A Constructive Logic of Multiple Subtyping

*Harry G. Mairson**
Cambridge Research Laboratory
Digital Equipment Corporation
Cambridge, Massachusetts 02139

Abstract

We show how a higher order logic, the calculus of constructions, can be used to give a simple, first principles treatment of record calculi, polymorphism, and subtyping. The development follows the constructive idiom of extracting implementations of equationally specified programs from proofs of their termination, with a logic for reasoning about programs, and a semantics that comes as a gratuity. In this framework, records are finitely specified functions where equality is decidable over the domain, with types that are a particular kind of logical assertion. By proving that a record specification satisfies its type, we can extract its implementation. While program extraction serves as a sort of compiler, proof normalization serves as an interpreter; the latter serves to ensure in some sense the coherence of the translation embedded in the former.

This simple minded approach lets us show, for example, that many inference rules found in record and object calculi can be derived — they are just provable lemmas in higher order logic. We see explicitly how from subtyping proofs we can extract, using conventional methods, coercion functions between underlying representations of data types. By further exploiting the computational metaphor of higher order logic, we can realize an interpreter for recursively defined objects, as well as subtype and inheritance relations between them. Recursive types for objects are explained by primitive recursion in higher types. The approach also gives a computational understanding of F-bounded polymorphism.

*On leave from Brandeis University, Waltham, Massachusetts. Supported in part by NSF Grant CCR-9017125, and grants from Texas Instruments and from the Tyson Foundation.

1 Introduction

Type theory has played an important role in the design and analysis of programming languages, since types can be used to prevent certain run-time errors resulting from type mismatch, to formally reason about the behavior of programs, and to specify module interfaces. The latter point, in particular the checking of module interfaces through type compatibility, has been taken up by the proponents of object-oriented programming, where module interfaces describe the functionality of *methods* invoked by differing vocabularies of *messages*, and the notions of subtyping and inheritance formally indicate where one kind of module can be substituted for another.

Records provide a simple and well defined functional programming metaphor for object oriented programming, since records associate *labels* (object oriented *messages*) with *values* (object oriented *methods*) [Car88]. Adding and removing fields from a record then corresponds to adding and removing messages and methods from an object; these values can of course be procedures in a functional programming setting. Adding and removing fields from a record type similarly corresponds to mutation of object oriented class declarations.

Many proposals have been made for typed record calculi. Some of the original proposals stayed close to the ML type system, where both type checking and type inference are practically feasible, and type principality is achievable. More sophisticated type systems and logics have led some researchers away from such feasibility in return for powerful mathematical elegance. The calculi are typically presented as a set of inference rules, with proofs that the inference system satisfies typical properties such as Church-Rosser, strong normalization, etc., with added constructions of semantic models of the calculi. Research in this area has also dealt with how to effect translations of programs from one such calculus to another, and to characterize the so-called *coherence* of these translations.

This general program of research raises an important question: to what extent are these inference systems mere logic? By "logic" we do not mean "well defined" or "rational" in the colloquial sense, but *real* logic, i.e., predicate calculus and its higher order generalizations. Are these type systems for records an elegant syntactic sugar for proving assertions about programs? We want to show that many (but to be fair, not all) of the desirable features of typed programming languages for records and objects with subtyping and inheritance can be achieved using simple, well known principles of higher-order logic.

We provide supporting evidence for this belief, for example, by providing an outline of a soundness proof for the Cardelli-Mitchell record calculus. We do this by giving a higher-order logical specification of what records are, as well as showing how to code types as record specifications. Record types are represented explicitly as dependent products in a logical sense. This formalization is carried out in the *calculus of constructions* [CH88], a computational framework for formalizing intuitionistic arguments. As a consequence, we show that the Cardelli-Mitchell inference rules follow from principles of "pure logic." Questions about confluence, normalization, and soundness can then be deferred to their general treatment in the calculus of constructions.

Also discussed is the issue of *realizability* in the context of logic. In recent work, for example, Cardelli has proposed a justification of record calculi via a translation into a variant of the Girard/Reynolds type system F_2, augmented with subtyping rules [Car91]; Breazu-Tannen, Coquand, Gunter, and Scedrov have also described how programs in a type discipline with subsumption can be translated into one with explicit coercion instead [BCGS91, BGS90]. In a similar spirit, Church invented an informal "justification" of arithmetic via a translation into lambda calculus, i.e., Church numerals. It was shown by Girard, and later by others in variants of his basic insight, that Church's coding could be explained by "purely logical" reasons: namely, by extracting programs (i.e., lambda terms) from proofs of termination of *recursive* programs, where these programs were defined over inductively specified data (i.e., integers). Can a translation of a record calculi into lambda terms be carried out "automatically" (that is, logically) using similar methods? We show that this is indeed the case, though the programs extracted are hardly extraordinary. We discuss how the basic idea of *coherence* of translations gets reinterpreted in this context, as well as the computational necessity of coercion functions.

