
!check pr_equal = P C (code → code → bool) (λ{x}, {y}. equal{{x}}{{y}}).
```

Quantification

In 3.9 we defined logical connectives and, or for booleans and and_semi, or_semi for semibooleans. In SKJ we can even eliminate semiboolean existential quantifiers over flat domains with a semiset of total terms. For example from num : **V** and Join_num : Sset num we can define[8]

```
if_exists_num := Test (Test num) (λφ. Join_num λn. φ n).
!check if_exists_num = semi∘Join_num.
```

Now in SKJO , the oracle **O** allows us to eliminate also semiboolean universal quantifiers over flat domains with totality tests. For example, given num : **V** and *a code* {test_num} : Code{Test num} for the totality test

[8]Recall Test a = a → semi, and Pred a = a → bool.

we can eliminate quantifiers of *quoted* tests with

```
if_forall_num  :=  Test∘Code{Test num} (λ{φ}. O{test_num}{λn. φ n}).
!check if_forall_num  ⊒  O{test_semi}.
```

To iterate these universal quantifiers requires a little extra structure, however, since an outer bound variable must be quoted to appear in an inner quantified predicate, e.g.

```
if_forall_num{λn. if_forall{λm. φ m n}}        error: n cannot be quoted!
```

We can work around this in our typical flat domains, by using a quoter $\mathsf{quote}_\mathsf{a} : \mathsf{a} \to \mathsf{Code}\{\mathsf{a}\}$, as e.g.

```
if_forall_num{λn. let{n} := quote_num n. if_forall{λm. φ m n}}     well-defined
```

But it would be more convenient to wrap the quoter into the quantifier, so that we could simply bind quoted terms. This motivates an improved definition

```
if_forall_num′  :=  Test∘Code{ Test∘Code{num} } (
    λ{φ}. O{test_num}{λn. φ∘quote_num n}
).
```

which we can iterate using the easier notation

```
if_forall_num{λ{n}. if_forall{λ{m}. φ m n}}
```

Now given any flat domain a with a semiset $\mathsf{Join}_\mathsf{a} : \mathsf{Sset\ a}$ of total terms, a totality test $\mathsf{test}_\mathsf{a} : \mathsf{Test\ a}$, and a quoter $\mathsf{quote}_\mathsf{a} : \mathsf{Quoter\ a}$, we can package up an (existential,universal) pair of semiboolean quantifiers to eliminate boolean quantifiers. We will do this uniformly, defining a polymorphic type

```
Quantifier  :=  (Code{V} → V) (λa. Pred∘Code{Pred∘Code a}).
```

and functions to build semiboolean quantifiers and package them into boolean pairs

```
Exists_  :=  (
    ∀{a:V}. Sset a → Code{Test{a}} → Code{Quoter{a}} → Quantifier{a}
) (
    λ−, j, {t}, {q}.  j if∘φ∘q true  |  O{t}{if∘not∘φ∘q} false
).
Forall_  :=  (
    ∀{a:V}. Sset a → Code{Test{a}} → Code{Quoter{a}} → Quantifier{a}
) (
    λ−, j, {t}, {q}.  O{t}{if∘φ∘q} true  |  j if∘not∘φ∘q false
).
```

```
exists_  :=  Exists_ ⊥.
forall_  :=  Forall_ ⊥.
```

For example we can define quantifiers over our standard flat domains.

Example 5.6.2. Quantifying over numerals.

```
exists_num  :=  Exists_ {num} Join_num {test_num} {quote_num}.
forall_num  :=  Forall_ {num} Join_num {test_num} {quote_num}.
!check (∀{φ:Pred num}. not∘forall_num {φ} = exists_num {not∘φ}).
!check (∀{φ:Pred num}. not∘exists_num {φ} = forall_num {not∘φ}).
```

We can use this to state a structure decomposition theorem for total inhabitants

```
!check true = forall_num {λ{n}.
    or (eq_num none n). exists_num {λ{n′}. eq_num n. some n′}
}.
```

Example 5.6.3. Quantifying over SKJ terms.

```
forall_term := Forall_ {term} Join_term {test_term} {quote_term}.
exists_term := Exists_ {term} Join_term {test_term} {quote_term}.
!check (∀{φ:Pred term}. not∘forall_term {φ} = exists_term {not∘φ}).
!check (∀{φ:Pred term}. not∘exists_term {φ} = forall_term {not∘φ}).
```

Example 5.6.4. Quantifying over SKJ codes.

```
exists_skj := Exists_ {skj} Join_skj {test_skj} {Q}.
forall_skj := Forall_ {skj} Join_skj {test_skj} {Q}.
!check (∀{φ:Pred skj}. not∘forall_skj {φ} = exists_skj {not∘φ}).
!check (∀{φ:Pred skj}. not∘exists_skj {φ} = forall_skj {not∘φ}).
```

As an example application, we can check that SKJ has binary meets, using three levels of binding

```
!check true =
    forall_skj {λ{x}.
      forall_skj {λ{y}.
        below_xy := (skj → bool) (λz. and (less z x) (less z y)).
        exists_skj {λ{z}.
          and (below_xy x).
          forall_skj {λ{z′}. impiles (below_xy z′) (less z′ z) }
    } } }.
```

and check the equivalence of the skj and term quantifiers

```
!check forall_term{φ∘term_to_code∘E} = forall_skj{φ∘E}.
```

Example 5.6.5. Quantifying over SKJO codes.

```
exists_code := Exists_ {code} Join_code {test_code} {Q}.
forall_code := Forall_ {code} Join_code {test_code} {Q}.
!check (∀{φ:Pred code}. not∘forall_code {φ} = exists_code {not∘φ}).
!check (∀{φ:Pred code}. not∘exists_code {φ} = forall_code {not∘φ}).
```

Example 5.6.6. Quantifying over indexed Codes.

```
exists_Code := (∀{a:V}. Quantifier{Code{a}}) (
    λ{a:V}. Exists_ {Code{a}} (Join_Code{a}) {test_code} {Quote{a}}
).
forall_Code := (∀{a:V}. Quantifier{Code{a}}) (
    λ{a:V}. Forall_ {Code{a}} (Join_Code{a}) {test_code} {Quote{a}}
).
!check (∀{a:V}, {φ:Pred.Code{a}}.
    not (forall_Code{a}{φ}) = exists_Code{a}{not∘φ}
).
!check (∀{a:V}, {φ:Pred.Code{a}}.
    not (exists_Code{a}{φ}) = forall_Code{a}{not∘φ}
).
```

In 6.2 we present an extended example using quantifiers over a variety of flat domains.

5.7 Types using reflection

Our main tools in SKJ were types-as-closures (in 3.7) and tests (in 3.10). In SKJO we have already studied a variety of newly definable tests (in 5.6). In this section we define new types-as-closures that we could not define in SKJ .

Principle types

Recall from 3.7 that

Definition 5.7.1. The principle type $\mathsf{unit}_{\mathsf{M}}$ of a term M is the closure with two inhabitants $\{\mathsf{M}, \top\}$.

Theorem 5.7.2. *All principle types of* SKJ *-terms are definable in* SKJO *. Moreover, principle types are uniformly definable by a type constructor* $\mathsf{Unit}:\mathsf{code\_SKJ}\to\mathbf{V}$ *with* $\mathrm{inhab}(\mathsf{Unit}\{\mathsf{x}\}) = \{\mathsf{x}, \top\}$.

Proof. Let x be an SKJ -term. Since $\mathsf{a\ x} \sqsubseteq \mathsf{x}$ is uniformly semidecidable in SKJO , we can join over $\{\mathsf{a} \mid \mathsf{a\ x} \sqsubseteq \mathsf{x}\}$. □

```
Unit := (skj → V) (λ{x}::test_skj. Join_skj λ{a}. if_less{a x}{x} a).
!check (
    ∀{x}::test_skj. x:Unit{x}   AND
    ∀{y}::test_skj. Unit{x}y = x    OR    Unit{x}y = ⊤
).
```

For example,

```
!check unit = Unit{I}.
!check nil = Unit{⊤}.
!check div = Unit{⊥}.
```

Question 5.7.3. *Which principle types of* SKJO *-terms are* SKJO *-definable?*

Quotient types

Recall from 3.7 that we can define quotient types modulo Σ^0_1 equational theories. Now with $\mathbf{O}$ in our langauge, we can extend this to Π^1_1 theories. E.g. we can define a quotient of SKJ terms modulo the Π^0_2 theory $\mathcal{H}^*$.

```
skj_Hstar := P code (λx. Join_skj λy. if_equal x y y).
!check {skj_Hstar} !:: test_skj.
```

However this is not as useful as our SKJO -codes modulo Johann's equality, since Johann also has a theory of how **O** behaves.

Chapter 6

Examples and Applications

6.1 Proving properties of reduction in SKJ

This section is a case study illustrating the use of join in proving theorems, under the Curry-Howard interpretation of theorems as closure,test pairs. The main tool being illustrated is the use of *proof sketching*, a form of type-inference whereby a theorem-as-closure can raise a proof sketch without indices to a total proof with indices.

To see how type inference works, consider a simple theorem ”$\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J}\ \mathbf{J} \twoheadrightarrow \mathbf{K}\ \mathbf{J}\ \mathbf{J}$”, which follows from the proof sketch “follows from transitivity: apply the **K** rule to the LHS, then the **K** rule again” A more thorough proof would mention the intermediate terms

1.	$\mathbf{K}\ \mathbf{K}\ \mathbf{S} \twoheadrightarrow \mathbf{K}$	*by* **K** *rule*
2.	$\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J} \twoheadrightarrow \mathbf{K}\ \mathbf{J}$	*by left-monotonicity: 1*
3.	$\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J}\ \mathbf{J} \twoheadrightarrow \mathbf{K}\ \mathbf{J}\ \mathbf{J}$	*by left-monotonicity: 2*
4.	$\mathbf{K}\ \mathbf{J}\ \mathbf{J} \twoheadrightarrow \mathbf{J}$	*by* **K** *rule*
5.	$\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J}\ \mathbf{J} \twoheadrightarrow \mathbf{J}$	*by transitivity: 3 then 4*

Let us abbreviate terms using quotation marks for a moment, so that “$\mathbf{K}\ \mathbf{K}\ \mathbf{S}$” $= \mathsf{ap}(\mathsf{ap}\ \underline{\mathbf{K}}\ \underline{\mathbf{K}})\underline{\mathbf{S}}$. Formally, we want to prove a theorem

$$\mathsf{thm} := \mathsf{Red}\ \text{“}\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J}\ \mathbf{J}\text{”}\ \text{“}\mathbf{J}\text{”}.$$

with a proof sketch

$$\mathsf{pf} := \mathsf{trans}_\mathsf{r}\ (\mathsf{lhs}_\mathsf{r}\ (\mathsf{lhs}_\mathsf{r}\ \mathbf{K}_\mathsf{r}))\ \mathbf{K}_\mathsf{r}.$$

Now the join operation allows the theorem-as-closure to raise the proof to

$$\begin{aligned}\mathsf{thm}\ \mathsf{pf} = \mathsf{Trans}_\mathsf{r}\ &\text{“}\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J}\ \mathbf{J}\text{”}\ \text{“}\mathbf{J}\text{”}\\ &(\mathsf{Lhs}_\mathsf{r}\ \text{“}\mathbf{K}\ \mathbf{K}\ \mathbf{S}\ \mathbf{J}\text{”}\ \text{“}\mathbf{K}\ \mathbf{J}\text{”}\ \text{“}\mathbf{J}\text{”}\\ &\quad(\mathsf{Lhs}\ \text{“}\mathbf{K}\ \mathbf{K}\ \mathbf{S}\text{”}\ \text{“}\mathbf{K}\text{”}\ \text{“}\mathbf{J}\text{”}\\ &\qquad\mathbf{K}_\mathsf{r}\ \text{“}\mathbf{K}\text{”}\ \text{“}\mathbf{S}\text{”}\qquad))\\ &(\mathbf{K}_\mathsf{r}\ \text{“}\mathbf{J}\text{”}\ \text{“}\mathbf{J}\text{”})\end{aligned}$$

(we will not use this quotation mark notation again).

Reduction theorems for derived terms

From the reduction axioms for $\mathbf{S}, \mathbf{K}, \mathbf{J}, \top$ in 3.14, we can derive reduction theorems for other defined combinators, e.g., $\mathbf{I}, \mathbf{B}, \mathbf{C}, \mathbf{W}, \mathbf{Y}$.

The identity $\mathbf{I} = \mathbf{S}\ \mathbf{K}\ \mathbf{K}$ reduces via $\mathbf{I}\ \mathsf{x} \twoheadrightarrow \mathsf{x}$.

```
I   :=  ap (ap S K) K.
I_R  :=  (
      ∀x. Red (ap I x) x
) ( λ− .
      trans_r S_r.               S K K x  ↠  K x(K x)
      K_r                                 ↠  x
).
I_r  :=  I_R ⊥.
```

Right projection $\mathbf{F} = \mathbf{K}\ \mathbf{I}$ reduces via $\mathbf{F}\ \mathsf{x}\ \mathsf{y} \twoheadrightarrow \mathsf{y}$.

```
F   :=  ap K I.
F_R1  :=  (
      ∀x. Red (ap F x) I
) ( λ− .
      K_r                        K I x  ↠  I
).
F_e1  :=  F_R1 ⊥.
F_R  :=  (
      ∀x, y. Red (ap(ap F x)y) y
) ( λ−, −.
      trans_r (lhs_r F_e1).      K I x y  ↠  I y
      I_r                                 ↠  y
).
F_r  :=  F_R ⊥ ⊥.
```

Composition $\mathbf{B} = \mathbf{S}(\mathbf{K}\ \mathbf{S})\mathbf{K}$ reduces via $\mathbf{B}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{x}(\mathsf{y}\ \mathsf{z})$.

```
B   :=  ap (ap S (ap K S)) K.
compose  :=  (term → term → term) (λf, g. ap (ap B f) g).
```

```
B_R  :=  (
      ∀x, y, z. Red (ap(ap(ap B x)y)z) (ap x(ap y z))
) ( λ−, −, −.
      trans_r (2 lhs_r.
            trans_r S_r.                     B x  ↠  K S x(K x)
            lhs_r K_r                             ↠  S(K x)
      ).
      trans_r S_r.                           B x y z  ↠  K x z(y z)
      lhs_r K_r                                       ↠  x(y z)
).
B_r  :=  B_R ⊥ ⊥ ⊥.
```

Transposition $\mathbf{C} = \mathbf{S}(\mathbf{B}\ \mathbf{B}\ \mathbf{S})(\mathbf{K}\ \mathbf{K})$ reduces via $\mathbf{C}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{x}\ \mathsf{z}\ \mathsf{y}$.

```
C   :=  ap (ap S (ap (ap B B) S)) (ap K K).
transpose  :=  ap C.
```

```
C_R  :=  (
     ∀x, y, z.  Red  (ap(ap(ap  C)x)y)z)  (ap(ap  x  z)y)
) (
     λ−, −, −.
     trans_r  (lhs_r.
          trans_r  (lhs_r.
               trans_r  S_r.                          C  x   ↠   B  B  S  x(K  K  x)
               trans_r  (lhs_r  B_r).                        ↠  B(S  x)(K  K  x)
               rhs_r  K_r                                    ↠  B(S  x)K
          ).
          B_r                                         C  x  y   ↠   S  x(K  y)
     ).
     trans_r  S_r.                                    C  x  y  z   ↠   x  z(K  y  z)
     rhs_r  K_r                                                 ↠   x  z  y
)
C_r   :=   C_R  ⊥  ⊥  ⊥.
```

The diagonal $\mathbf{W} = \mathbf{C}\ \ \mathbf{S}\ \ \mathbf{I}$ reduces via $\mathbf{W}\ \ x\ \ y \twoheadrightarrow x\ \ y\ \ y$.

```
W   :=   ap  (ap  C  S)  I.
W_R   :=   (
     ∀x, y.  Red  (ap(ap  W  x)y)  (ap(ap  x  y)y)
) (
     λ−, −.
     trans_r  (lhs_r  C_r).                 W  x  y   ↠   S  x  I  y
     trans_r  S_r.                                    ↠   x  y(I  y)
     rhs_r  I_r                                       ↠   x  y  y
).
W_r   :=   W_R  ⊥  ⊥.
```

Turing's fixed-point combinator $\mathbf{Y} = \mathbf{B}(\mathbf{S}\ \ \mathbf{I})(\mathbf{W}\ \ \mathbf{I})(\mathbf{B}(\mathbf{S}\ \ \mathbf{I})(\mathbf{W}\ \ \mathbf{I}))$ satisfies the fixed-point equation $\mathbf{Y}\ \ f \twoheadrightarrow f(\mathbf{Y}\ \ f)$. (note that this is more often denoted Θ).

```
Y   :=   W  ap  (ap(ap  B(ap  S  I))(ap  W  I)).
rec   :=   ap  Y.
```

```
Y_R   :=   (
     ∀f.  Red  (ap  Y  f)  (ap  f(ap  Y  f))
) (  λ − .
     trans_r  (lhs_r.
          trans_r  B_r.                  Y   ↠   S  I(W  I(B(S  I)(W  I)))
          rhs_r.                             ↠   S  I(I(B(S  I)(W  I))(B(S  I)(W  I)))
          trans_r  (lhs_r  W_r).             ↠   S  I(B(S  I)(W  I)(B(S  I)(W  I)))
          I_r
     ).
     trans_r  S_r.                       Y  f   ↠   I  f(Y  f)
     lhs_r  I_r                                 ↠   f(Y  f)
).
Y_r   :=   Y_R  ⊥.
```

A bottom element $\perp = \mathbf{Y}\ \ \mathbf{K}$ is constant $\perp\ \ x \twoheadrightarrow \perp$.

```
Bot   :=   rec  K.
Bot_r   :=   (∀x.  Red  (ap  Bot  x)  Bot)  (λ − .  trans_r  (lhs_r  Y_r)  K_r).
bot_r   :=   Bot_r  Bot.
```

Althought we added a top element to the basis in 3.14, a top element $\top = \mathbf{Y}\ \mathbf{J}$ with the same reduction rule $\top\ x \twoheadrightarrow x$ is now definable.

```
⊤′    :=  rec J.
Top′_r  :=  (∀x. Red (ap ⊤′ x) ⊤′) (λ− . trans_r (lhs_r Y_r) J_r1).
top′_r  :=  Top_r ⊥.
Top_e  :=  (∀x. Red (ap ⊤′ x) x) (λ− . trans_r (lhs_r Y_r) J_r2).
top_e  :=  Top_e ⊥.
```

The probe div $= \mathbf{Y}(\mathbf{J}\ \mathbf{F}(\mathbf{C}\ \mathbf{C}\ \top))$ can reduce in either of the two ways: div $x \twoheadrightarrow$ div $x\ \top$ or div $x \twoheadrightarrow x$.

```
div    :=  rec (join F. 2 (ap C) ⊤).
probe  :=  ap div.
```

```
Div1  :=  (
    ∀x. Red (probe x) x
) ( λ− .
    trans_r (lhs_r.
        trans_r Y_r.          div  ↠  J F(C C ⊤)div
        lhs_r J_r1.                ↠  F div
    ).
    F_r                       div x  ↠  x
).
div1  :=  Div1 ⊥.
```

```
Div2  :=  (
    ∀x. Red (probe x) (ap(probe x)⊤)
) ( λ− .
    trans_r (lhs_r.
        trans_r Y_r.                 div  ↠  J F(C C ⊤)div
        trans_r (lhs_r J_r2).             ↠  C C ⊤ div
        c_r                               ↠  C div Top
    ).
    C_r                              div x  ↠  div x Top
).
div2  :=  Div2 ⊥.
```

Convergence proofs

Convergence proofs (see 3.14) generally take the form of feeding some term some number of $\top$'s as arguments, and then showing the result reduces to $\top$.

```
feed  :=  (∀x:term, n:nat. Red (n(c ap ⊤)x) ⊤ → Conv x) (
    λ−, n. n next; done
).
feed  :=  feed ⊥.
```

For example

```
top_c  :=  Conv ⊤ (feed 1. t_r).     t t  ↠  t
k_c  :=  Conv K (feed 2. k_r).       k t t  ↠  t
j_c  :=  Conv J (feed 2. j_1).       j t t  ↠  t
i_c  :=  Conv I (feed 1. i_r).       i t  ↠  t
```

```
w t t  ↠  t t t  ↠  t t  ↠  t
w_c  :=  Conv W (feed 2. trans_r w_r. trans_r (lhs t_r). t_r).

b t t t  ↠  t(t t)  ↠  t
b_c  :=  Conv B (feed 3. trans_r b_r. t_r).

c t t t  ↠  t t t  ↠  t t  ↠  t
c_c  :=  Conv C (feed 3. trans_r c_r. trans_r (lhs t_r). t_r).

s t t t  ↠  t t(t t)  ↠  t(t t)  ↠  t
s_c  :=  Conv S (feed 3. trans_r s_r. trans_r (lhs t_r). t_r).
```

We can also prove the handy lemmas: reduction respects converence

```
resp_c  :=  (
    ∀x:term, y:term. Red x y → Conv y → Conv x
) (
    yλr. λx, y, xy. (
        trans_r xy; done,                              x  ↠  y!
        r (ap x ⊤) (ap y ⊤) (lhs xy); next             x t  ↠  y t!
    )
).
resp_c  :=  resp_c ⊥ ⊥.
```

and pushing (x ↦ x ⊤) preserves convergence

```
Another  :=  (
    ∀x:term. Conv x → Conv (ap x ⊤)
) (
    yλa. λx. (
        trans_r t_r; done,
        a (ap x ⊤); next
    )
).
another  :=  Another ⊥.
```

Proofs for information ordering

Scott's information order is defined by the principle

$$\frac{\forall c{:}\mathsf{context}.\ \mathsf{Conv}(c\ x) \implies \mathsf{Conv}(c\ y)}{x \sqsubseteq y}\ (\mathcal{H}^*)$$

To encode proofs of information ordering, we thus need a notion of context.

```
context  :=  V (
    Yλc. Or (Prod c c). Maybe term
).
ap_c  :=  (context→context→context) (λc1, c2. inl (c1, c2)).
id_c  :=  context (inr none).
term_c  :=  (term→context) (2 inr).
```

```
compose_c  :=  (context → context → context) (
    λc. 𝐘λcomp. (λ(c1, c2). ap_c (comp c1) (comp c2),  c,  term_c)
).
```

```
at  :=  (context → term → term) (
    λx. 𝐘λe. (λ(c1, c2). ap (e c1) (e c2),  x,  𝐈)
).
```

The type of information ordering proof is thus

```
Ord  :=  (term → term → 𝐕) (
    λx, y. ∀c. (Conv. at c y) → (Conv. at c x)
).
```

```
Resp  :=  (
    ∀x, y. If (Red x y). Ord x y
) ( λ−, −, xy.
    λ−, cy_c.
    resp_c (rhs xy) cy_c                c x  ↠  c y!
).
resp  :=  Resp ⊥ ⊥.
```

Finally we can show that terms and information ordering proofs between them form a monadic category (i.e. whose skeleton is a poset).

```
Refl  :=  (
    ∀x. Ord x x
) ( λ − .
    λ−, x_c.
    x_c
).
refl  :=  Refl ⊥.
```

```
Trans  :=  (
    ∀x, y, z. If (Ord x y). If (Ord y z). Ord x z
) ( λ−, −, −, xy, yz.
    λc, z_c.
    xy c. yz c. z_c
).
trans  :=  Trans ⊥ ⊥ ⊥.
```

We leave to the reader the proofs of motonicity.

```
Rhs  :  (∀f, x, y. If (Ord x y). Ord (ap f x) (ap f y))
Lhs  :  (∀f, g, x. If (Ord f g). Ord (ap f x) (ap g x))
```

6.2 Type inference and termination in Gödel's T

Our main theorem in this section will be a proof of termination in Gödel's T (more precisely, weak termination at base type, which is still enough to prove consistency of peano arithmetic). Although we have not implemented this level of reasoning in SKJO , the example demonstrates the potential for expressing complicated logical statements in SKJO , even beyond first-order logic.

Our development of the language of Gödel's T also illustrates the power of join as a type inference mechanism. Here terms will be Church-style combinators annotated with types at each term and subterm. The type-as-closure term has a baked-in type inference algorithm, so that partially-annotated terms are automatically raised to the most-annotated possible terms, given the partial information. This is done by propagating type information among occurrences of each type, using the join operation.

Primitive recursion over natural numbers

We will prove the termination of well-typed terms using primitive recursion over natural numbers.

```
zero : num
succ : num → num
 rec : ∀a:V. num → (num → a → a) → a → a
```

Recall that most of this has been defined already.[1]

```
!check num = V (Yλa. Maybe a).
zero := num none.
succ := (num → num) some.
rec := (∀a:V. num → (a → num → a) → a → a) (
    Yλρ. λn, f, x.  n x λn′.  ρ n′ (f∘succ) (f zero x)
).
!check succ = Some num.
```

Now we can, for example, add

```
add := (num → num → num) (λm, n. rec nat_ty n (K succ) m).
!check add zero zero = zero.
!check (∀m::test_num, n::test_num. add(succ m)n = add m(succ n)).
```

Now to represent proofs of totality, we need simple types over num

```
pre_ty := V (Yλa. Maybe. Prod a a).
check_ty := Check ty (Yλτ. check_Maybe. check_Prod τ τ).
!define ty := Checked pre_ty check_ty.
!check check_ty : Check ty.
```

```
!define test_ty := Test ty (Yλτ. test_Maybe. test_Prod τ τ).
```

with introduction forms

```
num_ty := ty none.
exp_ty := (ty → ty → ty) (λσ, τ. some(σ, τ)).
!check num_ty :: test_ty.
!check (∀σ::test_ty, τ::test_ty. exp_ty σ τ::test_ty).
```

and checked elimination forms

```
case_num := (ty → unit) (I, ⊤).
case_exp := (ty → W Prod ty) (⊤, I).
!check case_num num_ty = I.
!check case_exp num_ty = error.
!check (∀σ::test_ty, τ::test_ty.
    case_exp (exp_ty σ τ) = (σ, τ)      AND
    case_num (exp_ty σ τ) = error
).
```

[1] The definitions in this section conflict with those in earlier sections. This conflict could be averted using a module system or some namespace management, which we have not implemented. Thus this section serves as an example of what *could* be described in the system SKJO , but what has never been parsed by Johann.

We can evaluate these type codes to types-as-closures with

$$\begin{array}{l}
\mathsf{ty\_eval} \;:=\; (\mathsf{ty} \to \mathbf{V})\ (\mathbf{Y}\lambda \mathsf{e}.\ (\mathsf{num},\ \lambda\sigma,\tau.\ \mathsf{e}\ \sigma \to \mathsf{e}\ \tau)).\\
!\mathsf{check}\ \ \mathsf{eval\_ty}\ \ \mathsf{num\_ty} = \mathsf{num}.\\
!\mathsf{check}\ (\\
\quad \forall\sigma :: \mathsf{test\_ty},\ \ \tau :: \mathsf{num\_ty}.\\
\quad \mathsf{eval\_ty}\ \ (\mathsf{exp\_ty}\ \ \sigma\ \ \tau) = \mathsf{eval\_ty}\ \ \sigma \to \mathsf{eval\_ty}\ \ \tau\\
).
\end{array}$$

We can also discriminate between total types, with truth values bool, semi, and unit.

$$\begin{array}{l}
\mathsf{eq\_ty} \;:=\; \mathbf{P}\ \mathbf{C}\ (\mathsf{ty} \to \mathsf{ty} \to \mathsf{bool})\ (\\
\quad \mathbf{Y}\lambda \mathsf{e}.\ ((\mathsf{true},\ \mathbf{K}\ \mathsf{false}),\ \lambda \mathsf{a},\mathsf{b}.(\mathsf{false},\ \lambda \mathsf{a}',\mathsf{b}'.\ \mathsf{and}\ (\mathsf{e}\ \mathsf{a}\ \mathsf{a}')\ (\mathsf{e}\ \mathsf{b}\ \mathsf{b}')))\\
).\\
!\mathsf{check}\ \ \mathbf{W}\ \mathsf{eq\_ty}\ \ \mathsf{num\_ty} = \mathbf{K}.\\
!\mathsf{check}\ \ \mathsf{eq\_ty}\ \ \tau\ \ \tau = \mathsf{test\_ty}\ \ \tau\ \ \mathbf{K}.\\
!\mathsf{check}\ \ (\forall\sigma :: \mathsf{test\_ty},\ \ \tau :: \mathsf{test\_ty}.\ \ \mathsf{eq\_ty}\ \ \mathsf{num\_ty}\ \ (\mathsf{exp\_ty}\ \ \sigma\ \ \tau) = \mathbf{F}).\\
!\mathsf{check}\ \ (\forall\sigma :: \mathsf{test\_ty},\ \ \tau :: \mathsf{test\_ty}.\ \ \mathsf{eq\_ty}\ \ (\mathsf{exp\_ty}\ \ \sigma\ \ \tau)\ \ (\mathsf{exp\_ty}\ \ \sigma\ \ \tau) = \mathbf{K}).\\
!\mathsf{check}\ (\\
\quad \forall\sigma :: \mathsf{test\_ty},\ \ \sigma' :: \mathsf{test\_ty},\ \ \tau :: \mathsf{test\_ty},\ \ \tau' :: \mathsf{test\_ty}.\\
\quad \mathsf{eq\_ty}\ \ (\mathsf{exp\_ty}\ \ \sigma\ \ \tau)\ \ (\mathsf{exp\_ty}\ \ \sigma'\ \ \tau') = \mathsf{and}\ \ (\mathsf{eq\_ty}\ \ \sigma\ \ \sigma')\ \ (\mathsf{eq\_ty}\ \ \tau\ \ \tau')\\
).
\end{array}$$

$$\begin{array}{l}
\mathsf{if\_eq\_ty} \;:=\; \mathbf{P}\ \mathbf{C}\ (\mathsf{ty} \to \mathsf{ty} \to \mathsf{semi})\ (\\
\quad \mathbf{Y}\lambda \mathsf{e}.\ ((\mathbf{I},\bot),\ \lambda \mathsf{a},\mathsf{b}.\ (\bot,\ \lambda \mathsf{a}',\mathsf{b}'.\ \mathsf{and\_semi}(\mathsf{e}\ \mathsf{a}\ \mathsf{a}')(\mathsf{e}\ \mathsf{b}\ \mathsf{b}')))\\
).\\
!\mathsf{check}\ \ \mathsf{if\_eq\_ty}\ \ \sigma\ \ \tau = \mathsf{if}\ \ (\mathsf{eq\_ty}\ \ \sigma\ \ \tau).\\
!\mathsf{check}\ \ \mathsf{if\_eq\_ty}\ \ \tau\ \ \tau = \mathsf{test\_ty}\ \ \tau.
\end{array}$$

$$\begin{array}{l}
\mathsf{assert\_eq\_ty} \;:=\; \mathbf{P}\ \mathbf{C}\ (\mathsf{ty} \to \mathsf{ty} \to \mathsf{unit})\ (\lambda\sigma,\tau.\ \mathsf{check\_ty}\ (\sigma \mid \tau)).\\
!\mathsf{check}\ \ \mathsf{assert\_eq\_ty}\ \ \sigma\ \ \tau = \mathsf{assert}\ \ (\mathsf{eq\_ty}\ \ \sigma\ \ \tau).\\
!\mathsf{check}\ \ \mathsf{assert\_eq\_ty}\ \ \tau\ \ \tau = \mathsf{check\_ty}\ \ \tau.
\end{array}$$

Below we will often have to assert the equality of three or four terms at once. Thus it is convenient to use the shorthand

$$\begin{array}{l}
!\mathsf{check}\ \ \mathsf{check\_ty}\ \ (\sigma \mid \tau) = \mathsf{assert\_eq\_ty}\ \ \sigma\ \ \tau.\\
!\mathsf{check}\ \ \mathsf{check\_ty}\ \ (\rho \mid \sigma \mid \tau) = \mathsf{assert\_eq\_ty}\ \ \rho\ \ \sigma\ \ \mid\ \ \mathsf{assert\_eq\_ty}\ \ \sigma\ \ \tau.
\end{array}$$

The equality assertion also gives us an alternate representation for case_num.

$$!\mathsf{check}\ \ \mathsf{case\_num} = \mathsf{assert\_eq\_ty}\ \ \mathsf{num\_ty}.$$

Finally we will need to join over types in proof search, so we define

$$\begin{array}{l}
\mathsf{Join\_ty} \;:=\; \mathsf{Sset}\ \ \mathsf{ty}\ (\mathbf{Y}\lambda \mathsf{j}.\ \langle \mathsf{num\_ty}\rangle\ \mid\ \mathsf{j}\lambda\sigma.\ \mathsf{j}\lambda\tau.\ \langle \mathsf{exp\_ty}\ \ \sigma\ \ \tau\rangle).\\
!\mathsf{check}\ \ \mathsf{Join\_ty}\ \ \mathsf{test\_ty} = \mathbf{I}.
\end{array}$$

Well-behaved terms are well-typed combinators

$$\frac{\mathsf{M}:\sigma\to\tau \qquad \mathsf{N}:\sigma}{\mathsf{M}\ \ \mathsf{N}:\tau}\,(\mathsf{ap}) \qquad \frac{}{\mathbf{S}:(\rho\to\sigma\to\tau)\to(\rho\to\sigma)\to\rho\to\tau} \qquad \frac{}{\mathbf{K}:\sigma\to\tau\to\sigma} \qquad \frac{}{\mathsf{zero}:\mathsf{num}}$$

$$\frac{}{\mathsf{succ}:\mathsf{num}\to\mathsf{num}} \qquad \frac{}{\mathsf{rec}:\mathsf{num}\to(\mathsf{num}\to\tau\to\tau)\to\tau\to\tau}$$

which are coded as Church-style (type,term) pairs that typecheck. In contrast to the cut-free case of reduction proofs in 3.14, typability allows cut via the (ap) rule; thus we need to annotate the types of subterms. We begin with terms with possibly-incorrect typing annotations

```
pre_term  :=  V (Yλa.
    Prod ty.                              type annotation
    Sum (Prod a a).                       application
    3 Maybe bool                          five atoms
).
test_pre_term  :=  Test pre_term (Yλτ.
    test_Prod test_ty.
    test_Sum (test_Prod τ τ).
    3 test_Maybe test_bool
).
check_pre_term  :=  Check pre_term (Yλc.
    check_Prod check_ty.
    check_Sum (check_Prod c c).
    3 check_Maybe check_Bool
).
```

We define the type of well-typechecked terms as a closure under the simple back-and-forth type-inference algorithm

```
!define term := $\mathbf{P}$ (Checked pre_term check_term) (
    ce := case_exp.   e := exp_ty.   n := num_ty.

    $\mathbf{Y}\lambda a, (\tau, u).\ \langle u \rangle$ (
    Case: ap(f : $\sigma \to \tau$)(x : $\sigma$)  :  $\tau$
        $\lambda(st, -) : a, (\sigma, -) : a.$   ce st $\lambda\sigma', \tau'$.
        $\underline{\sigma} := \sigma \mid \sigma'$.   $\underline{\tau} := \tau \mid \tau'$.
        ($\underline{\tau}$,  inl ((e $\underline{\sigma}$ $\underline{\tau}$,  $\bot$),  ($\underline{\sigma}$,  $\bot$)),
    Case: $\mathbf{S} : (\rho \to \sigma \to \tau) \to (\rho \to \sigma) \to \rho \to \tau$
        ce $\tau$ $\lambda$rst, rs_rt.   ce rst $\lambda\rho$, st.   ce st $\lambda\sigma, \tau$.
        ce rs_rt $\lambda$rs, rt.   ce rs $\lambda\rho', \sigma'$.   ce rt $\lambda\rho'', \tau'$.
        $\underline{\rho} := \rho \mid \rho' \mid \rho''$.   $\underline{\sigma} := \sigma \mid \sigma'$.   $\underline{\tau} := \tau \mid \tau'$.
        (e (e $\underline{\rho}$. e $\underline{\sigma}$ $\underline{\tau}$). e (e $\underline{\rho}$ $\underline{\sigma}$). e $\underline{\rho}$ $\underline{\tau}$,  $\bot$),
    Case: $\mathbf{K} : \rho \to \sigma \to \rho$
        ce $\tau$ $\lambda\rho$, sr.   ce sr $\lambda\sigma, \rho'$.
        $\underline{\rho} := \rho \mid \rho'$.   $\underline{sr}$ := sr | e $\sigma$ $\underline{\rho}$.
        (e $\underline{\rho}$ $\underline{sr}$,  $\bot$),
    Case: zero : num
        $\underline{\tau} := \tau \mid$ n.
        ($\underline{\tau}$,  $\bot$),
    Case: succ : num $\to$ num
        ce $\tau$ $\lambda$n$'$, n$''$.
        $\underline{\tau}$ := n | n$'$ | n$''$.
        ($\underline{\tau}$,  $\bot$),
    Case: rec : num $\to$ (num $\to \sigma \to \sigma$) $\to \sigma \to \sigma$
        ce $\tau$ $\lambda$n$'$, nss_ss.   ce nss_ss $\lambda$nss, ss.
        ce nss $\lambda$n$''$, ss$'$.   ce ss$'$ $\lambda\sigma, \sigma'$.   ce ss $\lambda\sigma'', \sigma'''$.
        $\underline{n}$ := n | n$'$ | n$''$.   $\underline{\sigma} := \sigma \mid \sigma' \mid \sigma'' \mid \sigma'''$.   $\underline{ss}$ := e $\underline{\sigma}$ $\underline{\sigma}$.
        (e $\underline{n}$. e (e $\underline{n}$ $\underline{ss}$) $\underline{ss}$,  $\bot$)
    )
).
```

In each case, we propagate type information by
(1) splitting type annotations using ce = case_exp,
(2) join together all types that should be the same, and
(3) building a better-typed term with the joined annotations.
The ap case also reconstruct better-typed subterms, so that typing information can pass across subterms. This form of type inference allows type sketching in proofs below.

Theorem 6.2.1. *(type inference) The closures* term *raises partially-annotated terms as follows*
Inferring $\mathsf{ap}(\mathsf{f}:\sigma\rightarrow\tau)(\mathsf{x}:\sigma)\ :\ \tau$

```
!check (∀σ : ty :: check_ty, τ : ty :: check_ty.
    e := exp_ty

    term (⊥,  ap ⊥)  =
         (⊥,  ap(e ⊥ ⊥, ⊥)(⊥, ⊥))  AND

    term (⊥,  ap(e σ ⊥, ⊥)(⊥, ⊥))  =
    term (⊥,  ap(e ⊥ ⊥, ⊥)(σ, ⊥))  =
         (⊥,  ap(e σ ⊥, ⊥)(σ, ⊥))  AND

    term (τ,  ap(e ⊥ ⊥, ⊥)(⊥, ⊥))  =
    term (⊥,  ap(e ⊥ τ, ⊥)(⊥, ⊥))  =
    term (⊥,  ap(e ⊥ ⊥, ⊥)(τ, ⊥))  =
         (τ,  ap(e ⊥ τ, ⊥)(⊥, ⊥))
).
```

Inferring $\mathbf{S}:(\rho\rightarrow\sigma\rightarrow\tau)\rightarrow(\rho\rightarrow\sigma)\rightarrow\rho\rightarrow\tau$

```
!check (∀ρ : ty :: check_ty,  σ : ty :: check_ty,  τ : ty :: check_ty.
    e := exp_ty.   ss := inr none.