It is not difficult to extend the presented framework for records and record types to (recursively defined) objects. In the propositions-as-types analogy, recursive types translate naively into infinitely long logical formulas. It seems massive overkill to employ infinitary logic to describe points, stacks, and numbers; also, strong normalization of proofs in higher order logic appears untenable in the context of infinitely long formulas. Instead, we introduce primitive recursion at the level of higher order types to iterate object type constructors. This construction allows us to simulate all the usual examples of objects, and to show the soundness of a standard rule for proving a subtyping relation between recursively defined types. We argue that inheritance is not subtyping [CHC90] because inheritance is not logical. We also show how our general approach provides a computational interpretation of F-bounded polymorphism [CCHOM89].

In this watered-down *Principia Mathematica* of typed objects and records, we realize that many people are searching for an aestheic way to talk about these computational phenomena; as a consequence, differing perceptions of beauty provoke differing opinions as to the appropriateness of various calculi. This presentation was motivated by the simple idea that types are propositions about programs, so that an analogy develops between type checking and proof checking on one hand, and type inference and theorem proving on the other hand. How far can this simple analogy be pushed to solve new problems with old technology? Given that powerful calculi have been proposed for developing typed objects and records, where the computational feasibility of ML-inspired calculi (e.g., [Rém89, Wan89]) has been traded for abstract mathematical power (e.g., [CCHOM89, CHC90, CM91]) —type inference for systems based on Girard's F_2 is a well known open problem, and in fact, type checking for a variant of F_2 with subtyping has been proven undecidable [Pie91]—why not see what happens when the inspiration of the Curry-Howard correspondence is followed up with a very powerful logic?

A final warning: terms in the calculus of constructions are not easy to read. Describing record and object calculi in CC amounts to writing an interpreter in the logical equivalent of microcode. Even high level constructs as simple as records can look complicated when examined at such a microscopic level.

1.1 Guidelines

Here are the goals of the proposed interpretation of records and objects. First, *simplicity:* the theory should not be complicated, because neither records nor objects are computationally complex. The inference rules should be kept to a bare minimum, of sufficient generality to derive others. Next, *fidelity:* the theory should address practical issues and situations concerning reasoning about programs — this exercise is not merely an excuse to do mathematics; rather, to use standard logical methods giving a working understanding of programming language principles. This brings us to the goal of *mundaneness:* the theory should be constructed out of familiar constructs: higher-order logic (merely higher-order programming well known to functional programmers, where the "data values" are logical formulas) and realizability via "contraction" of proofs to programs. Finally, the theory should be *syntax directed:* it should be constructive, oriented completely towards syntactic manipulation and away from semantic reasoning. Model theory and denotational semantics should be avoided.

These general goals have been further detailed in the following design decisions. Programs are equational theories, and types are propositions about programs. Records are a special kind of function, defined over a finite subset of a fixed set of *labels*, with types that are dependent products. When we prove a theorem corresponding to typing a program, we should be able to computationally *realize* the proof by extracting from it a λ-term which embodies the property asserted by the type. The logic used to prove the typing assertion should have the strong normalization property, so that normalization of proofs can be used as an interpreter, and extraction of λ-terms can be used as a compiler.

1.2 Calculus of constructions

To carry out our presentation, we use the *calculus of constructions* (CC), a higher-order logic with dependent products. The higher-order aspect will be essential to building complex types: for instance, taking our coding of a record type and augmenting it with a new field of some type. The dependent product aspect will be an essential substitution mechanism, since rather than programming directly in the calculus, we will use it to prove theorems. In more detail, we will "program" in an equational theory, and the "types" of CC terms will be the logical assertions we want to make about our programs. The CC terms will then be "proof objects" that show how to construct proofs of our assertions. This may all sound complicated, but it isn't: *the calculus of constructions is just a functional notation for manipulating proofs.*

A number of summaries of the calculus of constructions exist [Hue90, CH88]; we provide here the bare minimum to understand what follows. Abstraction is denoted by square brackets, and type quantification denoted by curly brackets. For instance, the identity function for type τ of *kind* Prop (for *logical proposition*) can be written $[x:\tau]x$, which has type $\{x:\tau\}\tau$.[1] Application of a term (read, "proof") $[x:\tau]M$ of type (read, "proposition") $\{x:\tau\}P(x)$ (read, "for all x, $P(x)$") to a term N of type τ yields a term $([x:\tau]M)N$ of type $P(N) \equiv [N/x]P$.

This substitution of the "argument" N into the *type* of the function is particularly disconcerting to those who have never seen dependent products. The rationale can be explained very well by a canonical example: proving that the encoding of equality in second order logic satisfies associativity. Suppose we define an equivalence relation Q over first order terms by the second order logical formula $Q(x,y) \equiv \forall^2 P.P(x) \rightarrow P(y)$ (the so-called *Leibniz equality*), where quantification[2] is over all predicates P. We know $Q(x,y) \rightarrow x = y$ (i.e., Q really does code equality) by instantiating P as $P(z) \equiv z = x$; substituting this definition for P in $Q(x,y)$, we derive $x = x \rightarrow x = y$; the result follows by the axiom $x = x$ and modus ponens.

We would hope that if $Q(x,y)$ is really an equivalence relation, then $\forall^1 xy.Q(x,y) \rightarrow Q(y,x)$ should be provable. In second order logic, take $Q(x,y)$ and instantiate P as $P(z) \equiv Q(z,x)$, deriving $Q(x,x) \rightarrow Q(y,x)$. The premise $Q(x,x) \equiv \forall^2 P.P(x) \rightarrow P(x)$, asserting that an equivalence relation is reflexive, can be easily proved, so by modus ponens, $Q(y,x)$.