    term (⊥, ss)  =
         (e (e ⊥. e ⊥ ⊥). e (e ⊥ ⊥). e ⊥ ⊥,  ss)  AND

    term (e (e ρ. e ⊥ ⊥). e (e ⊥ ⊥). e ⊥ ⊥,  ss)  =
    term (e (e ⊥. e ⊥ ⊥). e (e ρ ⊥). e ⊥ ⊥,  ss)  =
    term (e (e ⊥. e ⊥ ⊥). e (e ⊥ ⊥). e ρ ⊥,  ss)  =
         (e (e ρ. e ⊥ ⊥). e (e ρ ⊥). e ρ ⊥,  ss)  AND

    term (e (e ⊥. e σ ⊥). e (e ⊥ ⊥). e ⊥ ⊥,  ss)  =
    term (e (e ⊥. e ⊥ ⊥). e (e ⊥ σ). e ⊥ ⊥,  ss)  =
         (e (e ⊥. e σ ⊥). e (e ⊥ σ). e ⊥ ⊥,  ss)  AND

    term (e (e ⊥. e ⊥ τ). e (e ⊥ ⊥). e ⊥ ⊥,  ss)  =
    term (e (e ⊥. e ⊥ ⊥). e (e ⊥ ⊥). e ⊥ τ,  ss)  =
         (e (e ⊥. e ⊥ τ). e (e ⊥ ⊥). e ⊥ τ,  ss)  AND
).
```

Inferring $\mathbf{K}:\rho\rightarrow\sigma\rightarrow\rho$

```
!check (∀σ : ty :: check_ty,  τ : ty :: check_ty.
    e := exp_ty.   kk := inr none.

    term (⊥, kk)  =
         (e ⊥. e ⊥ ⊥,  kk)  AND

    term (e σ. e ⊥ ⊥,  kk)  =
    term (e ⊥. e ⊥ σ,  kk)  =
         (e σ. e ⊥ σ,  kk)  AND
).
```

Inferring $\mathsf{zero}:\mathsf{num}$ *and* $\mathsf{succ}:\mathsf{num}\to\mathsf{num}$

```
!check ( zz := 2 inr none.   term (⊥, zz) = (num_ty, zz)).
!check ( ss := 3 inr true.   term (⊥, ss) = (W exp_ty num_ty, ss)).
```

Inferring $\mathsf{rec}:\mathsf{num}\to(\mathsf{num}\to\sigma\to\sigma)\to\sigma\to\sigma$

```
!check (∀τ : ty :: check_ty.
    e := exp_ty.   n := num_ty.   rr := 3 inr false.

    term (⊥, rr) =
         (e n. e (e n. e ⊥ ⊥). e ⊥ ⊥, rr)  AND

    term (e n. e (e n. e τ ⊥). e ⊥ ⊥, rr) =
    term (e n. e (e n. e ⊥ τ). e ⊥ ⊥, rr) =
    term (e n. e (e n. e ⊥ ⊥). e τ ⊥, rr) =
    term (e n. e (e n. e ⊥ ⊥). e ⊥ τ, rr) =
         (e n. e (e n. e τ τ). e τ τ, rr)
).
```

Proof. Left as exercise for Johann. □

Now we can define types of typechecked terms of both general and specific types

```
check_term := Check term check_pre_term.
!define test_term := Test term test_pre_term.
!check check_term = test2check test_term.
```

```
!define Term := (ty → P term) (λτ. Above (τ, ⊥)).
!check term = Term ⊥.
!check Term τ <: term.
```

```
!define test_Term := (∀τ : ty. Test. Term ty) test_pre_term.
!check test_term = test_Term ⊥.
!check test_Term τ <:: test_term.
```

with intro forms parametric parametric in types.

```
ap := (term → term → term) (λf, x. (⊥, inl(f, x))).
S := (ty → ty → ty → term) (
    λρ, σ, τ. (e := exp_ty. e (e ρ. e σ τ). e (e ρ σ). e ρ τ, inr none)
).
K := (ty → ty → term) (λσ, τ. (exp_ty ρ. exp_ty σ ρ, 2 inr none)).
zero := term (nat_ty, 3 inr none).
succ := term (W exp_ty nat_ty, 4 inr true).
rec := (ty → term) (
    λτ. (e := exp_ty. e num_ty. e (e num_ty. e τ τ). e τ τ, 4 inr false)
).
```

```
!check (
    ∀σ :: test_ty,  τ :: test_ty.
    ∀f :: test_Term(exp_ty σ τ),  x :: test_Term σ.
    ap f x :: test_term
).
!check (∀ρ :: test_ty,  σ :: test_ty,  τ :: test_ty.  S ρ σ τ :: test_term).
!check (∀σ :: test_ty,  τ :: test_ty.  K σ τ :: test_term).
!check zero :: test_term.
!check succ :: test_term.
!check (∀τ :: test_ty.  rec τ :: test_term).
```

We also define compound intro forms for convenience

```
B  :=  λap(ap S(ap K S))K.
compose  :=  (λf, g.  ap(ap B f)g).
recurse  :=  (λn, f, x.  ap(ap(ap(rec ⊥)n)f)x).
```

where the type of rec in recurse is inferred from f's or x's type.

Lemma 6.2.2. $\mathrm{inhab}(\mathsf{term}) = \{\top\} \cup$*"well-typed partial terms".*

Proof. By typechecking, all type errors are raised to $\top$. □

We can also evaluate terms to their proper types

```
eval  :=  (term → any) (
    Yλe, (τ, x).  eval_ty τ.  x (λy, z.  (e y)(e z),  S, K, zero, succ, rec)
).
Eval  :=  (∀τ : ty.  Term τ  →  any) (λ − .  eval).
!check eval = Eval ⊥.
```

Lemma 6.2.3. *(reduction) Evaluation respects application and the reduction rules*

```
S x y z = x z(y z)
K x y = x
rec zero f x = x
rec (succ n) f x = rec n (f∘succ) (f zero x)
```

Formally checking,

```
!check (
    ∀σ :: test_ty,  τ :: test_ty.
    ∀f :: test_Term(exp_ty σ τ),  x :: test_Term σ.
    eval(ap f x) = (eval f)(eval x)
).
!check (
    ∀ρ :: test_ty,  σ :: test_ty,  τ :: test_ty.
    ∀x :: test_Term(exp_ty ρ. exp_ty σ τ).
    ∀y :: test_Term(exp_ty ρ σ),  z :: test_Term ρ.
    eval(ap(ap(ap(S ρ σ τ)x)y)z)  =
    eval(ap(ap x z)(ap y z))
).
!check (
    ∀σ :: test_ty,  τ :: test_ty.
    ∀x :: test_Term σ,  y :: test_Term τ.
    eval(K σ τ x y)  =
    eval x
).
!check ( n := num_ty. e := exp_ty.
    ∀τ :: test_ty.
    ∀f :: test_Term(e n. e τ τ),  x :: test_Term σ.
    eval(rec τ zero f x)  =
    eval x
).
!check ( n := num_ty. e := exp_ty.
    ∀τ :: test_ty.
    ∀n :: test_Term n,  f :: test_Term(e n. e τ τ),  x :: test_Term σ.
    eval(ap(ap( ap(rec τ)(ap succ n) )f)x)  =
    eval(ap(ap( ap(rec τ)n ) (compose f succ)) (ap (ap f zero) x))
).
```

Proving termination

The main theorem of this system is termination (weak normalization): any well-typed total term of type num_τ evaluates to a total numeral, expressed as the equivalence of testing code before and after evaluation.

Theorem 6.2.4. *(termination)*

```
!check (∀n :: test_Term num_ty. eval n :: test_num).
```

or simply

```
!check test_Term num_ty  <::  test_num∘(Eval num_ty).
```

Proof. By a hereditary termination argument/Tait's method/logical relations...

Definition 6.2.5. (hereditary termination) We say a well-typed term $\mathsf{m}:\tau$ is *reducible* in case

- $\tau = \mathsf{num}$ and m evaluates to a total numeral (i.e. $\mathsf{eval\ m} \in \mathsf{nums}$), or
- $\tau = \rho \to \sigma$ and for every reducible term $\mathsf{x}:\rho$, also $\mathsf{m\ x}:\sigma$ is reducible.

Lemma 6.2.6. *Every well-typed term is reducible.*

Proof. By induction on term structure:

Case: ap: $\mathsf{u}\ \mathsf{v}{:}\tau$, where u,v are reducible. By inversion, $\mathsf{u}{:}\sigma\to\tau$ and $\mathsf{v}{:}\sigma$ for some σ, whence u,v are reducible by hypothesis. Hence $\mathsf{u}\ \mathsf{v}$ is reducible.

Case: **S**: Let $\mathsf{x}{:}\rho\to\sigma\to\tau, \mathsf{y}{:}\rho\to\sigma, \mathsf{z}{:}\rho$ be reducible. Then $\mathsf{x}\ \mathsf{z}$ and $\mathsf{y}\ \mathsf{z}$ are reducible, so $\mathsf{x}\ \mathsf{z}(\mathsf{y}\ \mathsf{z}) = \mathbf{S}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z}$ is reducible. Hence $\mathbf{S}\ \mathsf{x}\ \mathsf{y}$, hence $\mathbf{S}\ \mathsf{x}$, hence **S** is reducible.

Case: **K**: Let $\mathsf{x}{:}\sigma, \mathsf{y}{:}\tau$ be reducible. Then $\mathbf{K}\ \mathsf{x}\ \mathsf{y} = \mathsf{x}$, hence $\mathbf{K}\ \mathsf{x}$, hence **K** is reducible.

Case: zero,succ: trivial.

Case: rec: Suppose $\mathsf{n}{:}\mathsf{num}, \mathsf{fs}{:}\mathsf{num}\to\tau\to\tau, \mathsf{x}{:}\tau$ are reducible. By hypothesis n reduces to a number n'. We show by strong induction on n' that $\mathsf{rec}\ \mathsf{n}\ \mathsf{fs}\ \mathsf{x}$ is reducible, whence $\mathsf{rec}\ \mathsf{n}\ \mathsf{fs}$, whence $\mathsf{rec}\ \mathsf{n}$, whence rec is reducible.

Subcase: $\mathsf{n}' = \mathsf{zero}$: $\mathsf{rec}\ \mathsf{zero}\ \mathsf{f}\ \mathsf{x} = \mathsf{x}$ which is reducible by hypothesis.

Subcase: $\mathsf{n}' = \mathsf{succ}\ \mathsf{n}''$: $\mathsf{rec}(\mathsf{succ}\ \mathsf{n}'')\mathsf{f}\ \mathsf{x} = \mathsf{rec}\ \mathsf{n}''(\mathsf{f}\circ\mathsf{succ})(\mathsf{f}\ \mathsf{zero}\ \mathsf{x})$. By outer hypothesis and assumption, $\mathsf{fs}\circ\mathsf{succ}$ and $\mathsf{fs}\ \mathsf{zero}\ \mathsf{x}$ are reducible; hence by inner hypothesis also $\mathsf{rec}\ \mathsf{n}''(\mathsf{f}\circ\mathsf{succ})(\mathsf{f}\ \mathsf{zero}\ \mathsf{x})$ is reducible.

□

Expressing termination in SKJO

The logical relations proof relies on induction over types, terms, and numerals, so we begin with the three induction lemmas. Letting ϕ be an arbitrary predicate, the induction schemata are

$$\frac{\phi\ \mathsf{zero} \qquad \phi\ \mathsf{n} \implies \phi(\mathsf{succ}\ \mathsf{n})}{\forall \mathsf{n}.\ \phi\ \mathsf{n}}\ (\mathsf{ind-num}) \qquad \frac{\phi\ \mathsf{num\_ty} \qquad \phi\ \sigma \implies \phi\ \tau \implies \phi(\sigma\to\tau)}{\forall \sigma.\ \phi\ \sigma}\ (\mathsf{ind-ty})$$

$$\frac{\begin{array}{c}\forall\rho,\sigma,\tau.\ \phi\ (\underline{\mathbf{S}}\ \rho\ \sigma\ \tau) \qquad \forall\sigma,\tau.\ \phi(\underline{\mathbf{K}}\ \sigma\ \tau)\\ \phi(\underline{\mathsf{zero}}) \qquad \phi(\underline{\mathsf{succ}}) \qquad \forall\tau.\ \phi(\underline{\mathsf{rec}}\ \tau) \qquad \forall\sigma,\tau,\mathsf{f}{:}\sigma\to\tau,\mathsf{x}{:}\sigma.\ \phi\ \mathsf{f} \implies \phi\ \mathsf{x} \implies \phi(\mathsf{ap}\ \mathsf{f}\ \mathsf{x})\end{array}}{\forall\tau,\mathsf{x}{:}\tau.\ \phi\ \mathsf{x}}\ (\mathsf{ind-term})$$

where the universally quantified results simply assert that ϕ is true for all total numerals, types, and terms, respectively. All three lemmas rely on the Simple type theorem. We need totality testers and universal quantifiers over terms and types

```
total_pred_num := (λ{φ:num → bool}. O{test_num}{test_bool∘φ}).
total_pred_ty := (λ{φ:ty → bool}. O{test_ty}{test_bool∘φ}).
total_pred_term := (λ{φ:term → bool}. O{test_term}{test_bool∘φ}).
```

```
forall_ty := Forall_ {ty} (O {test_ty}) join_ty.
forall_term := Forall_ {term} (O {test_term}) join_term.
forall_Term := (∀{τ:ty}. Quantifier{Term τ}) (
    λ{τ}. Forall_ {Term τ} (O {test_Term τ}) (join_Term τ)
).
```

Lemma 6.2.7. *(induction over numerals)*

```
!check (
    ∀{φ:num → bool}::total_pred_num.
    impiles (φ zero).
    implies (forall_num {λ{n}. φ(succ n)}).
    forall_num {λ{n}. φ n}
).
```

Lemma 6.2.8. *(induction over types)*

```
!check (
    ∀{φ:ty → bool}::total_pred_ty.
    implies (φ num_ty).
    implies (forall_ty{λ{σ}. forall_ty{λ{τ}. φ. exp_ty σ τ}}).
    forall_ty{λ{τ}. τ}
).
```

Lemma 6.2.9. *(induction over terms)*

```
!check (
    ∀{φ:term → bool}::total_pred_term.
    implies (
        forall_ty{λ{σ}.
        forall_ty{λ{τ}.
        forall_Term{exp_ty ρ σ}{λ{f}.
        forall_Term{σ}{λ{x}.
            implies (φ f).  implies (φ x).  φ (ap f x)
        } } } }
    ).
    implies (
        forall_ty{λ{ρ}. forall_ty{λ{σ}. forall_ty{λ{τ}. φ(S ρ σ τ) }}}
    ).
    implies (forall_ty{λ{σ}. forall_ty{λ{τ}. φ(K σ τ) }}).
    implies (φ(zero)).
    implies (φ(succ)).
    implies (forall_ty{λ{τ}. φ(rec τ) }).
    forall_term{λ{τ}. φ(τ) }
).
```

Next we need to formalize convergence and reducibility. At base type, the reducibility is just convergence (modulo β-η) However, since reducibility at exponential types has high complexity, we need to move to SKJO to define it.

Definition 6.2.10. The *reducibility* predicate is

```
!define Red := (∀τ:ty. Term τ → bool) (
    Y′{λ{Red}. (
        Case: num is just convergence under evaluation
        λn:Term num. let{n} := quote_term n. conv {eval n},

        Case: σ → τ is hereditary convergence
        λσ:ty.  let {σ} := quote_ty σ.
        λτ:ty.  let {τ} := quote_ty τ.
        λf:Term(Exp σ τ).
        forall_Term{σ}{λ{x}. implies (Red σ x) (Red τ. ap f x)}
    ) }
).
!check (∀τ::test_ty, x::test_Term τ. Red τ x::test_bool).
```

```
!define red := (term → bool) (λx. let (τ, −) := x. Red τ x).
!check (∀x::test_term. red x::test_bool).
```

```
!check test_bool∘(Red num_ty) = test_num∘eval.
```

Our goal now is to show true = forall_term{red}, or equivalently, if∘red = test_term.

Expressing the termination proof in SKJO

The induction proof that each typeable term is reducible relies on reducibility being preserved under β-η equality, which follows from an induction on types.

Definition 6.2.11. The *invariance* predicate of reducibility at various types is

```
eval_eq := (term → term → bool) (
    e := quote_term; A{eval}. λx, y. equal(e x)(e y)
).
```

```
Inv := (Code{ty} → bool) (
    λ{τ::test_ty}.
    forall_Term{τ}{λ{x}. forall_Term{τ}{λ{y}.
        implies (eval_eq x y).
        iff (red x) (red y)
    } }
).
!check test_bool∘Inv = test_Code {ty} test_ty.
```

Lemma 6.2.12. *(invariance of reducibility under β-η equality)*

```
!check true = forall_ty {Inv}.
!check if∘Inv = test_Code {ty} test_ty.
```

Proof. By induction on type structure.
Case: Numerals.

```
!check true =
    forall_Term{num_ty}{λx.
    forall_Term{num_ty}{λy.
        implies (eval x = eval y). iff (red x) (red y)
    } }.
!check true = Inv{num_ty}.
```

Case: Exponentials. Equality of functions

```
!check true =
    forall_ty {λ{σ}.
    forall_ty {λ{τ}.
        st := exp_ty σ τ.
        forall_Term{st}{λ{f}.
        forall_Term{st}{λ{f′}.
            and (
                implies (eval_eq f f′).
                forall_Term{σ}{λ{x}.
                    fx := ap f x.  fx′ := ap f′ x.
                    and (eval_eq fx fx′).
                    iff (red fx) (red fx′) }
            ).
            iff (red f) (red f′)
    } } } }.
!check true = forall_ty{λ{σ}. forall_ty{λ{τ}. Inv{exp_ty σ τ}}}.
```

Finally, we combine using the induction schema

```
!check true = forall_ty {Inv}.
!check if∘Inv = test_Code {ty} test_ty.
```

□

Finally we verify the induction proof of hereditary termination

Theorem 6.2.13. *(hereditary termination)*

```
!check true = forall_term {red}.
!check if∘red = test_term.
```

Proof. By induction on term structure.
Case: ap: u v : τ, where u : $\sigma \to \tau$ and v : σ are reducible. Hence u v is reducible.

```
!check true =
    forall_ty{λ{σ}.  forall_ty{λ{τ}.
    forall_Term{exp_ty σ τ}{λ{u}. implies (red u).
    forall_Term{σ}{λ{v}. implies (red v).
        red(ap u v) }}}}.
```

Case: **S**: Let x : a → b → c, y : a → b, z : a be reducible. Then x z and y z are reducible, and hence x z(y z) is reducible, hence **S** x y hence **S** x hence **S** is reducible.

```
!check true = (
    e := exp_ty.
    forall_ty{λ{ρ}. forall_ty{λ{σ}. forall_ty{λ{τ}.  S_ := S ρ σ τ.
      and (forall_Term{e ρ. e σ τ}{λ{x}. implies (red x).  Sx := S_ x.
        and (forall_Term{e ρ σ}{λ{y}. implies (red y).  Sxy := Sx y.
          and (forall_Term{ρ}{λ{z}. implies (red z).  Sxyz := Sxy z.
            and (
              xz := ap x z.        and (red xz).
              yz := ap y z.        and (red yz).
              xz_yz := ap xz yx.   and (red xz_yz).
              eval_eq xz_yz Sxyz
            ). red Sxyz
          }). red Sxy
        }). red Sx
      }). red S_
    }}}
).
```

Case: **K**: Let x : ρ, y : σ be reducible. Then **K** x y = x is reducible, hence **K** x, hence **K** is reducible.

```
!check true =
    forall_ty{λ{σ}. forall_ty{λ{τ}.  K_ := K σ τ.
      and (forall_term{λ{x}.  implies (red x).
        and (forall_term{λ{y}. implies (red y).
          and (eval_eq Kxy x).
          red Kxy
        }). red Kx
      }). red K_
    }}.
```

Hence **I** is reducible.

Case: zero: trivial.