Now let's code up this proof in CC. The predicate Q we have defined is written in CC as $Q = [x:\mathsf{Term}][y:\mathsf{Term}]\{P:\mathsf{Term}\rightarrow\mathsf{Prop}\}(Px)\rightarrow(Py)$. In CC, a unary first order predicate (like P) is coded as a term of type $\mathsf{Term}\rightarrow\mathsf{Prop}$, since it is a function mapping input (first order terms) to output (logical proposition with some truth value). The binary predicate Q has type $\mathsf{Term}\rightarrow\mathsf{Term}\rightarrow\mathsf{Prop}$, reflecting its two curried inputs.

If *Prf* is a term (read, "proof") of type (read, "theorem") $Qxy = \{P:\mathsf{Term}\rightarrow\mathsf{Prop}\}(Px)\rightarrow(Py)$, then *Prf* can be *applied* to *any* term P of type $\mathsf{Term}\rightarrow\mathsf{Prop}$, where *Prf* P then has type $(Px)\rightarrow(Py)$. Suppose $R = [z:\mathsf{Term}]Qzx$; then *Prf* R has type

[1] In fact, the notation $\alpha \rightarrow \beta$ is a familiar variant of $\{x:\alpha\}\beta$, where x does not appear in β.

[2] The notation $\forall^i$ means ith-order quantification.

$(Qxx) \rightarrow (Qyx)$. Similarly, *Prf R* can be applied to a term I of type Qxx, to derive *Prf R I* of type Qyx; since $Qxx = \{P : \mathsf{Term} \rightarrow \mathsf{Prop}\}(Px) \rightarrow (Px)$, we let $I = [P : \mathsf{Term} \rightarrow \mathsf{Prop}][w : Px]w$.

Notice that from the *assumption* $\mathit{Prf} : Qxy$, we constructed a *proof* $\mathit{Prf}([z : \mathsf{Term}]Qzx)\,([P : \mathsf{Term} \rightarrow \mathsf{Prop}][w : Px]w)$ having type Qyx. The proof uses a purely functional notation to describe *all* the steps in the second order logic proof. Second order logic may seem easier to read than CC *types*, but lacks an equally formal notation for representing *proofs*. In its entirety, we write the proof as

$$\begin{array}{l} [\mathsf{Term} : \mathsf{Type}] \vdash \\ \quad [x : \mathsf{Term}][y : \mathsf{Term}][\mathit{Prf} : Qxy] \\ \qquad \mathit{Prf}\ ([z : \mathsf{Term}]\ Qzx) \\ \qquad\quad ([P : \mathsf{Term} \rightarrow \mathsf{Prop}][w : Px]w) \\ \quad : \{x : \mathsf{Term}\}\{y : \mathsf{Term}\}(Qxy) \rightarrow (Qyx) \end{array}$$

where Type is the mother of all types; the context declares that Term is a type of type Type.

1.3 Terms, programs, types

We assume programs are first order equational theories, and types are assertions about programs. For instance, here is a typical equational theory:[3]

$$\begin{array}{rcl} \forall^1 x.\mathbf{id}\ x & = & x \\ \forall^1 f.\mathbf{map}\ f\ \mathbf{nil} & = & \mathbf{nil} \\ \forall^1 fyt.\mathbf{map}\ f\ (\mathbf{cons}\ y\ t) & = & \mathbf{cons}\ (f\ y)\ (\mathbf{map}\ f\ t) \end{array}$$

The programming language assertion that $\mathbf{id}$ has type $\forall P.P \rightarrow P$ of $\mathbf{id}$ is interpreted as the (second order) logical assertion $\forall^2 P.\forall^1 x.P(x) \rightarrow P(\mathbf{id}\ x)$. Given the equation defining $\mathbf{id}$ above, the assertion is easy to prove in second order logic.

How is the proof coded in CC? Write the first axiom above as $[\mathit{Id} : \{x : \mathsf{Term}\}Q(\mathit{apply}\,\mathbf{id}\,x)\ x]$, where $\mathbf{id} : \mathsf{Term}$, and $\mathit{apply} : \mathsf{Term} \rightarrow \mathsf{Term} \rightarrow \mathsf{Term}$ builds terms from smaller ones. The programming language "typing" $\mathbf{id} : \forall P.P \rightarrow P$ is then the CC type $\{P : \mathsf{Term} \rightarrow \mathsf{Prop}\}\{x : \mathsf{Term}\}(Px) \rightarrow (P(\mathit{apply}\ \mathbf{id}\,x))$. This type has proof $\Pi = [P : \mathsf{Term} \rightarrow \mathsf{Prop}][x : \mathsf{Term}][\mathit{Prf} : Px]\ \mathit{Id}\ x\,P\,\mathit{Prf}$. By judiciously "crossing out" some subterms in Π, we "realize" $\mathbf{id}$ as $[P : \mathsf{Prop}][\mathit{Prf} : P]\mathit{Prf}$, the usual coding of the polymorphic identity function as a λ-term (with brackets denoting abstraction).

[3] Boldface denotes constants, italic face denotes variables.

Not only does CC provide a framework for such realization of programs (λ-terms) from equational specifications, it provides "for free" a model theory (a semantics), and a logic for reasoning about these specifications, and indirectly, the extracted programs. Here is a trivial example of the logic: Suppose we had some $f : \mathsf{Term}$ and a proof Π' of $\{P : \mathsf{Term} \rightarrow \mathsf{Prop}\}\{x : \mathsf{Term}\}(Px) \rightarrow (P(\mathit{apply}\ f\,x))$. — f has programming language type $\forall P.P \rightarrow P$. We can then *prove* in CC $\{x : \mathsf{Term}\}Q\ (\mathit{apply}\,f\,x)\,(\mathit{apply}\,\mathbf{id}\,x)$ — namely, f *must be* the identity function, though $Q\ f\ \mathbf{id}$ need not be provable.

If this methodology works for programs as silly as the identity function, it ought to work for records, objects, subtyping, etc. In general, we read the programming language notation $f : A \rightarrow B$ in logic as $\forall x.A(x) \rightarrow B(fx)$, or in CC as $\{x : \mathsf{Term}\}(Ax) \rightarrow (B\langle fx\rangle)$.[4] Similarly, the programming language notation $A <: B$ for subtyping is read logically as $\forall x.A(x) \rightarrow B(x)$, in CC $\{x : \mathsf{Term}\}(Ax) \rightarrow (Bx)$. By analyzing typing proofs, we extract code; by analyzing subtyping proofs, we realize coercion functions between the underlying realizations of the datatypes (see section 1.5).

1.4 Inductive types

The logical framework lets us code up all the usual inductive types; for instance the Peano axiom $\mathit{Int}(n) \equiv \forall^2 P.(\forall^1 x.P(x) \rightarrow P(\mathbf{succ}\ x)) \rightarrow P(\mathbf{0}) \rightarrow P(n)$ gets coded by putting constants $\mathbf{succ}$ and $\mathbf{0}$ (of type Term) in the context, defining

$$\begin{array}{l} \mathit{Int} = [n : \mathsf{Term}]\{P : \mathsf{Term} \rightarrow \mathsf{Prop}\} \\ \qquad (\{x : \mathsf{Term}\}(Px) \rightarrow (P\langle \mathbf{succ}\ x\rangle)) \\ \qquad\quad \rightarrow (P\ \mathbf{0}) \rightarrow (P\ n) \end{array}$$

where Int is a term of type $\mathsf{Term} \rightarrow \mathsf{Prop}$ (a predicate on terms, abbreviated as Pred), thus $\mathit{Int}\ \langle \mathbf{succ}(\mathbf{succ}\ \mathbf{0})\rangle$ has proof

$$\begin{array}{l} \nu_2 = [P : \mathsf{Pred}][s : \{x : \mathsf{Term}\}(Px) \rightarrow (P\langle \mathbf{succ}\ x\rangle)] \\ \qquad [z : P\ \mathbf{0}]s\ \langle \mathbf{succ}\ \mathbf{0}\rangle\ (s\ \mathbf{0}\ z) \end{array}$$

By removing all the "first-order" information from the proof term, we get $[P : \mathsf{Prop}][s : P \rightarrow P][z : P]\ s\ (s\ z)$, the familiar Church numeral. This *contraction* procedure is the essence of extracting programs from proofs [BB85, Gir72, GLT89, Lei90].

[4] The bracket notation $\langle e\rangle$ abbreviates iterated use of $\mathit{apply} : \mathsf{Term} \rightarrow \mathsf{Term} \rightarrow \mathsf{Term}$ to build term e.

1.5 Subtyping and coercion

We can now define subtyping, and an example "type" *Even* that is meant to be a subtype of *Int*:

$$\begin{aligned}
&\mathit{Subtype} = \\
&\quad [A : \mathsf{Term} \to \mathsf{Prop}][B : \mathsf{Term} \to \mathsf{Prop}] \\
&\qquad \{x : \mathsf{Term}\}(A\ x) \to (B\ x) \\
&\mathit{Even} = \\
&\quad [n : \mathsf{Term}]\{P : \mathsf{Term} \to \mathsf{Prop}\} \\
&\qquad (\{x : \mathsf{Term}\}(Px) \to (P\langle \mathbf{succ}(\mathbf{succ}\ x)\rangle)) \\
&\qquad\quad \to (P\ \mathbf{0}) \to (P\ n)
\end{aligned}$$

Thus *Subtype* A B is the logical assertion $\forall^1 x.A(x) \to B(x)$. We can show that $\mathit{Even}\langle \mathbf{succ}(\mathbf{succ}\ \mathbf{0})\rangle$ is provable, by the construction

$$\begin{aligned}
\epsilon_2 = [P : \mathsf{Pred}][s : \{x : \mathsf{Term}\}(Px) \\
\to (P\langle \mathbf{succ}(\mathbf{succ}\ x)\rangle)] \\
[z : P\ \mathbf{0}]\ s\ \mathbf{0}\ z.
\end{aligned}$$