```
!check true = red $\underline{\text{zero}}$.
```

Case: succ:

```
!check true  =
    forall_Term{num_ty}{$\lambda$\{n\}. implies (red n). red($\underline{\text{succ}}$ n)}.
```

Case: rec: We show by induction on n : num that rec n is reducible for each n.

```
!check true = (
    forall_ty{$\lambda$\{$\tau$\}.
      rec$_\tau$   :=  $\underline{\text{rec}}$ $\tau$.
      forall$_n$  :=  forall_Red num_ty.
      forall$_f$  :=  forall_Red(exp_ty num_ty(exp_ty $\tau$ $\tau$)).
      forall$_x$  :=  forall_Red $\tau$.

      Case: zero
      and (
        rz  :=  ap rec$_\tau$ $\underline{\text{zero}}$.
        and (forall$_f$ {$\lambda$\{f\}.  rfz := rz f.
          and (forall$_x$ {$\lambda$\{x\}.  rzfx := ap rzf x.
            and (eval_eq rzfx x) (red rzfx)
          }). red rzf
        }). red rz
      ).

      Case: successor
      and (forall$_n$ {$\lambda$\{n\}.  n' := ap $\underline{\text{succ}}$ n.  and (red n').
        rn  :=  ap rec$_\tau$ n.
        rN  :=  ap rec$_\tau$ n'.
        and (forall$_f$ {$\lambda$\{f\}.  f' := ap(ap $\mathbf{B}$ $\underline{\text{succ}}$)f.  and (red f').
            rnF  :=  ap rn f'.
            rNf  :=  ap rN f.
            and (forall$_x$ {$\lambda$\{x\}.  x' := ap(ap f $\underline{\text{zero}}$)x.  and (red x').
                rnFX  :=  ap rnF x'.
                rNfx = ap rNf x.
                and (eval_eq rnFX rNfx).
                implies (red rnFX) (red rNfx)
            }). implies (red rnF) (red rNf)
        }). implies (red rn) (red rN)
      }).

      Finally, conclude
      red rec$_\tau$
    }).
```

$\square$

Chapter 7

Implementation of the Johann system

This chapter details some aspects of the Johann system. Most of the syntactic algorithms have already been described in 2 or elsewhere. Now we will focus on Johann's database in 7.2 and the AI algorithms behind randomized proofs search in 7.3, automated conjecturing in 7.4, and numerical basis optimization in 7.5. These three AI algorithms all make use of statistical calculations on the Johann's database, essentially approximations to Kolmogorov complexity.

7.1 Background and related work

Forward-Chaining with Equations

Perhaps the first use of forward-chaining in equational theories was the algorithm of Todd and Coxeter [TC36] (more recently [CDHW73]) for enumerating the cosets of a finitely presented finite group. Given a finitely presented group, say

$$\langle \mathsf{r}, \mathsf{s} \;\mid\; \mathsf{rr} = 1,\; \mathsf{rsrs}^{-1} = 1,\; \mathsf{s}^4 = 1 \rangle$$

the Todd-Coxeter algorithm starts with a database of relations of the form $\mathsf{x.y} = \mathsf{z}$:

$$\{\; \mathsf{r.r} = 1,\; \mathsf{rsr.s}^{-1} = 1,\; \mathsf{s}^3\mathsf{.s} = 1,\; \mathsf{r.r}^{-1} = 1,\; \mathsf{r}^{-1}\mathsf{.r} = 1,\; \mathsf{s.s}^{-1} = 1,\; \mathsf{s}^{-1}\mathsf{.s} = 1 \;\}$$

This database is then EXTENDed by inference rules for each pair $(\mathsf{g}, \mathsf{g}')$ of generators or inverses (e.g. a pair $(\mathsf{r}^{-1}, \mathsf{s}^{-1})$):

$$\frac{\mathsf{x}\;\;.\;\;\mathsf{g} = \mathsf{xg}}{\mathsf{xg}\;\;.\;\;\mathsf{g}' = \mathsf{xgg}'}\;(\mathsf{Extend} - \mathsf{g}, \mathsf{g}')$$

It is also necessary to backward-subsume equations following from the rules

$$\frac{\mathsf{y} = \mathsf{y}'}{\mathsf{xy} = \mathsf{xy}'}\;(\mathsf{Reduce} - \nu) \qquad\qquad \frac{\mathsf{x} = \mathsf{x}'}{\mathsf{xy} = \mathsf{x}'\mathsf{y}}\;(\mathsf{Reduce} - \mu)$$

Here it is assumed that substitution and string pattern-matching is automatic. If the presented group is finite, and the database is extended "sufficiently uniformly" (e.g., by breadth-first search), then the database will saturate to have $2|P||G|$ equations, where $|G|$ is the order of the group and $|P|$ is the number of generators.

The Todd-Coxeter algorithm is more natural in the setting of semigroups, and has many generalizations to equational theories of other algebraic structures, e.g., nonassociative- and many-sorted algebras, and even categories ([CW91],[BLW03]). For our interests, we will generalize to nonassociative algebras with a join operation (thus single-sorted semilattices). Although the join operation $x \mid y$, and thus the ordering relation $x \sqsubseteq y$ and its negation $x \not\sqsubseteq y$, is definable ($x \sqsubseteq y \iff x \mid y = y$), it is much more practical to introduce extra logical inference rules relating the ordering to the equational fragment, as in the term ordering rules below.

A major limitation of the Todd-Coxeter algorithm is the restriction to finite groups (or other structures). An often-used alternative to Todd-Coxeter is Knuth-Bendix completion based on reduction strategies ([KB70],[BN99]). Our application is concerned with algebraic structures which are not only infinite but also uncomputable in a sense that thwarts also the Knuth-Bendix algorithm, thus we use a probabilistic version of the Todd-Coxeter algorithm as a workaround.

Kolmogorov-Solomonoff-Chaitin Complexity

In the early 1960's, Ray Solomonoff (in [Sol64]), Andrey Kolmogorov (in [Kol65]), and Gregory Chaitin (in [Cha66]) all independently defined a notion of complexity of binary strings and streams that was based on the size of computer program generating them. In their definition, one assigns a length to each computer program in some model of computation, and defines the *complexity* $\mathbf{K}(x)$ of a binary string x as the length of the smallest program generating it. The resulting field of Algorithmic Complexity is surveyed in V'Yugin's paper [Yug99] and detailed in Li and Vitayni's comprehensive reference [LV97].

The main theorems about algorithmic complexity come from the three inventors different perspectives.

Theorem 7.1.1. *(Kolmogorov)* $\mathbf{K}(x)$ *is independent of machine model, up to additive and multiplicative constants.*

Proof. By Church's Thesis, there is a universal virtual machine and a translator between any two machines. The additive factor accounts for the size of the virtual machine, and the multiplicative factor accounts for the translation blow-up. □

Theorem 7.1.2. *(Chaitin)* $\mathbf{K}(x)$ *is computable from below, but not from above.*

Proof. From below, simply enumerate programs, and add to result when a program converges. From above, we could solve the halting problem by computing Kolmogorov complexity of all binary strings to a given precision. □

Chaitin used this result to define a real number whose value was provably uncomputable.

Theorem 7.1.3. *(Solomonoff, [Sol78]) among all computable Bayesian priors for predicting future values of a binary stream,* $\mathsf{exp}(-\mathbf{K}(x))$ *is Pareto optimal w.r.t., say, a zero-one loss loss function.*

Proof. Follows from machine independence. □

Our interest is mostly in Solomonoff's result, as it motivates our proof search and complexity definition. Solomonoff's original interest was in optimally estimating and predicting values in a binary stream. He defined a pseudo-algorithm to do this prediction by running all programs at once, a process now known as Solomonoff Induction. This idea has been generalized to agent systems by Hutter [Hut05], who extends Solomonoff's theorem in many directions.

One of the problems in applying standard Kolmogorov complexity to our problems is the non-differentiability caused by minimizing over programs. The solution lies in Solomonoff's original application, where the probability of a string was not a minimum of anything, but rather a *sum* over all program probabilities, where program probabilities are roughly $\mathsf{exp}(-\mathsf{program\_length})$. This is exactly what motivates our definition of complexity in 7.3.

Simulated Annealing and The Metropolis-Hastings Algorithm

Consider the Monte Carlo problem of sampling a complicated data structure with complicated constraints from a probability distribution. A naive attempt might try to randomly generate such data structures until one satisfying the constraint is found, however for complicated constraints this is often infeasible due to the low probability of satisfaction.

One class of algorithms to solve this problem are Markov-Chain Monte Carlo (MCMC) algorithms ([JS96]). MCMC algorithms have been studied since 1950's, when they were used to compute solutions to integral equations for nuclear devices. The general MCMC approach is to start with an example such feasible data structure, and randomly perturb it over time, effectively randomly walking around a state graph of feasible structures. As one walks farther and farther, the distribution over states asymptotically approaches a *steady state* distribution.

The relationship between the "microscopic" perturbation likelihoods and the "macroscopic" steady state distribution is called *detailed balance* ([JS96]), and can be considered the "correctness" statement of such an algorithm. Subject to correctness, it is also important to define the microscopic perturbations and likelihoods so that the *mixing rate* is high; that is, so that the initial distribution quickly diffuses and converges to the steady state distribution.

The MCMC algorithm closest to that employed in the Johann system is the Metropolis-Hastings algorithm ([CG95]). This algorithm is characterized by a *pair* of probability distributions governing perturbation: one to randomly choose a candidate perturbation, and one to decide whether to reject the candidate or proceed. The advantage afforded by the second step is that constraints that are difficult to predict ahead of time may be easy to check given a candidate (think P vs. NP).

Johann uses an MCMC algorithm to randomly generate a database of theorems in combinatory algebra, where the database is constrained to be saturated, and the database should be relevant to a specified goal, on average. (This will be explained in more detail below.) The point of divergence from the pure Metropolis-Hastings algorithm is that the database is expanded by a randomly chosen term, but the rejection step is allowed to remove any existing term, not just the latest-added.

The MCMC method can be applied to a maximization problem by treating the objective function as a likelihood distribution and randomly sampling from the distribution. The resulting sample will thus likely have a large value of the objective function. More generally, given an objective function $\mathsf{f}:\mathsf{X}\to[0,1]$ on the space of structures X, we can define a *temperature*-parameterized family $\mathsf{f}^{\beta}(-)$ of likelihood distributions (where $\beta = 1/\tau$ is the inverse temperature). The higher the temperature, the more f is smoothed-out, and the easier it is to randomly walk around the space. The lower the temperature, the sharper f is, and the more likely a random sample x will have a high value of $\mathsf{f}(\mathsf{x})$. To achieve a large value of f, one strategy is to walk around from the starting position at high temperature (to mix), and slowly decrease the temperature until a large value of f is found (to maximize). This strategy is called *simulated annealing*, in analogy to the process of slowly cooling metals to achieve a solid which maximizes grain size.

7.2 Saturating a database

In this section we overview Johann's database and the deterministic forward-chaining algorithms that exhaustively search for equational proofs within the database. The remaining sections in this chapter will discuss the artificial intelligence algorithms behind shaping this database: how randomly add and remove terms from the database, how to conjecture or find missing equations, and how to fine-tuning the probabilistic basis.

Formally, Johann's *database* consists of the following data:

- a finite set of *obs* (equivalence classes of terms), called the *support*;
- a small set of labelled obs, for *atoms* (e.g., $\mathbf{S}, \mathbf{K}, \mathbf{J}$);[1]
- a set of application equations $\mathsf{app} = \mathsf{lhs}\ \mathsf{rhs}$, for $\mathsf{app}, \mathsf{lhs}, \mathsf{rhs}$ in the support;
- a set of ordered pairs $\mathsf{x} \sqsubseteq \mathsf{y}$ known to be true, for x, y in the support;
- and a similar set for the negated relation $\mathsf{x} \not\sqsubseteq \mathsf{y}$.

We would like databases to satisfy a set of inference rules, e.g., simple constraints like $\dfrac{\mathsf{x} = \mathsf{y}}{\mathsf{f}\ \mathsf{x} = \mathsf{f}\ \mathsf{y}}$ for maintaining well-definedness of the application function, and more complicated rules like $\dfrac{\mathsf{x} \sqsubseteq \mathsf{z} \qquad \mathsf{y} \sqsubseteq \mathsf{z}}{\mathbf{J}\ \mathsf{x}\ \mathsf{y} \sqsubseteq \mathsf{z}}$ for

[1]This set of constants is independent of the probabilistic basis; these constants let Johann know which inference rules to enforce, whereas the terms in the basis determine what obs to add to the database.

the join operation **J** of 3.1. A variety of such inference rules will be developed throughout this thesis, as we axiomatize various extensions of untyped λ-calculus.

Each rule concludes with an equation or an order relation between an application of obs. Because there are only finitely many obs, there can be only finitely many rule-firings until all possible inference rules have been enforced. When all applicable inference rules have been fired in a database, we say the database is *saturated*. Everywhere outside of this section, we will assume that the database is saturated. In the remainder of this section we briefly describe how Johann saturates the database w.r.t. a given set of rules.

Each time a new ob (equivalence class) is added to the support, rules for equality and order relations must all be enforced until saturation. Since combining equivalence classes may decrease the number of order rules need to be enforced (a smaller support means fewer pairs to check), it saves time to enforce first those rules most likely to induce new equations, only checking positive order rules once the database is saturated w.r.t. equations, and only checking negative rules once the database is saturated w.r.t. positive order. The inference rules roughly stratify into

(0) merger of equivalence classes, subject to database constraints like functionality application,
(1) equational rules (β conversion, etc.),
(2) positive order $\mathsf{x} \sqsubseteq \mathsf{y}$, and
(3) negative order $\mathsf{x} \not\sqsubseteq \mathsf{y}$,

where lower conclusions seldom fire higher rules. Some exceptions are the antisymmetry rule when (2) induces a merger (0), and the div typing rules in 3.5 where apartness (3) can lead to mergers (0) or new equations (1).

Saturation of (0) is a generalization of the Todd-Coxeter algorithm of groups, where we relax associativity and do not require inverses. This involves merging of equivalence classes, which in turn requires merging of the order relations. The main rule being enforced is functionality of application, so that each lhs, rhs pair defines at most one application.

Saturation of (1) generalizes enforcement of group relations in the Todd-Coxeter algorithm, where in our setting, equations hold between nonassociative terms. The relevant inference rules are all hard-coded and hand-optimized in C++.

Saturation of (2) and (3) is similar, and simply involves keeping track of newly added hypotheses and checking for applicable rules. Together the positive and negative tables define a partially-known table with truth values true, false, unknown. Whenever it is proved that both $\mathsf{x} \sqsubseteq \mathsf{y}$ and $\mathsf{x} \not\sqsubseteq \mathsf{y}$ (i.e. inconsistency), then Johann aborts and we begin debugging to look for a bad assumption.

Empirically, saturating equations (1) is the most expensive, negative order (3) is the second most expensive, and positive order (2) is very cheap (since not much is known). Let O, E, P, N be the numbers of Obs (equivalence classes) in the support, application Equation triples, Positive order relations, and Negative order relations, respectively. We have observed that after annealing a large database (as described in 7.3), the following density relations hold independently of database size:

$$\begin{aligned} E/O^2 &\approx 0.05 - 0.3 \\ P/O^2 &\approx 0.05 \\ N/O^2 &\approx 0.85 - 0.95 \end{aligned}$$

where the variation in density is generally across theories or fragments.[2] Thus all three relations require space quadratic in the size of the support. The worst-case time-complexity for saturation is cubic in the support size, and takes about one second per new equivalence class at $O \approx 4800$ equivalence classes on a 2.8GHz Intel Core Duo. Mcallester in [McA99] discusses complexity analysis techniques for forward-chaining algorithms in logic programming.

In annealing a database, obs are randomly added to and removed from the database support w.r.t. sampling distributions that take quadratic time to compute. What makes randomization possible here, despite the high cost of calculating sampling distributions, is the unusually large amount of work done between random choices, in saturating the database, i.e., cubic versus quadratic.

[2]For example in SKJ, $E/O^2 \approx 0.05 - 0.1$, whereas in SK, $E/O^2 \approx 0.2 - 0.3$.

Implementation details

Our cache-conscious implementation uses lookup tables for the main data structures, with some auxiliary data structures to optimize for reading. The equation table is stored as an $O \times O$ array, partitioned into 8x8 blocks of 32-bit words, to reduce cache misses when traversing through both rows and columns. We also speed up traversal through this $\sim 5\%$ sparse table by keeping tables of bits recording whether there is an entry for a given $\mathsf{lhs}, \mathsf{rhs}$ pair. We keep two such "sketches", one oriented in each of row- and column-major, so that an entire cache line can be fetched at once (for, say, 512 table entries).

The negative and positive tables are similarly stored as pairs of bit arrays, one oriented in each of row- and column- major. Because there are far more table reads (hypothesis checks) than writes (rule-firings), it is cheaper to maintain consistency between two read-optimized tables, than keep a single write-optimized table. Storing order as bit-arrays also permits vectorized hypothesis checking, where an entire machine word (say 32 hypotheses) can be checked at once. We found that in our unvectorized version, negative order was the most expensive rule to enforce, and that after vectorization, it took only a small portion of the total time.

Finally, we need to be able to traverse through, for a given $(\mathsf{app}, \mathsf{lhs})$ pair, all rhs values that satisfy $\mathsf{lhs}\ \mathsf{rhs} = \mathsf{app}$; and similarly for given $(\mathsf{app}, \mathsf{rhs})$ pairs. We implement these two data structures as sets of splay trees ([ST85]), each with one tree node per proved equation $\mathsf{a} = \mathsf{l}\ \mathsf{r}$. We actually define a single node per equation, that participates in both trees but keeps only one copy of the $\mathsf{lhs}, \mathsf{rhs}, \mathsf{app}$ keys. By storing these nodes in an array, we can also easily traverse through all $\mathsf{lhs}, \mathsf{rhs}, \mathsf{app}$ triples. In practice, splay trees performed much better than an alternative ad hoc tree ordering we tried, possibly due to their probabilistic caching properties.

7.3 Annealing a database of interest

This section describes Johann's algorithm for randomly annealing a database, when motivated to simplify a corpus of terms. The main theorem is a *detailed balance* theorem, relating the microscopic operations of adding and removing obs to the macroscopic fitness of the database. We have also looked at more precise motives to accumulate evidence for automated conjecturing, but these turned out to too computationally expensive for pracitical purposes. In defining the motive we do implement, we make some linearization approximations to achieve an easily computable sum-product representation.

At the core of Johann is a database of facts about combinators. The database represents knowledge in three main data structures: a support (a time-varying set of say 1K–12K obs or equivalence classes), a application table (a multiplication table for the function-value application operation), and true/false/unknown-valued order table for the partial order relation $\sqsubseteq$. The application table and partial order table both require space quadratic in the number of obs.

Johann acquires knowledge by randomly adding and removing obs from the support, saturating the database at each step. To specify what knowledge to accumulate we specify a *motive*, in the form of a pair of probability distributions $\mathsf{e}(\mathsf{l}, \mathsf{r}), \mathsf{c}(\mathsf{a})$, from which random lhs-rhs pairs are drawn to extend the support, and from which random obs are drawn for removal, to contract the support. This pair of distributions is chosen so that the steady-state distribution over databases favors interesting databases (depending on our notion of interesting).

Expanding and Contracting the Database

A database's *support* consists of a few (~ 10) basic atoms (e.g., $\mathbf{S}, \mathbf{K}, \mathbf{J}$), a small number ($\sim 200 - 2000$) of terms used in axiomatizing our theory, and a variable set of other terms ($\sim 1000 - 10000$). Following Lakatos' terminology of the *hard core* of a scientific theory [Lak76], we call the fixed axiomatization portion of the database the *core*; all possible databases in a given theory extend the minimal core.

We consider two kinds of mutations to the database: *expansion*, by adding to the support random applications of existing terms, and *contraction*, by removing from the support randomly selected terms. Expansions and contractions are balanced to keep the database support within some size window. Because of the equational theorem proving in forward-chaining, "expansions" may sometimes *decrease* the size of the support, by creating proofs that existing pairs of terms are equal, and should be merged.

Suppose we have a notion R(db) of relevance of databases, and a corresponding notion of relevance of an ob w.r.t. a database:

```
               R(db+x)
    R(x;db) = -------
                R(db)
```

To specify mutation likelihoods w.r.t. relevance, we need a pair of distributions from which to sample, say e(l, r) to expand and $1/c(x)$ to contract (we use the inverse so that c and e both increase with relevance):

```
    e(l,r) = "simple estimate of R(l r;db)"
    c(x)   = "remainder of R(x;db) not accounted for in Sum lr=x. e(l,r)"
```

The statement of correctness for random backward-chaining is now a detailed balance statement for the expansion and contraction mutations:

Definition 7.3.1. (Detailed Balance) An expansion-contraction pair e(l, r), c(x) shows *detailed balance* iff state-transitions are proportional to relevance, i.e., for every db (extending the core) and term $x \in db$ (outside the core),

```
     R(db)
    -------   =  c(x) Sum lr=x in db. e(l,r)
    R(db-x)
```

Note that the restriction to databases extending the core is not a problem, as we can simply restrict and renormalize the steady-state distribution π. Our proofs for various e, c pairs will have the form
Proof Template: (Correctness) Let $x \in db$ be a term outside the core, and $db' := db - x$. Then

```
    c(x) Sum l r=x in db. e(l,r)  =  ...algebra...  =  R(x)
```

In constructing specific e, c pairs, we need to rewrite the macroscopic relevance ratio in two different microscopic forms:
- for expansion, where $l, r \in db$ but lr is not; and
- for contraction, where $x \in db$.

We prove correctness for a generic class of e, c-pairs as follows.

Theorem 7.3.2. *(Generic Detailed Balance) Suppose an expansion-contraction pair* e(l, r), c(x) *satisfies*
(1) e(l, r) *is positive for every* $l, r \in db$;
(2) e(l, r) *is independent of whether* lr *is in the database; and*
(3) c(x) *is defined by*

```
                          R(x;db)
    c(x) = -------------------------------------------
             Sum l,r in db. if lr=x and l!=x!=r then e(l,r)
```

Then e(l, r), c(l, r) *shows detailed balance.*

Proof. It is sufficient to show that

```
              Sum l,r. if lr=x                 e(l,r)  #evaluated in db
    const = ----------------------------------------
              Sum l,r. if lr=x and l!=x!=r then e(l,r)  #evaluated in db+x
```

for some constant independent of the database. Here two sums and two values e(l, r) are different, since they are evaluated in different databases, the numerator in the original db, and the denominator in the extended $db + x$.

Since this is a statement about steady-state databases, we may assume that no new theorems were found in expanding by x (besides those explicitly mentioning x. Under this assumption, the two sums indeed have the same domain.

Finally condition (2) guarantees that the two evaluations of e(l, r) are within a constant factor. □

Subject to correctness (i.e., that together e, c account for all of R) the more accurate $\mathsf{e}(\mathsf{l}, \mathsf{r})$ is, the higher the mixing rate will be, and the faster proofs will be found; that is: it's better to know how to look for a proof with $\mathsf{e}(\mathsf{l}, \mathsf{r})$, than to only know one when you see it with $\mathsf{c}(\mathsf{x})$. But accuracy must not come at the expense of computability: $\mathsf{e}(\mathsf{l}, \mathsf{r})$ must be easy to sample from; e.g., it would be nice if e factored into

```
e(l,r) = e_L(l) e_R(r) (1-e_rej(l,r))
```

where the rejection probability $\mathsf{e}_{\mathsf{rej}}$ is usually not too close to one. We now examine a few versions of expansion-contraction pairs, exercising the freedom of the $\mathsf{e} - \mathsf{c}$ tradeoff.

Subject to correctness (i.e., that together e, c account for all of R) the more accurate $\mathsf{e}(\mathsf{l}, \mathsf{r})$ is, the higher the mixing rate will be, and the faster proofs will be found; that is: it's better to know how to look for a proof with $\mathsf{e}(\mathsf{l}, \mathsf{r})$, than to know one when you see it with $\mathsf{c}(\mathsf{x})$. But accuracy must not come at the expense of computability: $\mathsf{e}(\mathsf{l}, \mathsf{r})$ must be easy to sample from; e.g., it would be nice if e factored into

```
e(l,r) = e_L(l) e_R(r) (1-e_rej(l,r))
```

where the rejection probability $\mathsf{e}_{\mathsf{rej}}$ is usually not too close to one. We now examine a few versions of expansion-contraction pairs, exercising the freedom of the $\mathsf{e} - \mathsf{c}$ tradeoff.

Annealing to simplify a corpus of terms

The motive we will examine for annealing a database is the motive to simplify a term, or more generally to simplify a corpus of terms. Let C denote a corpus, say a weighted set, or a probability mass function (pmf) over some set of terms. First we need a notion of complexity of obs in the database.

Definition 7.3.3. The syntactic *probability* $P(\mathsf{M})$ of an intensional term M database is defined by the probabilistic inference rules

$$\frac{\mathsf{x} \in \mathsf{basis}}{\mathsf{x\ term}}\ (\mathsf{P}_{\mathsf{basis}}(\mathsf{x})) \qquad\qquad \frac{\mathsf{x\ term} \qquad \mathsf{y\ term}}{\mathsf{x\ y\ term}}\ (\mathsf{P}_{\mathsf{app}})$$

so that, e.g., $\mathsf{P}(\mathbf{S}\ \mathbf{K}\ \mathbf{K}) = \mathsf{P}_{\mathsf{app}}^2\ \mathsf{P}_{\mathsf{basis}}(\mathbf{S})\ \mathsf{P}_{\mathsf{basis}}(\mathbf{K})^2$. The semantic *Solomonoff probability* $P(\mathsf{x})$ of an ob in the database is the sum over syntactic terms M in its equivalence class

$$P(\mathsf{x}) = \sum_{\mathsf{M} = \mathsf{x}} P(\mathsf{M})$$

The sum-product form of Solomonoff probability allows us to efficiently compute it as the fixed-point of a contraction equation

$$P(\mathsf{x}) = \mathsf{P}_{\mathsf{basis}}(\mathsf{x}) + \mathsf{P}_{\mathsf{app}} \sum_{\mathsf{l\ r} = \mathsf{x}} P(\mathsf{l}) P(\mathsf{r})$$

Now given a corpus C, we define the steady-state distribution $\pi(\mathsf{db}; C)$ of the database to be proportional to relevance:

Definition 7.3.4. The *relevance* of a database for simplifying a corpus is a real number

$$R(\mathsf{db}; C) = \prod_{c \in C} P(c; \mathsf{db})^{C(c)}$$

where $P(c; \mathsf{db})$ is the Solomonoff probability of the ob c as evaluated in a database db (c is assumed to be in the database), and $C(c)$ is the weight given by the corpus C to the ob c.

At steady-state we want the database to be relevant, so consider the transition probabilities of adding / removing a term x. We factor these probability ratio from a macroscopic function of the database to a mesoscopic product of sums of combinatory expressions M and expressions M[x] with specifiec occurrences of x, and finally into microscopic effects of sigle terms R(x).

$$\begin{aligned}
\frac{R(\mathsf{db}+\mathsf{x};C)}{R(\mathsf{db};C)} &= \prod_{c\in C}\frac{P(c;\mathsf{db}+\mathsf{x})^{C(c)}}{P(c;\mathsf{db})^{C(c)}} \\
&= \prod_{c\in C}\left(\frac{\sum_{\mathsf{M}=c\in\mathsf{db}+\mathsf{x}}P(\mathsf{M})}{\sum_{\mathsf{M}=c\in\mathsf{db}}P(\mathsf{M})}\right)^{C(c)} && \textit{summing over expressions evaluating to } c \\
&= \prod_{c\in C}\left(1+\sum_{\substack{\mathsf{M}=c\in\mathsf{db}+\mathsf{x}\\ \mathsf{x}\in\mathsf{M}}}P(\mathsf{M})\right)^{C(c)} \\
&\approx 1+\sum_{c\in C}C(c)\sum_{\substack{\mathsf{M}=c\in\mathsf{db}+\mathsf{x}\\ \mathsf{x}\in\mathsf{M}}}P(\mathsf{M}) && \textit{homogeneous corpus approximation} \\
&= 1+\sum_{\substack{\mathsf{M}\in\mathsf{db}+\mathsf{x}\\ \mathsf{x}\in\mathsf{M}}}P(\mathsf{M})C(\mathsf{M}) \\
&\approx 1+\sum_{\mathsf{M}\in\mathsf{db}+\mathsf{x}}\mathsf{occ}(\mathsf{x},\mathsf{M})P(\mathsf{M})C(\mathsf{M}) && \textit{extensional/intensional approximation} \\
&= 1+\sum_{\mathsf{M}[\mathsf{x}]\in\mathsf{db}+\mathsf{x}}P(\mathsf{M}[\mathsf{x}])C(\mathsf{M}) \\
&= 1+P(\mathsf{x})\sum_{\mathsf{M}[\mathsf{x}]\in\mathsf{db}+\mathsf{x}}P(\mathsf{M}[\ ])C(\mathsf{M}) && \textit{factoring } P(\mathsf{M}[\mathsf{x}])=P(\mathsf{x})P(\mathsf{M}[\ ]) \\
&=: 1+P(\mathsf{x})R(\mathsf{x}) && \textit{in terms of relevance}
\end{aligned}$$

We make two linearization assumptions here. The first assumes most terms in the corpus are of the same size, and tends to devalue the effect of large terms. The second approximation allows us to multiple-count terms M containing x by counting the occurrences of x in M. This latter is discussed by Pearl in [Pea88] as an approximation of *intensional* semantics of sum-product expressions as *extensional* semantics. This approximation allows us to compute R(x) using a sum-product algorithm for evidence propagation (also discussed by Pearl).

The reverse sum-product computation of R(x) is dual to the forward sum-product propagation of P(x) in computing term probability. We thus calculate relevance as the fixed-point of a propagation equation

$$\begin{aligned}
R(\mathsf{x}) &= C(\mathsf{x})+\sum_{\mathsf{w}}C(\mathsf{w\ x})\frac{P(\mathsf{w})\mathsf{P}_{\mathsf{app}}P(\mathsf{x})}{P(\mathsf{w\ x})}+\sum_{\mathsf{y}}C(\mathsf{x\ y})\frac{P(\mathsf{x})\mathsf{P}_{\mathsf{app}}P(\mathsf{y})}{P(\mathsf{x\ y})} \\
&= C(\mathsf{x})+P(\mathsf{x})\mathsf{P}_{\mathsf{app}}\sum_{\mathsf{z}}P(\mathsf{z})\left[\frac{C(\mathsf{z\ x})}{P(\mathsf{z\ x})}+\frac{C(\mathsf{x\ z})}{P(\mathsf{x\ z})}\right]
\end{aligned}$$

Now we can factor the transition probability as

$$1+P(\mathsf{x})R(\mathsf{x}) = P(\mathsf{x})\left(R(\mathsf{x})+\frac{1}{P(\mathsf{x})}\right)$$

and define an expansion,contraction pair of distributions

Theorem 7.3.5. *(detailed balance) The expansion,contraction pair*

$$e(\mathsf{l},\mathsf{r}) = P(\mathsf{l})P(\mathsf{r}) \qquad\qquad c(x) = R(\mathsf{x}) + \frac{1}{P(\mathsf{x})}$$

shows detailed balance for corpus simplification (up to the above approximations).

In practice, Johann ignores the $1/P(\mathsf{x})$ term in contraction, and contracts w.r.t. relevance alone. This is important, since some obs in the database may have been added for reasons other than random selection. For example, very complex obs are added during the process of checking theorems; these may have near zero probability, and hence would almost never be removed with the $1/P(\mathsf{x})$ term included. Another perspective is that we are factoring database relevance as

$$1 + P(\mathsf{x})R(\mathsf{x}) = \left(\frac{1}{R(\mathsf{x})} + P(\mathsf{x})\right) R(\mathsf{x})$$

so that Johann randomly adds terms w.r.t. $P(\mathsf{x})$ and randomly removes terms w.r.t. $R(\mathsf{x})$, and assume that the user randomly adds terms w.r.t. the distribution $\frac{1}{R(\mathsf{x})}$.

7.4 Automated Conjecturing

In this section we develop theory whereby Johann can query a user for *important* but *difficult* problems that seem to be *plausible*. We call this procedure *automated conjecturing* because Johann will guess what's true based on evidence, before a proof or disproof is known. Our theory starts by restricting the type of conjectures/assumptions that may be made to positive order relations $\mathsf{x} \sqsubseteq \mathsf{y}$. We then suggest a method to numerically quantify the evidence supporting conjectures, so that Johann can pose ranked conjectures to the user. Finally we demonstrate some theoretical limitations on what exactly can be conjectured in this way.

Sensible Assumptions

We attempt with Johann's theory to approach the Π^0_2-complete theory $\mathcal{H}^*$, but this cannot be done with finite or even r.e. axioms or axiom schemata. Thus as we prove properties of more complicated programs, we must continually make new assumptions. What assumptions should we make? Can we statistically quantify this?

Let us start with the most basic statements $\mathsf{x} \sqsubseteq \mathsf{y}$. At any time, some of these will be unproven, and some of those will be unprovable even in principle. It is consistent to assume such an unprovable statement either way, but we will restrict to positive assumptions $\mathsf{x} \sqsubseteq \mathsf{y}$, and let negative statements $\mathsf{u} \not\sqsubseteq \mathsf{v}$ follow from the single axiom $\top \not\sqsubseteq \bot$ from 2.1. Among positive assumptions, we restrict even further:

Definition 7.4.1. a statement $\mathsf{x} \sqsubseteq \mathsf{y}$ is *sensible* iff y converges whenever x does,

$$\forall \mathsf{f}.\ \mathsf{f}\ \mathsf{x} \not\sqsubseteq \bot \implies \mathsf{f}\ \mathsf{y} \not\sqsubseteq \bot$$

and write in that case $\mathsf{x} \sqsubseteq \mathsf{y}$ sensible.

Remark. Sensibility is derived from Hyland's and Wadsworth's $\mathcal{H}^*$ axiom

$$\frac{\forall \mathsf{C}[\ ].\ \mathsf{C}[\mathsf{x}]\ \mathsf{conv} \implies \mathsf{C}[\mathsf{y}]\ \mathsf{conv}}{\mathsf{x} \sqsubseteq \mathsf{y}}\ (\mathcal{H}^*)$$

where $\mathsf{C}[\ ]$ ranges over contexts, and conv is an r.e. predicate (see [Bar84]). In our simpler combinatory setting, variable binding may be ignored, so $\mathsf{C}[\ ]$ may be restricted to combinators (even traces). Also, for symmetry, we use both left and right monotony axioms.

Here sensibility is Π^0_2; the outer-most quantifier is universal. Thus we can accumulate evidence in favor of $\mathsf{x} \sqsubseteq \mathsf{y}$ by enumerating contexts f in which it is known that $\mathsf{fx}\ \mathsf{conv} \implies \mathsf{fy}\ \mathsf{conv}$. Indeed this suggests a way to approach $\mathcal{H}^*$ from η:

Theorem 7.4.2. *by making only simple, sensible assumptions, we arrive after ω steps at $\mathcal{H}^*$.*

Proof. Consider a chain $\mathcal{T}(-)$ of such theories, with limit $\mathcal{T}(\omega)$

$$\eta = \mathcal{T}(0) \ \subseteq\ \mathcal{T}(1) \ \subseteq\ \dots\ \subseteq\ \mathcal{T}(\mathsf{i}) \ \subseteq\ \dots\ \subseteq\ \mathcal{T}(\omega) \ \overset{?}{=}\ \mathcal{H}^*$$

Since $\mathcal{T}(0)$ is sensible, and each step $\mathcal{T}(\mathsf{i}+1) - \mathcal{T}(\mathsf{i})$ is sensible, each $\mathcal{T}(\mathsf{i})$ is sensible, and hence their limit $\mathcal{T}(\omega)$ is sensible. Since $\mathcal{H}^*$ is the unique $\sqsubseteq$-complete sensible theory, it suffices to show that $\mathcal{T}(\mathsf{i})$ is $\sqsubseteq$-complete; thus consider any statement $\mathsf{x} \sqsubseteq \mathsf{y}$. If $\eta \vdash \mathsf{x} \sqsubseteq \mathsf{y}$ then we are done. If $\mathsf{x} \not\sqsubseteq \mathsf{y}$ then $\mathsf{x} \sqsubseteq \mathsf{y}$ is non-sense, and will not be assumed. Otherwise, let $\mathsf{n} >\mid \mathsf{x} \mid + \mid \mathsf{y} \mid$ be a bound on the syntactic complexity of xy. Then after assuming the simplest $\mathsf{exp(n)}$sensible statements, we must have assumed $\mathsf{x} \sqsubseteq \mathsf{y}$. ☐

The proof requires all assumptions to be sensible, but sensibility is undecidable in general. Thus it would be more satisfying to prove a statement of probable approximate correctness (PAC, [Val84]), where correct means complete and consistent:

Conjecture 7.4.3. *For any $\delta, \epsilon \in (0,1)$, we can enumerate sufficiently much evidence to randomly generate a simple apparently sensible theory $\mathcal{T}$ such that*
(i) averaging over $\mathcal{T}$: $\mathsf{P}(\mathcal{T}\ \ \mathsf{sensible}) > 1 - \delta$
(ii) for each $\mathcal{T}$, averaging over x,y: $\mathsf{P}(\mathcal{T}\ \ \mathsf{answers}\ \ \mathsf{x} \overset{?}{\sqsubseteq} \mathsf{y}) > 1 - \epsilon$

However, even if provable, it likely cannot be proved constructively: given an amount of evidence E, we may be able to make random assumptions until $1 - \epsilon$ of all queries are known to be provable. Furthermore, it is likely to be uncomputable how much evidence $\mathsf{E}(\epsilon, \delta)$ is necessary to achieve given bounds ϵ, δ:

Conjecture 7.4.4. *The evidence function*

$$\mathsf{E}'(\mathsf{m},\mathsf{n}) = \mathsf{ceil}(\ \ -\log(1\ \ -\ \ \mathsf{E}(\exp(-\mathsf{m}), \exp(-\mathsf{n})))\ \ )$$

is not recursively bounded.

Propagating evidence

We now describe Johann's algorithm for generating conjectures. Despite the severe limitations on the possibility of generating *true* conjectures, we have had considerable practical success generating *useful* conjectures, some of which were proven true by other means, and some of which were false but led to insight as to what axioms could improve Johann's reasoning ability.

We generate conjectures by propagating evidence for questions $\mathsf{x} \overset{?}{\sqsubseteq} \mathsf{y}$, and then collecting the best few conjectures, typically 10^2 out of 10^7 total. Evidence acclumulates through four probabilistic inference rules for plausibility, the inverses of the monotonicity and transitivity rules of for order.

$$\frac{\mathsf{f\ \ term} \qquad \mathsf{f\ x} \sqsubseteq \mathsf{f\ \ y\ \ plaus}}{\mathsf{x} \sqsubseteq \mathsf{y\ \ plaus}}\,(\mu) \qquad\qquad \frac{\mathsf{f\ \ x} \sqsubseteq \mathsf{g\ \ x\ \ plaus} \qquad \mathsf{x\ \ term}}{\mathsf{f} \sqsubseteq \mathsf{g\ \ plaus}}\,(\nu)$$

$$\frac{\mathsf{x} \sqsubseteq \mathsf{y\ \ true} \qquad \mathsf{y\ \ term} \qquad \mathsf{y} \sqsubseteq \mathsf{z\ \ plaus}}{\mathsf{x} \sqsubseteq \mathsf{z\ \ plaus}}\,(\tau) \qquad\qquad \frac{\mathsf{x} \sqsubseteq \mathsf{y\ \ plaus} \qquad \mathsf{y\ \ term} \qquad \mathsf{y} \sqsubseteq \mathsf{z\ \ true}}{\mathsf{x} \sqsubseteq \mathsf{z\ \ plaus}}\,(\tau')$$

Now given a database of facts $\mathsf{u} \sqsubseteq \mathsf{v}$ and $\mathsf{u} \not\sqsubseteq \mathsf{v}$, a complexity pmf $\mathsf{P(x)}$ on terms, and rule weights $\mathsf{P}_\mu + \mathsf{P}_\nu + 2\ \mathsf{P}_\tau = 1$, we iteratively propagate evidence $\mathsf{P}(\mathsf{u} \sqsubseteq \mathsf{v})$ to find a fixedpoint of the contraction equation

$$\begin{aligned}\mathsf{P}(\mathsf{u} \sqsubseteq \mathsf{v}) \;&= \textbf{if } \mathsf{u} \sqsubseteq \mathsf{v} \textbf{ then } 1 \textbf{ else} \\ &\quad\ \textbf{if } \mathsf{u} \not\sqsubseteq \mathsf{v} \textbf{ then } 0 \textbf{ else} \\ &\quad (\ \mathsf{P}_\mu \textstyle\sum_\mathsf{f}\ \mathsf{P(f)}\ \mathsf{P(f\ u} \sqsubseteq \mathsf{x\ v)} \\ &\quad +\ \mathsf{P}_\nu \textstyle\sum_\mathsf{x}\ \mathsf{P(u\ x} \sqsubseteq \mathsf{v\ x)}\ \mathsf{P(x)} \\ &\quad +\ \mathsf{P}_\tau \textstyle\sum_\mathsf{y}\ \textbf{if } \mathsf{u} \sqsubseteq \mathsf{y} \textbf{ then } \mathsf{P(y)}\ \mathsf{P(y} \sqsubseteq \mathsf{v)} \textbf{ else } 0 \\ &\quad +\ \mathsf{P}_\tau \textstyle\sum_\mathsf{y}\ \mathsf{P(u} \sqsubseteq \mathsf{y)}\ \mathsf{P(y)}\ \textbf{if } \mathsf{y} \sqsubseteq \mathsf{v} \textbf{ then } 1 \textbf{ else } 0\ )\end{aligned}$$

Note that normalization ensures $\mathsf{P}(\mathsf{u} \sqsubseteq \mathsf{v}) \in [0, 1]$.

Each propagation iteration takes time cubic in database size, since we need to examine each unproved questions $\mathsf{u} \overset{?}{\sqsubseteq} \mathsf{v}$ (roughly $0.05 \cdot O^2 \sim 5 \cdot 10^6$ for $O = 10^4$ obs), and for each question, sum over all f's, x's, and y's above. Because the τ rule is significantly slower than either of the μ, ν rules, we often leave it out by setting $\mathsf{P}_\tau = 0$.

7.5 Fitting a basis to a corpus of expressions

Johann enumerates a finite chunk of an infinite structure, by randomly adding simple terms to the database. Building this chunk has quadratic space complexity and cubic time complexity in the number of obs or equivalence classes. Thus it is very important that Johann has a "correct" notion of simplicity or complexity —that the weighted combinatory basis reflects our interest in the infinite structure.

This section describes a basis fitting algorithm that Johann uses to statistically fit the probabilistic basis to a corpus of "interesting" terms, e.g., the expressions appearing in this thesis. We basically fit a bag-of-words-modulo-equality model to maximize the likelihood of the corpus, or minimize KL-divergence between the corpus and the model. This local weight optimization procedure is similar to the standard bag-of-words model, but can use Johann's equational database to balance multiple possible parses of a given expression. We also describe a basis extension procedure that search the database for possible obs to add to the basis.

Local optimization

We begin with the standard bag-of-words model over an algebraic signature. Let

$$\mathsf{L} = \{\mathsf{app}@\mathsf{P}_\mathsf{app},\ \mathbf{S}@\mathsf{P}_\mathbf{S},\ \mathbf{K}@\mathsf{P}_\mathbf{K},\ \ \dots\}$$

be a probabilistic basis (a probabilistic context-free grammar [JLM92]) generating a probabilistic language

$$\mathsf{L}^* = \{\mathbf{S}@\mathsf{P}_\mathbf{S},\ \mathbf{K}@\mathsf{P}_\mathbf{K},\ \mathbf{S}\ \mathbf{S}\ @\ \mathsf{P}_\mathsf{app}\ \mathsf{P}_\mathbf{S}\ \mathsf{P}_\mathbf{S},\ \mathbf{S}\ \mathbf{K}\ @\ \mathsf{P}_\mathsf{app}\ \mathsf{P}_\mathbf{S}\ \mathsf{P}_\mathbf{K},\ \ \dots\}$$

and let C be a corpus of expressions, a probability distribution over the support of L^*. Note that not all such languages have finite entropy; the infinite entropy-languages are meaningless, and not accessible by gradient descent methods ([ABNK+87], [AN93]). The Kullback-Leibler divergence from C to L^* is

$$H(C|L^*) = \sum_x C(x) \log \frac{C(x)}{L^*(x)}$$

We now solve the local optimization problem

Given:	a corpus C and initial basis $\mathbf{S}$
Vary:	weights $\mathsf{P}_\mathsf{app}, \mathsf{P}_\mathbf{S}, \mathsf{P}_\mathbf{K}, \dots$ of the basis $\mathbf{S}$
To Minimize:	$\mathsf{H}(\mathsf{C} \mid \mathsf{L}^*)$
Subject To:	$\mathbf{S}$ being a probability distribution,
	i.e. $0\ \le\ \mathsf{P_i}\ \le\ 1$ and $\sum_\mathsf{i}\ \mathsf{P_i} = 1$.

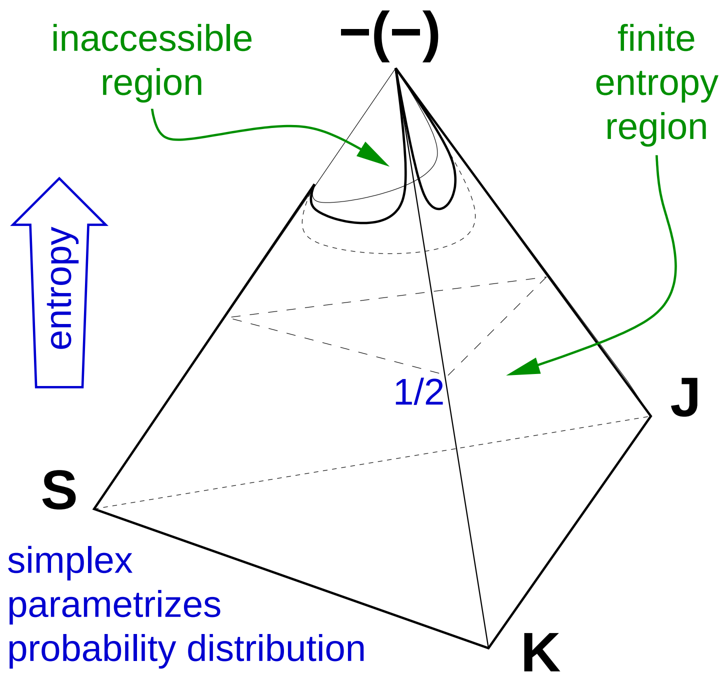

Figure 7.1: The parameter space of a weighted basis.

Next under an equational theory $\sim$, we perform the same optimization, with C and L^* probability distributions over *equivalence classes* of expressions modulo $\sim$. Thus the sum in $\mathsf{H}(\mathsf{C} \mid \mathsf{L}^*)$ now ranges over equivalence classes.

In practice we use a conjugate gradient optimization solver, that iteratively selects a conjugate gradient descent direction, and then line-searches in the chosen subspace. Each gradient computation and each objective function evaluation further requires iteration, as both complexity and relevance are defined as fixedpoints of propagation equations. When solution approaches a boundary, i.e., some atom is given extremely low weight, that atom is removed from the basis, and optimization continues with the reduced basis (as done in active set methods). We have also experimented with using the natural information metric to compute the conjugate direction ([Ama98]), but had problems with non-convergence and NaNs near boundaries.

Extending the basis

Thus far, we have shown how to locally optimize a basis with given support. This naturally leads to a naive algorithm to extend the support with additional atoms:

```
For each ob in database support:
    add ob to basis;  locally optimize;
    if cost decreased then keep ob else remove
```

However this naive algorithm has two problems:
(1) it is too slow to iterate through all $\sim 10\mathsf{k}$ obs; and more importantly
(2) the optimal result would be complete overfitting, i.e., defining $\mathsf{Basis} = \mathsf{Corpus}$, $\mathsf{P}_{\mathsf{app}} = 0$, and never generating new ideas or parses of existing obs.

Our solutions are (1) to locally estimate how important an ob *would be* if added to the basis, and (2) to manually split the large terms in the corpus into polynomials in $\mathsf{P}_{\mathsf{app}}$ and their subterms below some complexity threshold.

To estimate the improvement any one ob x would have in extending the basis, we compute the partial derivative of cost w.r.t. basis weight of an ob x

$$R(x) = \left. \frac{\partial H(C|(\delta \mathsf{x} + (1-\delta)L)^*)}{\partial \delta} \right|_{\delta=0}$$

It turns out that we have already computed this quantity under the name of "relevance" of an ob x to a corpus C. Thus to extend the basis, we can search among the best few (e.g. $|L|$ or $\log(O)$) extension candidates, then locally optimize, as in the naive case.

To deal with overfitting, we split the corpus (a weighted set of obs) into an ob polynomial in $\mathsf{P}_{\mathsf{app}}$, so that each ob has complexity below some threshhold. For example we might split up

$$\begin{aligned} \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ +\ \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ &\mapsto\ \mathsf{P}_{\mathsf{app}}\ +\ \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ +\ \mathbf{S}\ +\ \mathbf{S}\ \mathbf{S}\ \mathbf{S}\ \mathbf{S} \\ &\mapsto\ 2\ \mathsf{P}_{\mathsf{app}}\ +\ 2(\mathbf{S}\ \mathbf{S}\ \mathbf{S})\ +\ \mathbf{S} \end{aligned}$$

Because this splitting puts $\mathsf{P}_{\mathsf{app}}$ into the objective function, a global extremum cannot completely fit the corpus, as it must give $\mathsf{P}_{\mathsf{app}}$ some positive value.

Appendix A

Syntactic algorithms

A.1 Term syntax

We first present a relatively large language of terms, and then show how to compile terms down to a small combinatory fragment.

Algebraic Terms

$$\frac{\mathsf{x}\ \ \mathsf{var}}{\mathsf{x}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{I}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{K}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{B}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{C}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{W}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{S}\ \ \mathsf{term}} \qquad \frac{}{\mathbf{J}\ \ \mathsf{term}} \qquad \frac{}{\mathsf{code}\ \ \mathsf{term}}$$

$$\frac{}{\mathsf{quote}\ \ \mathsf{term}} \qquad \frac{}{\mathsf{eval}\ \ \mathsf{term}} \qquad \frac{\mathsf{M}\ \ \mathsf{term} \qquad \mathsf{N}\ \ \mathsf{term}}{\mathsf{M}\ \mathsf{N}\ \ \mathsf{term}} \qquad \frac{\mathsf{M}\ \ \mathsf{term} \qquad \mathsf{N}\ \ \mathsf{term}}{\mathsf{M} \mid \mathsf{N}\ \ \mathsf{term}} \qquad \frac{\mathsf{M}\ \ \mathsf{term} \qquad \mathsf{N}\ \ \mathsf{term}}{\mathsf{M}\circ\mathsf{N}\ \ \mathsf{term}}$$

$$\frac{\mathsf{M}\ \ \mathsf{term} \qquad \mathsf{N}\ \ \mathsf{term}}{\mathsf{M}\to\mathsf{N}\ \ \mathsf{term}} \qquad \frac{\mathsf{M}_1\ \ \mathsf{term} \qquad \dots \qquad \mathsf{M}_n\ \ \mathsf{term}}{\langle \mathsf{M}_1,\dots,\mathsf{M}_n\rangle\ \ \mathsf{term}}\,(n\geq 0) \qquad \frac{\mathsf{M}\ \ \mathsf{term}}{\{\mathsf{M}\}\ \ \mathsf{term}} \qquad \frac{\phi\ \ \mathsf{stmt}}{\{\phi\}\ \ \mathsf{term}}$$

Patterns and Binding

$$\frac{\mathsf{x}\ \ \mathsf{var}}{\mathsf{x}\ \ \mathsf{patt}'} \qquad \frac{\mathsf{p}_1\ \ \mathsf{patt} \qquad \dots \qquad \mathsf{p}_n\ \ \mathsf{patt}}{\langle \mathsf{p}_1,\dots,\mathsf{p}_n\rangle\ \ \mathsf{patt}'} \qquad \frac{\mathsf{p}\ \ \mathsf{patt}}{\{\mathsf{p}\}\ \ \mathsf{patt}'} \qquad \frac{\mathsf{p}\ \ \mathsf{patt}'}{\mathsf{p}\ \ \mathsf{patt}} \qquad \frac{\mathsf{p}\ \ \mathsf{patt}' \qquad \mathsf{M}\ \ \mathsf{term}}{\mathsf{p}:\mathsf{M}\ \ \mathsf{patt}}$$

$$\frac{\mathsf{p}\ \ \mathsf{patt}' \qquad \mathsf{M}\ \ \mathsf{term}}{\mathsf{p}::\mathsf{M}\ \ \mathsf{patt}} \qquad \frac{\mathsf{p}\ \ \mathsf{patt}' \qquad \mathsf{M}\ \ \mathsf{term} \qquad \mathsf{N}\ \ \mathsf{term}}{\mathsf{p}:\mathsf{M}::\mathsf{N}\ \ \mathsf{patt}} \qquad \frac{\mathsf{p}\ \ \mathsf{patt} \qquad \mathsf{M}\ \ \mathsf{term} \qquad \mathsf{N}\ \ \mathsf{term}}{\mathsf{p} := \mathsf{M}.\mathsf{N}\ \ \mathsf{term}}$$

$$\frac{\mathsf{p}\ \ \mathsf{patt} \qquad \mathsf{M}\ \ \mathsf{term}}{\lambda\mathsf{p}.\mathsf{M}\ \ \mathsf{term}} \qquad \frac{\mathsf{p}\ \ \mathsf{patt} \qquad \mathsf{M}\ \ \mathsf{term}}{\forall\mathsf{p}.\mathsf{M}\ \ \mathsf{term}} \qquad \frac{\mathsf{p}\ \ \mathsf{patt} \qquad \mathsf{M}\ \ \mathsf{term}}{\exists\mathsf{p}.\mathsf{M}\ \ \mathsf{term}}$$

A.2 Compiling to combinators

(large step; **R** is similar to **J**; quoting is handled separately)

Algebraic

$$\frac{M \mapsto M' \qquad N \mapsto N}{M \mid N \mapsto \mathbf{J}\ M'\ N'}\ (\mathbf{J}) \qquad \frac{}{\langle\rangle \mapsto \mathbf{I}} \qquad \frac{M \mapsto M'}{\langle M\rangle \mapsto \mathbf{C}\ \mathbf{I}\ M'}$$

$$\frac{M \mapsto M' \qquad \langle N_1,\dots,N_n\rangle \mapsto N'}{\langle M, N_1,\dots,N_n\rangle \mapsto \mathbf{C}\ M'\ N'}\ (n \geq 1)$$

Abstraction

$$\frac{x \ \text{apart}\ M}{\lambda x.M \mapsto \mathbf{K}\ x}\ (\mathbf{K}) \qquad \frac{}{\lambda x.x \mapsto \mathbf{I}}\ (\mathbf{I}) \qquad \frac{x \ \text{apart}\ M}{\lambda x.M\ x \mapsto M}\ (\eta) \qquad \frac{x \ \text{apart}\ M \qquad \lambda x.N \mapsto N'}{\lambda x.M\ N \mapsto \mathbf{B}\ M\ N'}\ (\mathbf{B})$$

$$\frac{\lambda x.M \mapsto M'}{\lambda x.M\ x \mapsto \mathbf{W}\ M'}\ (\mathbf{W}) \qquad \frac{x \ \text{apart}\ N \qquad \lambda x.M \mapsto M'}{\lambda x.M\ N \mapsto \mathbf{C}\ M'\ N}\ (\mathbf{C}) \qquad \frac{\lambda x.M \mapsto M' \qquad \lambda x.N \mapsto N'}{\lambda x.M\ N \mapsto \mathbf{S}\ M'\ N'}\ (\mathbf{S})$$

$$\frac{\lambda x.M \mapsto M' \qquad \lambda x.N \mapsto N}{\lambda x.M \mid N \mapsto \mathbf{J}\ M'\ N'}\ (\mathbf{J}) \qquad \frac{\lambda x.M \mapsto M' \qquad \mathbf{V}\ a \mapsto a'}{\lambda x{:}a.M \mapsto \mathbf{B}\ M'\ a'} \qquad \frac{\lambda x.M \mapsto M' \qquad \text{test}\ t \mapsto t'}{\lambda x{::}t.M \mapsto \mathbf{S}\ t'\ M'}$$

$$\frac{\lambda x.M \mapsto M' \qquad \mathbf{V}\ a \mapsto a' \qquad \text{test}\ t \mapsto t'}{\lambda x{:}a{::}t.M \mapsto \mathbf{B}\ (\mathbf{S}\ t'\ M')\ a'} \qquad \frac{\lambda p_1,\dots,p_n.M \mapsto M'}{\lambda\langle p_1,\dots,p_n\rangle.M \mapsto \mathbf{C}\ \mathbf{I}\ M'}$$

Definitions

$$\frac{x \ \text{apart}\ M}{x := D.N \mapsto M} \qquad \frac{x \ \text{apart}\ M \qquad x := D.N \mapsto N'}{x := D.M\ N \mapsto M\ N'} \qquad \frac{x \ \text{apart}\ N \qquad x := D.M \mapsto M'}{x := D.M\ N \mapsto M'\ N}$$

$$\frac{\lambda x.M \mapsto M'}{x := D.M\ x \mapsto \mathbf{W}\ M'\ D} \qquad \frac{\lambda x.M \mapsto M' \qquad \lambda x.N \mapsto N}{x := D.M\ N \mapsto \mathbf{S}\ M'\ N'\ D} \qquad \frac{x_1 := D_1 \dots .x_n := D_n.M \mapsto M'}{\langle x_1,\dots,x_n\rangle := \langle D_1,\dots,D_n\rangle.M \mapsto M'}$$

$$\frac{\lambda x_1,\dots,x_n.M \mapsto M'}{\langle x_1,\dots,x_n\rangle := D.M \mapsto D\ M'}$$

Quantifiers / Polymorphism

$$\frac{\mathbf{V}\ \lambda x,y.\mathbf{V}\ M(y\ x) \mapsto M'}{\forall x.M \mapsto M'} \qquad \frac{\mathbf{V}\ \lambda x{:}N,y.\mathbf{V}\ M(y\ x) \mapsto M'}{\forall x{:}N.M \mapsto M'} \qquad \frac{\lambda\langle x,y\rangle{:}\text{pair}.\langle x, \mathbf{V}\ M\ y\rangle \mapsto M'}{\exists x.M \mapsto M'}$$

$$\frac{\lambda\langle x,y\rangle{:}\text{pair}.\langle \mathbf{V}\ N\ x, \mathbf{V}\ M\ y\rangle \mapsto M'}{\exists x{:}N.M \mapsto M'}$$

Quoting is more delicate, as it involves the code comonad. We need to define two types of compiling: one inside quotes, and one outside. Within a quoted environment, we can quote variables. Also, all compilation is done at the outer-most environment, so that, e.g., the abstraction algorithm works with $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{W}, \mathbf{S}$ instead of the more complicated quoted versions $\underline{\mathbf{I}}, \underline{\mathbf{K}}, \underline{\mathbf{B}}, \underline{\mathbf{C}}, \underline{\mathbf{W}}, \underline{\mathbf{S}}$.

Quoting

$$\frac{\{M\} \mapsto M' \qquad \{N\} \mapsto N'}{\{M\ N\} \mapsto \underline{\mathbf{A}}\ M'\ N'} \qquad \frac{\{M\} \mapsto M'}{\{\{M\}\} \mapsto \mathsf{quote}\ M'}$$

Code Comonad

$$\frac{}{\lambda\{x\}.\{x\} \mapsto \mathsf{code}} \qquad \frac{}{\lambda\{x{:}a\}.\{x\} \mapsto \mathsf{Code}\{a\}} \qquad \frac{}{\lambda\{x\}.x \mapsto \mathbf{E}} \qquad \frac{}{\lambda\{x{:}a\}.\{x\} \mapsto \mathsf{Eval}\{a\}}$$

$$\frac{}{\lambda\{x\}.\{\{x\}\} \mapsto \mathbf{Q}} \qquad \frac{}{\lambda\{x{:}a\}.\{\{x\}\} \mapsto \mathsf{Quote}\{a\}} \qquad \frac{\{x\}\ \mathsf{apart}\ M \qquad x\ \mathsf{not\ apart}\ M \qquad \lambda x.M \mapsto M'}{\lambda\{x\}.M \mapsto \mathbf{B}\ M'\ \mathbf{E}}$$

$$\frac{x\ \mathsf{apart}\ M}{\lambda\{x\}.M = \mathbf{K}\ M}(\{\mathbf{K}\}) \qquad \frac{\lambda\{x\}.M \mapsto \mathbf{K}\ M' \qquad \lambda\{x\}.N \mapsto \mathbf{B}\ N'\ \mathsf{code}}{\lambda\{x\}.M\ N \mapsto \mathbf{B}(\mathbf{B}\ M'\ N')\mathsf{code}}(\{\mathbf{B}\})$$

$$\frac{\lambda\{x\}.M \mapsto \mathbf{B}\ M'\ \mathsf{code} \qquad \lambda\{x\}.N \mapsto \mathbf{K}\ N'}{\lambda\{x\}.M\ N \mapsto \mathbf{B}(\mathbf{C}\ M'\ N')\mathsf{code}}(\{\mathbf{C}\})$$

$$\frac{\lambda\{x\}.M \mapsto \mathbf{B}\ M'\ \mathsf{code} \qquad \lambda\{x\}.N \mapsto \mathbf{B}\ N'\ \mathsf{code}}{\lambda\{x\}.M\ N \mapsto \mathbf{B}(\mathbf{S}\ M'\ N')\mathsf{code}}(\{\mathbf{S}\}) \qquad \frac{\{M\} \mapsto M' \qquad \lambda\{x\}.M' \mapsto M''}{\lambda\{x\}.M \mapsto M''}(\{\mathsf{code}\})$$

$$\frac{}{\lambda\{x\}.\ M \mapsto \lambda c{:}\mathsf{code}.\ [x := \mathbf{E}\ c,\ \{x\} := c,\ \{\{x\}\} := \mathbf{Q}\ c]\ M}$$

Note that the $\{\mathbf{K}\}$ rule prevents us from testing with test_code; we need to abstract partial terms as well.

A.3 Decompiling

One of the advancements of this is a successful decompiling algorithm from combinators SKJ to λ-let-terms. Although this is theoretically banal, it has major practical significance in allowing people to read combinators (and foiling the obfuscatory attempts of languages like unlambda, and Iota).

The main idea is to simplify (with Johann's database), translate left-linear terms to λ abstraction, translate left-copying terms ($\mathbf{W}, \mathbf{S}$) to let-expressions, decompile vectors $\lambda f.f\ M_1\ \ldots M_n \leftrightarrow \langle M_1, \ldots, M_n\rangle$, and η-reduce whenever possible.

Abstraction