(Notice this term contracts to $\underline{\epsilon_2} = [P : \mathsf{Prop}][s : P \to P][z : P]s\ z$, somewhat oddly, the Church numeral for *1*.) We prove *Subtype Even Int* by the construction

$$\begin{aligned}
&[n : \mathsf{Term}][E : \mathit{Even}\ n]E\ \mathit{Int} \\
&\quad ([x : \mathsf{Term}][I : \mathit{Int}\ x] \\
&\qquad \mathit{Successor}\ \langle \mathbf{succ}\ x\rangle\ (\mathit{Successor}\ x\ I))\ \nu_0
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{Successor} = \\
&[n : \mathsf{Term}][I : \mathit{Int}\ n][P : \mathsf{Pred}] \\
&\quad [s : \{x : \mathsf{Term}\}(Px) \to (P\langle \mathbf{succ}\ x\rangle)][z : P\ \mathbf{0}] \\
&I\ ([x : \mathsf{Term}](P\ \langle \mathbf{succ}\ x\rangle)) \\
&\quad ([x : \mathsf{Term}][\mathit{Prf} : P\ \langle \mathbf{succ}\ x\rangle] \\
&\qquad s\ \langle \mathbf{succ}\ x\rangle\ \mathit{Prf})\,(s\ \mathbf{0}\ z) \\
&\quad\ :\ \{n : \mathsf{Term}\}(\mathit{Int}\ n) \to (\mathit{Int}\ \langle \mathbf{succ}\ n\rangle) \\
&\nu_0 = \\
&[P : \mathsf{Pred}][s : \{x : \mathsf{Term}\}(Px) \to (P\langle \mathbf{succ}\ x\rangle)] \\
&\quad [z : P\ \mathbf{0}]\ z \\
&\qquad : \ \mathit{Int}\ \mathbf{0}
\end{aligned}$$

Given this framework, we see explicitly, using entirely standard and well-known translation methodology, how the computational interpretation of subtyping is related to coercion functions. By removing first-order information from the proofs of *Subtype Even Int* and $\{n : \mathsf{Term}\}(\mathit{Int}\ n) \to (\mathit{Int}\ \langle \mathbf{succ}\ n\rangle)$, we derive the "programs" and "types"

$$\underline{\mathit{Even}} = \underline{\mathit{Int}} = \{P : \mathsf{Prop}\}(P \to P) \to P \to P$$

$$\begin{aligned}
&\underline{\mathit{Successor}} = \\
&\quad [I : \underline{\mathit{Int}}][P : \mathsf{Prop}][s : P \to P][z : P] \\
&\qquad I\ P\ ([\mathit{Prf} : P]\ s\ \mathit{Prf})\ (s\ z) \\
&\underline{\mathit{Subtype\ Even\ Int}} = \\
&\quad [E : \underline{\mathit{Even}}]E\ \underline{\mathit{Int}} \\
&\qquad ([I : \mathit{Int}]\underline{\mathit{Successor}}\ (\underline{\mathit{Successor}}\ I)) \\
&\qquad\quad ([P : \mathsf{Prop}][s : P \to P][z : P]z)
\end{aligned}$$

Observe, then, that $\underline{\mathit{Subtype\ Even\ Int}\ \epsilon_2}$ β-reduces to $\underline{\nu_2}$ — namely, $\underline{\mathit{Subtype\ Even\ Int}}$ is a function that *coerces* the "even" representation of 2 to its "integer" representation. It should be clear that this extraction of coercions from subtyping, done along purely logical lines, will carry through for many kinds of data types.

1.6 Contravariance

The function space constructor is contravariant because logical implication satisfies principles of contravariance. Here is a typical "rule" for subtyping in a programming language (see, e.g., [Car89, Bru92]):

$$\frac{\Gamma \vdash A' <: A \qquad\qquad \Gamma \vdash B <: B'}{\Gamma \vdash A \to B <: A' \to B'}$$

Why does this rule need to be mandated *ex nihilo* when it follows, in the right framework, from principles of logic? It ought to be a derived rule.

In CC, suppose we have terms α and β where

$$\begin{aligned}
\alpha &: \mathit{Subtype}\ A'\ A = \{x : \mathsf{Term}\}(A'x) \to (Ax) \\
\beta &: \mathit{Subtype}\ B\ B' = \{x : \mathsf{Term}\}(Bx) \to (B'x)
\end{aligned}$$

Let $\mathit{Funtype} = [F : \mathsf{Term} \to \mathsf{Prop}][G : \mathsf{Term} \to \mathsf{Prop}][f : \mathsf{Term}]\{x : \mathsf{Term}\}(Fx) \to (G\langle fx\rangle)$; we want to construct a term (proof) for the conclusion of the contravariance rule:

$$\begin{aligned}
\gamma\ &:\ \mathit{Subtype}\ (\mathit{Funtype}\ A\ B)\ (\mathit{Funtype}\ A'\ B') \\
&\equiv\ \{f : \mathsf{Term}\}((\mathit{Funtype}\ A\ B)\ f) \\
&\qquad \to ((\mathit{Funtype}\ A'\ B')\ f)
\end{aligned}$$

Namely,

$$\begin{aligned}
\gamma\ =\ &[f : \mathsf{Term}]\{\mathit{Prf} : (\mathit{Funtype}\ A\ B)\ f\} \\
&\quad \{x : \mathsf{Term}\}\{F : A'f\} \\
&\qquad \beta\ \langle fx\rangle\,(\mathit{Prf}\ x\,(\alpha\ x\ F))
\end{aligned}$$

In the "body" of γ, $\alpha\ x\,F$ has type Ax, so that $\mathit{Prf}\ x\,(\alpha\ x\,F) : B\,\langle fx\rangle$, and $\beta\ \langle fx\rangle\,(\mathit{Prf}\ x\,(\alpha\ x\,F)) : B'\,\langle fx\rangle$.