$$\frac{}{\mathbf{I} \;\mapsfrom\; \lambda x.x} \qquad \frac{M\ \underline{N} \;\mapsfrom\; M'}{\mathbf{I}\ M\ \underline{N} \;\mapsfrom\; M'} \qquad \frac{}{\mathbf{K} \;\mapsfrom\; \lambda x,y.x} \qquad \frac{M \;\mapsfrom\; M'}{\mathbf{K}\ M \;\mapsfrom\; \lambda\bot.M'} \qquad \frac{M\ \underline{\mathbf{O}} \;\mapsfrom\; M'}{\mathbf{K}\ M\ N\ \underline{\mathbf{O}} \;\mapsfrom\; M'}$$

$$\frac{}{\mathbf{B} \;\mapsfrom\; \lambda x,y,z.x(y\ z)} \qquad \frac{M(y\ z) \;\mapsfrom\; M'}{\mathbf{B}\ M \;\mapsfrom\; \lambda y,z.M'} \qquad \frac{M(N\ z) \;\mapsfrom\; M'}{\mathbf{B}\ M\ N \;\mapsfrom\; \lambda z.M'} \qquad \frac{L(M\ N) \;\mapsfrom\; M'}{\mathbf{B}\ L\ M\ N \;\mapsfrom\; M'}$$

$$\frac{}{\mathbf{C} \;\mapsfrom\; \lambda x,y,z.x\ z\ y} \qquad \frac{M\ z\ y \;\mapsfrom\; M'}{\mathbf{C}\ M \;\mapsfrom\; \lambda y,z.M'} \qquad \frac{M\ z\ N \;\mapsfrom\; M'}{\mathbf{C}\ M\ N \;\mapsfrom\; \lambda z.M'} \qquad \frac{L\ N\ M \;\mapsfrom\; M'}{\mathbf{C}\ L\ M\ N \;\mapsfrom\; M'}$$

$$\frac{}{\mathbf{W} \;\mapsfrom\; \lambda x,y.x\ y\ y} \qquad \frac{M\ y\ y \;\mapsfrom\; M'}{\mathbf{W}\ M \;\mapsfrom\; \lambda y.M'} \qquad \frac{}{\mathbf{S} \;\mapsfrom\; \lambda x,y,z.x\ z(y\ z)} \qquad \frac{M\ z(y\ z) \;\mapsfrom\; M'}{\mathbf{S}\ M \;\mapsfrom\; \lambda y,z.M'}$$

$$\frac{M\ z(N\ z) \;\mapsfrom\; M'}{\mathbf{S}\ M\ N \;\mapsfrom\; \lambda z.M'}$$

Definitions

$$\frac{M\ x\ \underline{N} \;\mapsfrom\; M'}{\mathbf{W}\ M\ D\ \underline{N} \;\mapsfrom\; x := D.M'} \qquad \frac{M\ x(N\ x)\underline{\mathbf{O}} \;\mapsfrom\; M'}{\mathbf{S}\ M\ N\ D\ \underline{\mathbf{O}} \;\mapsfrom\; x := D.M'} \qquad \frac{M\ x \;\mapsfrom\; M' \qquad N\ x \;\mapsfrom\; N' \qquad \underline{L} \;\mapsfrom\; \underline{L}'}{\mathbf{J}\ M\ N\ \underline{L} \;\mapsfrom\; (x := D.\ M' \mid N')\ \underline{L}}$$

Join (Rand is identical)

$$\frac{}{\mathbf{J} \;\mapsfrom\; \lambda x,y.x \mid y} \qquad \frac{M \;\mapsfrom\; M'}{\mathbf{J}\ M \;\mapsfrom\; \lambda x.M' \mid x}$$

$$\frac{M\ x \;\mapsfrom\; M' \qquad x \text{ apart } M' \qquad N\ x \;\mapsfrom\; N' \qquad x \text{ apart } N' \qquad \mathbf{J}\ M'\ N'\ \underline{L} \;\mapsfrom\; L'}{\mathbf{J}\ M\ N\ D\ \underline{L} \;\mapsfrom\; L'}$$

$$\frac{M\ x \;\mapsfrom\; M' \qquad x \text{ apart } M' \qquad N\ x \;\mapsfrom\; N' \qquad \mathbf{J}\ M'\ (N\ D)\ \underline{L} \;\mapsfrom\; L'}{\mathbf{J}\ M\ N\ D\ \underline{L} \;\mapsfrom\; L'}$$

$$\frac{M\ x \;\mapsfrom\; M' \qquad N\ x \;\mapsfrom\; N' \qquad x \text{ apart } N' \qquad \mathbf{J}\ (M\ D)\ N'\ \underline{L} \;\mapsfrom\; L'}{\mathbf{J}\ M\ N\ D\ \underline{L} \;\mapsfrom\; L'}$$

Vectors and Tuples

$$\frac{M \;\mapsfrom\; M'}{\mathbf{C}\ \mathbf{I}\ M \;\mapsfrom\; \langle M'\rangle} \qquad \frac{M \;\mapsfrom\; M_1 \qquad N \;\mapsfrom\; \mathbf{C}\ \langle N_2,\ldots,N_n\rangle}{\mathbf{C}\ M\ N \;\mapsfrom\; \langle M, N_1,\ldots,N_n\rangle}\ (n \geq 2) \qquad \frac{M \;\mapsfrom\; M' \qquad N \;\mapsfrom\; \langle N_1, N_2\rangle}{\langle M, \langle N_1, N_2\rangle\rangle \;\mapsfrom\; (M, N_1, N_2)}$$

$$\frac{N \;\mapsfrom\; \langle N_1, N_2\rangle}{(M_1,\ldots,M_n,N) \;\mapsfrom\; (M_1,\ldots,M_n,N_1,N_2)}\ (n \geq 2)$$

Codes

$$\frac{M \;\mapsfrom\; \{M'\} \qquad N \;\mapsfrom\; \{N'\}}{\mathbf{A}\ M\ N \;\mapsfrom\; \{M\ N\}} \qquad \frac{M \;\mapsfrom\; \{M'\}}{\mathbf{E}\ M \;\mapsfrom\; M'} \qquad \frac{M \;\mapsfrom\; \{M'\}}{\mathbf{Q}\ M \;\mapsfrom\; \{\{M'\}\}}$$

A.4 Reduction Strategies

We consider both affine/linear reduction $\twoheadrightarrow$ and bounded reduction $\rightsquigarrow$. Since affine reduction is size-reducing (and hence terminates), we count only nonlinear steps towards the bound in arbitrary reduction.

Affine

$$\frac{}{\bot\ \mathsf{x} \twoheadrightarrow \bot} \qquad \frac{}{\top\ \mathsf{x} \twoheadrightarrow \top} \qquad \frac{}{\mathbf{I}\ \mathsf{x} \twoheadrightarrow \mathsf{x}} \qquad \frac{}{\mathbf{J}\ \top \twoheadrightarrow \top} \qquad \frac{}{\mathbf{J}\ \bot \twoheadrightarrow \mathbf{I}} \qquad \frac{}{\mathbf{V}\ \mathbf{V} \twoheadrightarrow \mathbf{V}}$$

$$\frac{}{\mathbf{V}(\mathbf{V}\ \mathsf{x}) \twoheadrightarrow \mathbf{V}\ \mathsf{x}} \qquad \frac{}{\mathbf{U}\ \mathbf{U} \twoheadrightarrow \mathbf{U}} \qquad \frac{}{\mathbf{U}(\mathbf{U}\ \mathsf{x}) \twoheadrightarrow \mathbf{U}\ \mathsf{x}} \qquad \frac{}{\mathbf{K}\ \mathsf{x}\ \mathsf{y} \twoheadrightarrow \mathsf{x}} \qquad \frac{}{\mathbf{F}\ \mathsf{x}\ \mathsf{y} \twoheadrightarrow \mathsf{y}}$$

$$\frac{}{\mathbf{J}\ \mathsf{x}\ \top \twoheadrightarrow \top} \qquad \frac{}{\mathbf{J}\ \mathsf{x}\ \bot \twoheadrightarrow \mathsf{x}} \qquad \frac{}{\mathbf{J}\ \mathsf{x}\ \mathsf{x} \twoheadrightarrow \mathsf{x}} \qquad \frac{}{\mathbf{R}\ \mathsf{x}\ \mathsf{x} \twoheadrightarrow \mathsf{x}} \qquad \frac{}{\mathbf{B}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{x}(\mathsf{y}\ \mathsf{z})}$$

$$\frac{}{\mathbf{C}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{x}\ \mathsf{z}\ \mathsf{y}}$$

Nonaffine

$$\frac{}{\mathbf{Y}\ \mathsf{x} \rightsquigarrow \mathsf{x}(\mathbf{Y}\ \mathsf{x})} \qquad \frac{}{\mathbf{V}\ \mathsf{f} \rightsquigarrow \mathbf{I} \mid \mathsf{f}\circ(\mathbf{V}\ \mathsf{f})} \qquad \frac{}{\mathbf{U}\ \mathsf{f} \rightsquigarrow \mathsf{f}\circ(\mathbf{U}\ \mathsf{f}) \mid \mathsf{f}} \qquad \frac{}{\mathbf{W}\ \mathsf{x}\ \mathsf{y} \rightsquigarrow \mathsf{x}\ \mathsf{y}\ \mathsf{y}}$$

$$\frac{}{\mathbf{W}\ \mathbf{W}\ \mathbf{W} \rightsquigarrow \bot} \qquad \frac{\mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{u}}{\mathbf{S}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \rightsquigarrow \mathsf{x}\ \mathsf{z}\ \mathsf{u}} \qquad \frac{\mathsf{x}\ \mathsf{z} \twoheadrightarrow \mathsf{u} \qquad \mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{v}}{\mathbf{J}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \rightsquigarrow \mathbf{J}\ \mathsf{u}\ \mathsf{v}} \qquad \frac{\mathsf{x}\ \mathsf{z} \twoheadrightarrow \mathsf{u} \qquad \mathsf{y}\ \mathsf{z} \twoheadrightarrow \mathsf{v}}{\mathbf{R}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} \rightsquigarrow \mathbf{R}\ \mathsf{u}\ \mathsf{v}}$$

$$\frac{}{\mathbf{R}\ \mathsf{x}(\mathbf{J}\ \mathsf{y}\ \mathsf{z}) \rightsquigarrow \mathbf{J}(\mathbf{R}\ \mathsf{x}\ \mathsf{y})(\mathbf{R}\ \mathsf{x}\ \mathsf{z})}$$

A.5 Statements

Statements are based on equations and Scott's information ordering. In SKJ and extensions we can also reason about types-as-closures (3.3) and types-as-tests (3.10).

Statement Formation

$$\frac{}{=\ \mathsf{rel}} \qquad \frac{}{\neq\ \mathsf{rel}} \qquad \frac{}{\sqsubseteq\ \mathsf{rel}} \qquad \frac{}{\not\sqsubseteq\ \mathsf{rel}} \qquad \frac{}{\sqsupseteq\ \mathsf{rel}} \qquad \frac{}{\not\sqsupseteq\ \mathsf{rel}} \qquad \frac{}{\sqsubsetneq\ \mathsf{rel}} \qquad \frac{}{\sqsupsetneq\ \mathsf{rel}} \qquad \frac{}{:\ \mathsf{rel}} \qquad \frac{}{!:\ \mathsf{rel}}$$

$$\frac{}{<:\ \mathsf{rel}} \qquad \frac{}{!<:\ \mathsf{rel}} \qquad \frac{}{:>\ \mathsf{rel}} \qquad \frac{}{!:>\ \mathsf{rel}} \qquad \frac{}{::\ \mathsf{rel}} \qquad \frac{}{!::\ \mathsf{rel}} \qquad \frac{}{<::\ \mathsf{rel}} \qquad \frac{}{!<::\ \mathsf{rel}} \qquad \frac{}{::>\ \mathsf{rel}}$$

$$\frac{}{!::>\ \mathsf{rel}} \qquad \frac{\mathsf{M}\ \mathsf{term} \qquad \sim\ \mathsf{rel} \qquad \mathsf{N}\ \mathsf{term}}{\mathsf{M} \sim \mathsf{N}\ \mathsf{stmt}} \qquad \frac{\mathsf{x}\ \mathsf{patt} \qquad \mathsf{p}\ \mathsf{stmt}}{\forall \mathsf{x}.\ \mathsf{p}\ \mathsf{stmt}} \qquad \frac{\mathsf{x}\ \mathsf{patt} \qquad \mathsf{M}\ \mathsf{term} \qquad \mathsf{p}\ \mathsf{stmt}}{\mathsf{x} := \mathsf{M}.\ \mathsf{p}\ \mathsf{stmt}}$$

$$\frac{\mathsf{p}\ \mathsf{stmt}}{\mathbf{NOT}\ \ \mathsf{p}\ \mathsf{stmt}} \qquad \frac{\mathsf{p}\ \mathsf{stmt} \qquad \mathsf{q}\ \mathsf{stmt}}{\mathsf{p}\ \ \mathbf{AND}\ \ \mathsf{q}\ \mathsf{stmt}} \qquad \frac{\mathsf{p}\ \mathsf{stmt} \qquad \mathsf{q}\ \mathsf{stmt}}{\mathsf{p} \implies \mathsf{q}\ \mathsf{stmt}}$$

In SKJ and extensions, we can eliminate universal quantifiers, universal quantifiers, and implications where the hypothesis is a testable assertion $\mathsf{x}::\mathsf{t}$ or $\mathsf{M} = \mathbf{I}$.

Let $\sim$ denote one of the basic relations $=$, $\sqsubseteq$, or $\sqsupseteq$.

Derived Relations

$$\frac{}{\mathsf{M} \sqsubsetneq \mathsf{N} \quad \mapsto \quad \mathsf{M} \sqsubseteq \mathsf{N} \;\; \mathbf{AND} \;\; \mathsf{M} \not\sqsupseteq \mathsf{N}} \qquad \frac{}{\mathsf{M} : \mathsf{N} \quad \mapsto \quad \mathbf{V} \;\; \mathsf{M} \;\; \mathsf{N} = \mathsf{M}} \qquad \frac{}{\mathsf{M} <: \mathsf{N} \quad \mapsto \quad \mathbf{V} \;\; \mathsf{M} : \mathbf{P} \;\; \mathsf{N}}$$

$$\frac{}{\mathsf{M} :: \mathsf{N} \quad \mapsto \quad \mathsf{test} \;\; \mathsf{N} \;\; \mathsf{M} = \mathbf{I}} \qquad \frac{}{\mathsf{M} <:: \mathsf{N} \quad \mapsto \quad \mathsf{M} = \mathbf{P}_{\mathsf{test}} \;\; \mathsf{N} \;\; \mathsf{M}}$$

Universal Quantification

$$\frac{\mathsf{p} \quad \mapsto \quad \mathsf{M} \sim \mathsf{N}}{\forall \mathsf{x}.\mathsf{p} \quad \mapsto \quad (\lambda \mathsf{x}.\mathsf{M}) \sim (\lambda \mathsf{x}.\mathsf{N})} \qquad \frac{\mathsf{p} \quad \mapsto \quad \mathsf{M} \sim \mathsf{N}}{\forall \mathsf{x} : \mathsf{a}.\mathsf{p} \quad \mapsto \quad (\lambda \mathsf{x} : \mathsf{a}.\mathsf{M}) \sim (\lambda \mathsf{x} : \mathsf{a}.\mathsf{N})} \qquad \frac{\mathsf{p} \quad \mapsto \quad \mathsf{M} \sim \mathsf{N}}{\forall \mathsf{x} :: \mathsf{t}.\mathsf{p} \quad \mapsto \quad (\lambda \mathsf{x} :: \mathsf{t}.\mathsf{M}) \sim (\lambda \mathsf{x} :: \mathsf{t}.\mathsf{N})}$$

Hypotheses

$$\frac{\mathsf{q} \quad \mapsto \quad \mathsf{M} = \mathbf{I} \qquad \mathsf{p} \quad \mapsto \quad \mathsf{N} \sim \mathsf{N}'}{\mathsf{q} \implies \mathsf{p} \quad \mapsto \quad \mathsf{semi} \;\; \mathsf{M} \;\; \mathsf{N} \sim \mathsf{semi} \;\; \mathsf{M} \;\; \mathsf{N}'}$$

We also use the notation $\mathsf{x} : \mathsf{a} \vdash \mathsf{p}$ and $\mathsf{x} :: \mathsf{t} \vdash \mathsf{p}$ to denote x being bound as a term inhabiting a type or passing a test; these have the same semantics as universal quantification.

$$\mathsf{x} : \mathsf{a} :: \mathsf{t} \vdash \mathsf{M} \sim \mathsf{N} \quad \mapsto \quad (\lambda \mathsf{x} : \mathsf{a} :: \mathsf{t}.\mathsf{M})\mathsf{x} \sim (\lambda \mathsf{x} : \mathsf{a} :: \mathsf{t}.\mathsf{N})$$

Appendix B

Reflected axiom schemata

In this section we reflect the hard-coded axiom schemata from `cpp/axiom_enforcement.C` to achieve the strong transitivity principle "if $\vdash \phi$ then $\vdash$ pr{ϕ}" (used in 5.4). Note that axiom schemata without hypotheses such as $\mathbf{K}$ x y = x are deducible from equations; these schemata were hard-coded for efficiency rather than logical strength. Since we have already in 5.2 !assumed equations implying these principles, we only !assume schemata with hypotheses here.

First we define some derived provability semipredicates, extending those from 5.3.