We can then *replace* a mandated inference rule by a construction *Contravariance* serving exactly the same purpose in type derivations. What is gained by all this? Instead of lots of inference rules, we only need a few, namely for CC term formation and type checking. Any specialized system (for records, objects, or whatever) can be built out of these generic parts. The terms (for instance, *Contravariance*) can then be interpreted as pieces of a soundness proof for the (record, object, etc.) calculus modulo the soundness of CC. Once again, the computational content of contravariant subtyping (again, extracting programs from proofs) is a coercion function for the realization of the underlying datatypes.

1.7 Coherence

Some brief, and not definitive, words about coherence. Recent work discussing the relationship between subtyping and coercion has raised the issue of *coherence*: though two structurally different assertions about the same subtyping relation result in two computationally different coercion functions, the functions should in some well-defined sense *do the same thing*. In our context, this means that the "contractions" of subtyping proofs to coercion functions should be equivalent. For motivation, there are two proofs of $\{x : \mathsf{Term}\}(\mathit{Int}\ x) \rightarrow (\mathit{Int}\ \langle \mathbf{succ}\, x\rangle)$: one contracts to <u>*Successor*</u> above, written as the untyped λ-term $\lambda n.\lambda s.\lambda z.ns(sz)$, and the other contracts to the equivalent of $\lambda n.\lambda s.\lambda z.s(nsz)$. Both of these λ-terms "realize" successor on Church numerals, though they are not $\beta\eta$-equivalent (they are both in normal form). Yet it should be clear that *on Church numerals* they do the same thing.

An informal argument to establish the coherence of our translation exploits the Curry-Howard isomorphism: namely, that *normalizing proofs* in our "source" logic corresponds to β-reduction of our "target" terms. Normalizing a proof does not change its conclusion, this the logical interpretation of the λ-calculus *subject reduction lemma*. Hence the conclusion of a normalized proof, where the conclusion is an inductively defined type (roughly equivalent to the idea of *observable type* in [BCGS91]), could be alternately produced by pure equational reasoning. If using two different proofs of the same logical assertion leads to two different answers, it would mean that our equational axioms were inconsistent.

2 Records

Records are functions. They map a finite domain (chosen from a set of *labels*) to a range of values. A record type is a dependent product. Equality of labels should be decidable. We should be able to reason about field extension and removal. For example, we could define a record $\mathbf{r}$ by axioms $\mathbf{r}\,\mathbf{a} = \mathbf{succ}\,\mathbf{0}$ and $\mathbf{r}\,\mathbf{b} = \mathbf{true}$, where $\mathbf{a}$ and $\mathbf{b}$ are labels. From a logical assertion $\mathcal{R}(\mathbf{r})$ asserting the equivalent of a dependent type, we should be able to prove, for example, $\mathcal{R}(\mathbf{r}) \rightarrow \mathit{Int}(\mathbf{r}\,\mathbf{a})$.

2.1 Intuitionistic logic

We employ the usual intuitionistic cast of characters for logic:

$$
\begin{aligned}
&\textsc{absurd} = \bot = \{P : \mathsf{Prop}\}P \\
&\textsc{true} = \{P : \mathsf{Prop}\}P \rightarrow P \\
&\textsc{not}\ [P : \mathsf{Prop}] = P \rightarrow \bot \\
&\textsc{and}\ [P : \mathsf{Prop}][Q : \mathsf{Prop}] = \\
&\qquad \{R : \mathsf{Prop}\}(P \rightarrow Q \rightarrow R) \rightarrow R \\
&\textsc{or}\ [P : \mathsf{Prop}][Q : \mathsf{Prop}] = \\
&\qquad \{R : \mathsf{Prop}\}(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \\
&\textsc{cond}\ [P : \mathsf{Prop}][Q : \mathsf{Prop}][R : \mathsf{Prop}] = \\
&\qquad \textsc{and}\ (P \rightarrow Q)\ ((\textsc{not}\ P) \rightarrow R) \\
&\textsc{iff}\ [P : \mathsf{Prop}][Q : \mathsf{Prop}] = \\
&\qquad \textsc{and}\ (P \rightarrow Q)\,(Q \rightarrow P)
\end{aligned}
$$

We also use the infix $\leftrightarrow$ for IFF. Do not confuse the intuitionistic logical connectives with operations on type *Bool*, defined $\mathit{Bool} \equiv [b : \mathsf{Term}]\{P : \mathsf{Pred}\}(P\,\mathbf{true}) \rightarrow (P\,\mathbf{false}) \rightarrow (P\,b)$—there is no CC term of type $\bot$, but there is a constant $\mathbf{false} : \mathsf{Term}$.

2.2 Labels

We axiomatize labels by adding to the context

$$
\begin{aligned}
&[\mathit{Label} : \mathsf{Term} \rightarrow \mathsf{Prop}][\mathbf{eq} : \mathsf{Term}] \\
&[\mathit{Eq}_1 : \{x : \mathsf{Term}\}\{y : \mathsf{Term}\} \\
&\qquad (\mathit{Label}\,x) \rightarrow (\mathit{Label}\,y) \rightarrow (\mathit{Bool}\,\langle \mathbf{eq}\ x\ y\rangle)] \\
&[\mathit{Eq}_2 : \{x : \mathsf{Term}\}\{y : \mathsf{Term}\} \\
&\qquad (\mathit{Label}\,x) \rightarrow (\mathit{Label}\,y) \rightarrow \\
&\qquad\qquad ((Q\ \mathbf{true}\,\langle \mathbf{eq}\ x\ y\rangle) \leftrightarrow (Q\ x\ y))] \\
&[\mathit{Eq}_3 : \{x : \mathsf{Term}\}\{y : \mathsf{Term}\} \\
&\qquad (\mathit{Label}\,x) \rightarrow (\mathit{Label}\,y) \rightarrow \\
&\qquad\qquad ((Q\ \mathbf{false}\,\langle \mathbf{eq}\ x\ y\rangle) \leftrightarrow (\textsc{not}\ (Q\ x\ y)))] \\
&[\mathit{Eq}_4 : \textsc{not}\ (Q\ \mathbf{true}\ \mathbf{false})]
\end{aligned}
$$

The first three axioms Eq_i assert that $\mathbf{eq}$ is a function from a pair of labels to a Boolean function that is totally defined, and gives the expected answer. The last axiom asserts that if we can prove Q **true false**, then we can prove anything.

2.3 Records, prototypes, record types

The canonical empty record is the term **null**, and the canonical "empty" record type (asserting nothing about a record) is the term $Null = [rec : \mathsf{Term}]\{\ell : \mathsf{Term}\}\text{TRUE}$. (This is ridiculous, but gets the ball rolling.)

A *prototype* is a *binary* predicate (and a logical hack) on records and labels, of type $\mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}$; for example, $\textit{Null-prototype} = [rec : \mathsf{Term}][\ell : \mathsf{Term}]\text{TRUE}$. Prototypes can be augmented by adding an assertion about a field label and associated data type:

$$\begin{aligned}
&Augment = \\
&[\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}][lab : \mathsf{Term}][T : \mathsf{Pred}] \\
&\quad [rec : \mathsf{Term}][\ell : \mathsf{Term}] \\
&\qquad \text{COND}\ (Q\ \ell\ lab)\ (T\ \langle rec\ \ell\rangle)\,(\Phi\ rec\ \ell)
\end{aligned}$$

Variant records [CW85] can of course be developed by augmenting *Augment*, replacing conjunction with disjunction. *Record types* are constructed from prototypes by

$$\begin{aligned}
&Recordtype = \\
&\quad [\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}] \\
&\qquad [rec : \mathsf{Term}]\{lab : \mathsf{Term}\} \\
&\qquad\quad (Label\ lab) \to (\Phi\ rec\ lab)
\end{aligned}$$

Thus the record $\mathbf{r}$ defined by axioms $\mathbf{r\,a} = \mathbf{succ\,0}$ and $\mathbf{r\,b} = \mathbf{true}$ satisfies the predicate

$$\begin{aligned}
\Pi_1 \quad = \quad & Recordtype(Augment \\
& (Augment\ \textit{Null-prototype}\,\mathbf{a}\ Bool)\,\mathbf{b}\ Int).
\end{aligned}$$

Because prototypes describe functions on a finite domain, they can be given an inductive specification in higher order logic:

$$\begin{aligned}
&Proto = \\
&[\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}] \\
&\quad \{\mathcal{P} : (\mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}) \to \mathsf{Prop}\} \\
&\qquad (\{\Psi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}\} \\
&\qquad \{\ell : \mathsf{Term}\}\{T : \mathsf{Pred}\} \\
&\qquad\quad (Label\ \ell) \to (\mathcal{P}\ \Psi) \to \\
&\qquad\qquad (\mathcal{P}\ (Augment\ \Psi\,\ell\,T))) \\
&\qquad \to (\mathcal{P}\ \textit{Null-prototype}) \to (\mathcal{P}\ \Phi)
\end{aligned}$$

(Translation: to prove a property $\mathcal{P}$ of a prototype Φ, check it holds for the basis *Null-prototype*, and inductively for any augmentation.) This definition provides an induction axiom for formally reasoning about records (gory details omitted). In addition, given an equational theory defining records, when we prove for some $r : \mathsf{Term}$ that $(Recordtype\ \Phi)\ r$ and $Proto\ \Phi$, from the proof of $(Recordtype\ \Phi)\ r$ we can extract a program (i.e., a λ-term) realizing the properties of r asserted by $Recordtype\ \Phi$. Adding and removing fields from a record type is done using *Augment*; to remove a field ℓ from prototype Φ, we use $Augment\ \Phi\,\ell\,[x : \mathsf{Term}]\text{TRUE}$. Because the implementation is not effected with *row variables*, negative information about fields is not necessary. We remark that explicit use of prototypes can be eliminated, at the cost of additional complications in our logical coding.