```
if_pr_of_type  :=  (code → code → semi) (λ{x}, {a}. if_pr_equal{x}{V a x}).
if_pr_subtype  :=  (code → code → semi) (λ{a}, {b}. if_pr_equal{V a}{P a b}).
if_pr_passes   :=  (code → code → semi) (λ{x}, {t}. if_pr_equal{I}{test t x}).
if_pr_subtest  :=  (code → code → semi) (
    λ{s}, {t}. if_pr_equal{test s}{P_test s t}
).
!check (
    if_pr_of_type x y,
    if_pr_subtype x y,
    if_pr_passes x y,
    if_pr_subtest x y  ⊑  test_code x | test_code y
).
```

In the language of semibooleans and tests, we can express reasoning principles using and_semi to conjoin hypotheses and $\sqsubseteq$ to indicate implication between checked statements: semi (i.e. semiboolean values $\phi \sqsubseteq \mathbf{I}$). Similarly, reversible rules use equality for biimplication.

Order scheamata from 2.1.

```
!assume (∀{f}, {x}, {y}. if_pr_less{x}{y}  ⊑  if_pr_less{f x}{f y}).
!assume (∀{f}, {g}, {x}. if_pr_less{f}{g}  ⊑  if_pr_less{f x}{g x}).
```

Fixed-point schemata from 2.2

```
!assume (∀{y}. if_pr_equal{y}{S I y}  ⊑  if_pr_equal{Y}{y}).
!assume (∀{f}, {x}. if_pr_less{f x}{x}  ⊑  if_pr_less{Y f}{x}).
```

Join schemata from 3.1.

```
!assume (∀{f}, {x}, {y}. if_pr_less{x}{y}  ⊑  if_pr_less{f x}{f y}).
!assume (∀{f}, {g}, {x}. if_pr_less{f}{g}  ⊑  if_pr_less{f x}{g x}).
```

Universal retract schemata from 3.2.

```
!assume (∀{x}.  if_pr_equal{x}{U x} = if_pr_equal{x}{x∘x}).
!assume (∀{x}, {f}. if_pr_less{f x}{x}  ⊑  if_pr_equal{U f x}{x}).
```

Universal closure schemata from 3.3.

```
!assume (∀{x}.
    if_pr_of_type{x}{V} = and_semi (if_pr_equal{x}{x∘x}) (if_pr_less{I}{x})
).
!assume (∀{x}, {f}. if_pr_less{f x}{x}  ⊑  if_pr_equal{V f x}{x}).
```

Schemata for the types div, unit, semi from 3.8.

```
!assume (∀{x}.
    if_pr_nless{x}{⊥}  ⊑  if_pr_equal{div x}{T}    AND
    if_pr_equal{x ⊥}{⊥}  ⊑  if_pr_less{div}{x}
).

!assume (∀{x}.
    if_pr_nless{x}{I}  ⊑  if_pr_equal{unit x}{T}    AND
    if_pr_equal{x I}{I}  ⊑  if_pr_less{unit}{x}
).

!assume (∀{x}.
    if_pr_nless{x}{⊥}  ⊑  if_pr_less{I}{semi x}    AND
    if_pr_nless{x}{I}  ⊑  if_pr_equal{T}{semi x}   AND
    and_semi (if_pr_equal{x ⊥}{⊥})
             (if_pr_equal{x I}{I})  ⊑  if_pr_less{semi}{x}
).
```

A test intersection schema from 3.11.

```
!assume (∀{p}, {q}.
    and_semi (if_pr_less{check q}{check p})
             (if_pr_less{ div p }{ div q })  ⊑  if_pr_subtest{p}{q}
).
```

Code schemata from 5.2.

```
!assume (∀{x}, {y}.
    and_semi (if_pr_passes{x}{test_code})
             (if_pr_passes{y}{test_code})  ⊑  if_pr_equal{E(A x y)}{E x(E y)}
).

!assume (∀{c}, {c′}. (
    and_semi (if_pr_of_type{c}{code}).
    and_semi (if_pr_passes{c}{test_code}).
    and_semi (if_pr_of_type{c′}{code}).
    and_semi (if_pr_passes{c′}{check_code}).
             (if_pr_equal{E c}{E c′}))        ⊑  if_pr_less{c′}{c}
).

!assume (∀{c}, {c′}. (
    and_semi (if_pr_of_type{c}{code}).
    and_semi (if_pr_passes{c}{test_code}).
    and_semi (if_pr_of_type{c′}{code}).
             (if_pr_nless{E c′}{E c}))      ⊑  if_pr_equal{code(c | c′)}{T}.
)
```

Schemata for the oracle **O** from 5.5.

```
!assume (∀{p}, {q}.
    and_semi (if_pr_passes{{p}}{test_skj})
             (if_pr_subtest{p}{q})          ⊑  if_pr_equal{O{p}{q}}{I}
).

!assume (∀{p}, {p′}.
    and_semi (if_pr_passes{{p}}{test_skj})
             (if_pr_subtest{p}{p′})         ⊑  if_pr_subtest{O{p′}}{O{p}}
).

!assume (∀{q}, {q′}.
    if_pr_subtest{q}{q′}  ⊑  if_pr_subtest{C  O{q}}{C  O{q′}}
).

!assume (∀{p}, {q], {q′}.
    and_semi (if_pr_passes{p}{test_skj})
             (if_pr_less{q}{q′})            ⊑  if_pr_less{O{p}{q}}{O{p}{q′}}
).

!assume (∀{p}, {q}, {x}. (
    and_semi (if_pr_tested{{p}}{test_skj}).
    and_semi (if_pr_passes{x}{p}).
             (if_pr_equal{q  x}{T}))            ⊑  if_pr_equal{O{p}{q}}{T}
).

!assume (∀{p}, {q}, {x}. (
    and_semi (if_pr_tested{{p}}{test_skj}).
    and_semi (if_pr_passes{x}{p}).
             (if_pr_equal{q  x}{⊥}))            ⊑  if_pr_of_type{O{p}{q}}{div}
).
```

Index

Bibliography

[ABNK$^+$87] S.-I. Amari, O. E. Barndorff-Nielsen, R. E. Kass, S. L. Lauritzen, and C. R. Rao, *Differential geometry in statistical inference*, Lecture Notes - Monograph Series, vol. 10, Institute of Mathematical Statistics, 1987.

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken, *The semantics of reflected proof*, In Proc. of Fifth Symp. on Logic in Comp. Sci, IEEE Computer Society Press, 1990, pp. 95–197.

[Ama98] S.-I. Amari, *Natural gradient works efficiently in learning*, Neural Computation **10** (1998), 251–276.

[AN93] S.-I. Amari and Hiroshi Nagaoka, *Methods of information geometry*, Translations of Mathematical Monographs, vol. 191, American Mathematical Society, 1993.

[Bar84] Hendrik Pieter Barendregt, *The lambda calculus – its syntax and semantics*, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland Publishing Company, Amsterdam, 1984.

[Bar93] Henk Barendregt, *Constructive proofs of the range property in lambda calculus*, Theor. Comput. Sci. **121** (1993), no. 1-2, 59–69.

[Bar08] ———, *Towards the range property for the lambda theory h*, Theor. Comput. Sci. **398** (2008), no. 1-3, 12–15.

[BBD93] Henk Barendregt, Martin Bunder, and Will Dekkers, *Systems of illative combaintory logic complete for first-order propositional and predicate calculus*, Journal of Symbolic Logic **58** (1993), no. 3, 769–788.

[BBJ07] George S. Boolos, John P. Burgess, and Richard C. Jeffrey, *Computability and logic*, 5 ed., Cambridge University Press, September 2007.

[BG92] Stephen Brookes and Shai Geva, *Computational comonads and intensional semantics*, Applications of Categories in Computer Science: Proceedings LMS Symp., Durham, UK, 20–30 July 1991 (M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds.), vol. 177, Cambridge University Press, Cambridge, 1992, pp. 1–44.

[BLW03] M. R. Bush, M. Leeming, and R. F. C. Walters, *Computing left Kan extensions*, Journal of Symbolic Computation **35** (2003), 107–126.

[BN99] F. Baader and T. Nipkow, *Term rewriting and all that*, Cambridge University Press, 1999.

[BS01] F. Baader and W. Snyder, *Unification theory*, Handbook of Automated Reasoning (A. Robinson and A. Voronkov, eds.), vol. I, Elsevier Science, 2001, pp. 445–532.

[Car86] Luca Cardelli, *A polymorphic lambda calculus with Type:Type*, Tech. Report 10, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1986.

[CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson, *Implementation and analysis of the Todd-Coxeter algorithm*, Mathematics of Computation **27** (1973), no. 123, 463–490.

[CDJK99] H. Comon, M. Dincbas, J.-P. Jouannaud, and C. Kirchner, *A methodological view of constraint solving*, Constraints **4** (1999), no. 4, 337–361.

[CG95] Siddhartha Chib and Edward Greenberg, *Understanding the metropolis-hastings algorithm*, American Statistician **49** (1995), no. 4, 327–335.

[Cha66] Gregory J. Chaitin, *On the length of programs for computing finite binary sequences*, Journal of the ACM **13** (1966), 547–569.

[CHS72] H.B. Curry, J.R. Hindley, and J.P. Seldin, *Combinatory logic*, vol. II, North-Holland Publishing Company, Amsterdam, 1972.

[CM99] A. Cano and S. Moral, *A review of propagation algorithms for imprecise probabilities*, Proc. of 1st International Symposium on Imprecise Probabilities and their Applications, 1999.

[Con94] Robert L. Constable, *Using reflection to explain and enhance type theory*, Proof and Computation, volume 139 of NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, NATO Series F, Springer, 1994, pp. 65–100.

[CS88] Robert L. Constable and Scott Fraser Smith, *Computational foundations of basic recursive function theory*, Theoretical Computer Science, 1988, pp. 360–371.

[CW91] S. Carmody and R. F. C. Walters, *Computing quotients of actions on a free category*, Category Theory, Proceedings of the International Conference Held in Como, Italy (A. Carboni, M. C. Pedicchio, and G. Rosolini, eds.), Springer, July 1991.

[DCHA00] M. Dezani-Ciancaglini, F. Honsell, and F. Alessi, *A complete characterization of complete intersection-type theories*, ACM TOCL **4** (2000), 224–236.

[DCL02] M. Dezani-Ciancaglini and S. Lusin, *Intersection types and lambda theories*, Tech. Report cs.LO/0211011, The Computing Research Repository, November 2002.

[DGJP04] J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden, *Metrics for labelled markov processes*, Theor. Comput. Sci. **318** (2004), no. 3, 323–354.

[DM95] F.-N. Demers and J. Malenfant, *Reflection in logic, functional and object-oriented programming: a short comparative study*, Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI, 1995, pp. 29–38.

[DP96] Rowan Davies and Frank Pfenning, *A modal analysis of staged computation*, Journal of the ACM, ACM Press, 1996, pp. 258–270.

[DVH98] Anhai Doan, Van Vu, and Peter Haddawy, *Geometric foundations for interval-based probabilities*, Annals of Mathematics and Artificial Intelligence **24** (1998), no. 24, 582–593.

[Eda95] Abbas Edalat, *Domain theory in stochastic processes*, lics **00** (1995), 244.

[Fef92] Solomon Feferman, *Why a little bit goes a long way: Logical foundations of scientifically applicable mathematics*, PSA **II** (1992), 442–455.

[Fef05] ______, *Predicativity*, The Oxford Handbook of Philosophy of Mathematics and Logic (S. Shapiro, ed.), Oxford University Press, 2005, pp. 590–624.

[Geh95] Wolfgang Gehrke, *Problems in rewriting applied to categorical concepts by the example of a computational comonad*, Proceedings of the Sixth International Conference on Rewriting Techniques and Applications (Kaiserslautern, Germany) (Jieh Hsiang, ed.), Springer-Verlag LNCS 914, 1995, pp. 210–224.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont, *Proofs and types*, Cambridge University Press, New York, NY, USA, 1989.

[Har95] John Harrison, *Metatheory and reflection in theorem proving: A survey and critique*, Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[Hec94] Reinhold Heckmann, *Probabilistic power domains, information systems, and locales*, Lecture notes in computer science, Springer, 1994.

[Hin67] Roger Hindley, *Axioms for strong reduction in combinatory logic*, Journal of Symbolic Logic **32** (1967), no. 2, 224–236.

[Hin97] ______, *Basic simple type theory*, Cambridge Tracts in theoretical Computer Science, Cambridge University Press, 1997.

[Hut05] Markus Hutter, *Universal artificial intelligence*, Texts in Theoretical Computer Science, Springer-Verlag Berlin Heidelberg, 2005.

[JJ98] Giorgi Japaridze and Dick De Jongh, *The logic of provability*, Handbook of proof theory (Samuel R. Buss, ed.), Elsevier, Amsterdam, 1998.

[JLM92] F. Jelinek, J. Lafferty, and R. Mercer, *Basic methods of probabilistic context-free grammars*, Speech Recognition and Understanding: Recent Advances, Trends, and Applications (P. Laface and R. De Mori, eds.), Series F: Computer and Systems Sciences, vol. 75, Springer Verlag, 1992.

[Jon89] Claire Jones, *Probabilistic non-determinism*, Ph.D. thesis, University of Edinburgh, Edinburgh, Scotland, UK, 1989.

[JP89] C. Jones and G. Plotkin, *A probabilistic powerdomain of evaluations*, Proceedings of the Fourth Annual Symposium on Logic in computer science (Piscataway, NJ, USA), IEEE Press, 1989, pp. 186–195.

[JS96] M. Jerrum and A. Sinclair, *The Markov chain Monte Carlo method: an approach to approximate counting and integration*, Approximation Algorithms for NP-Hard Problems (D. S. Hochbaum, ed.), PWS Publishing Company, 1996.

[JT98] A. Jung and R. Tix, *The troublesome probabilistic powerdomain*, Third Workshop on Computation and Approximation, Electronic Notes in Theoretical Computer Science, vol. 13, Elsevier Science Publishers, 1998.

[KAF08] Steven Kieffer, Jeremy Avigad, and Harvey Friedman, *A language for mathematical language management*, CoRR **abs/0805.1386** (2008).

[KB70] D. E. Knuth and P. B. Bendix, *Simple word problems in universal algebra*, Proc. Conf. Computational Problems in Abstract Algebra '67, Pergmon Press, 1970, pp. 263–297.

[Kle55] Stephen C. Kleene, *Hierarchies of number-theoretic predicates*, Bulletin of the A.M.S. (1955), no. 61, 193–213.

[KMP97] D. Koller, D. McAllester, and A. Pfeffer, *Effective Bayesian inference for stochastic programs*, Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), 1997, pp. 740–747.

[Kol65] Andrey N. Kolmogorov, *Three approaches to the quantitative definition of information.*, Problems in Information Transmission **1** (1965), 1–7.

[Koz81] Dexter Kozen, *Semantics of probabilistic programs*, Journal of Computer System Sciences **22** (1981), 328–350.

[KS95] Kevin Kelly and Oliver Schulte, *The computable testability of theories with uncomputable predictions*, Erkenntnis **43** (1995), 29–66.

[Lak76] Imre Lakatos, *Proofs and refutations: The logic of mathematical discovery*, Cambridge University Press, Cambridge, 1976, Edited by John Worrall and Elie Zahar.

[LV97] Ming Li and Paul Vitanyi, *An introduction to kolmogorov complexity and its applications*, Texts in Computer Science), Springer, February 1997.

[McA99] David A. McAllester, *On the complexity analysis of static analyses*, Static Analysis Symposium, 1999, pp. 312–329.

[Mog91] Eugenio Moggi, *Notions of computation and monads*, Information and Computation **93** (1991), no. 1, 55–92.

[Odi92] P. Odifreddi, *Classical recursion theory: The theory of functions and sets of natural numbers*, new ed ed., North Holland, February 1992.

[Pea88] Judea Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[Plo82] Gordon Plotkin, *Probabilistic powerdomains*, Proceedings CAAP, 1982.

[PPT05] Sunwoo Park, Frank Pfenning, and Sebastian Thrun, *A probabilistic language based upon sampling functions*, POPL'05 (2005).

[Rog67] Hartley Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967.

[Sco76] Dana Scott, *Data types as lattices*, SIAM Journal of Computing **5** (1976), no. 3, 522–587.

[Sha97] Stewart Shapiro, *Philosophy of mathematics: structure and ontology*, Oxford university Press, 1997.

[Slo] N. J. A. Sloan, *The on-line encyclopedia of integer sequences*, urlhttp://www.research.att.com/˜njas/sequences/index.html.

[Smo77] Craig Smorynski, *The incompleteness theorems*, Handbook of mathematical logic (Jon Barwise, ed.), Amsterdam, North-Holland, 1977, pp. 821–865.

[Sol64] Ray Solomonoff, *A formal theory of inductive inference: Parts 1 and 2*, Information and Control **7** (1964), 1–22 and 224–254.

[Sol78] ______, *Complexity-based induction systems: Comparisons and convergence theorems*, IEEE Transactions on Information Theory **IT-24** (1978), 422–432.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan, *Self-adjusting binary search trees*, J. ACM **32** (1985), no. 3, 652–686.

[TC36] J. A. Todd and H. S. M. Coxeter, *A practical method for enumerating cosets of a finite abstract group*, Proc. of the Edinburgh Math. Soc. **5** (1936), 26–34.

[Tro02] John Tromp, *Kolmogorov complexity in combinatory logic*, see Tromp's web page, March 2002.

[Val84] L. G. Valiant, *A theory of the learnable*, Commun. ACM **27** (1984), no. 11, 1134–1142.

[Ver08] Rineke (L.C.) Verbrugge, *Provability logic*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), Stanford University, Fall 2008.

[Wad90] Philip Wadler, *Comprehending monads*, Mathematical Structures in Computer Science, 1990, pp. 61–78.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies, *Modal types as staging specifications for run-time code generation*, ACM Computing Surveys **30** (1998).

[Yug99] V. V. V' Yugin, *Algorithmic complexity and stochastic properties of finite binary sequences*, The Computer Journal **42** (1999), no. 4, 294–317.