Rules for subtyping on record types follows from principles of pure logic. For instance, many record calculi have an inference rule saying that record type A is a subtype of record type B if B defines a subset of A's messages, and on that subset, "pointwise" subtyping is preserved. This becomes in CC a simple lemma about conjunction. Program extraction from the proof just implements a coercion function that removes fields. Of course, the functionality of records makes this coercion function unnecessary. Observe that this is *not* the case in our example of *Subtype Even Int* (section 1.5).

2.4 Soundness of record calculi

Having set up a lot of logical machinery, we can use it. For example, we can show that the type inference rules of a variant of Cardelli and Mitchell's record calculus [CM91] are sound, by showing that they follow from principles of "pure logic." This way, we can bypass a lot of unnecessary model theory. We choose two examples from their paper. The first is rule (TE7): it essentially states that if R is a record type lacking a field with label ℓ and type A, extending R as such, and projecting on ℓ, should yield type A. We rewrite this in our constructive logic, ignoring the condition that R lacks an ℓ label (because we don't care):

$$\begin{aligned}
&\{\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}\}\{\mathcal{P} : Proto\ \Phi\} \\
&\quad \{\ell : \mathsf{Term}\}\{L : Label\ \ell\}\{A : \mathsf{Pred}\}\{x : \mathsf{Term}\} \\
&\qquad ((Project\ (Augment\ \Phi\,\ell\,A)\,\ell)\,x) \\
&\qquad\quad \leftrightarrow\ (A\ x)
\end{aligned}$$

where $Project = [\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}][\ell : \mathsf{Term}][y : \mathsf{Term}]\{rec : \mathsf{Term}\}(Q\ y\ \langle rec\ \ell\rangle) \to (\Phi\ rec\ \ell)$.

Our second example is rule (TE9): it states that given a record type R having a field of type A with label ℓ, R should be equivalent to the record type derived by removing the field, and adding it back:

$$\{R : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}\}\{rec : \mathsf{Term}\} \\ \{\mathcal{P} : \mathit{Proto}\ R\}((\mathit{Recordtype}\ R)\ rec) \leftrightarrow \\ ((\mathit{Recordtype} \\ (\mathit{Augment}\ R\ \ell\,(\mathit{Project}\ R\ \ell)))\ rec)$$

The proof of these assertions, via induction on prototypes, is given in the appendix.

3 Objects

Here is a canonical example of object-oriented programming: the one-dimensional point, presented equationally:

$$\begin{aligned} \forall^1 v.(\mathbf{makept}\ v)\ \mathbf{value} &= v \\ \forall^1 vd.(\mathbf{makept}\ v)\ \mathbf{move}\ d &= \mathbf{makept}\ (\mathbf{plus}\ v\ d) \\ \mathbf{two} &= \mathbf{makept} \\ &\qquad (\mathbf{succ}\,(\mathbf{succ}\ \mathbf{0})) \end{aligned}$$

The constant **makept** serves as an object constructor. The "object" **two** can now "receive messages" **value** and **move**: the former returns a value, the latter a "method." Ultimately, we would like to associate a "type" with such objects like $\mathit{Point} = \{\mathbf{value} : \mathit{Int}, \mathbf{move} : \mathit{Int} \to \mathit{Point}\}$.

3.1 Higher order primitive recursive iterators

The standard formal presentation of such *recursive types* has been through fixpoint operators at both the value and type level. In our logical framework, recursive types seem quite untenable: infinitely long logical formulas mean a loss of strong normalization (our interpreter), and types as induction axioms seem equally hopeless—we are led to consider what induction axioms will be *admissible*. Instead, we explain recursive types using a form of higher-order primitive recursion. Recall $\mathsf{Pred} \equiv \mathsf{Term} \to \mathsf{Prop}$; we also define $\mathsf{Pred}_0 \equiv \mathsf{Pred}$, $\mathsf{Pred}_{k+1} \equiv \mathsf{Pred}_k \to \mathsf{Pred}_k$, in order to define higher order Church numerals, to be used as iterators:

$$\mathrm{CN}\ [T : \mathsf{Pred}_2] = \\ \{\Pi : \mathsf{Pred}_2 \to \mathsf{Prop}\} \\ (\{X : \mathsf{Pred}_2\}\ \Pi\, X \to \Pi\,(\mathrm{Succ}\ X)) \\ \to \Pi\,\mathrm{Zero} \to \Pi\, T$$

$$\mathrm{Succ}\ [X : \mathsf{Pred}_2] = \\ [S : \mathsf{Pred} \to \mathsf{Pred}][Z : \mathsf{Pred}]\ S(XSZ) \\ \mathrm{Zero} = [S : \mathsf{Pred} \to \mathsf{Pred}][Z : \mathsf{Pred}]\ Z$$

We can construct a term of type

$$\{T : \mathsf{Pred}_2\}(\mathrm{CN}\ T) \to (\mathrm{CN}\ (\mathrm{Succ}\ T))$$

(i.e., our numerals have successors):

$$\Upsilon = [T : \mathsf{Pred}_2][\mathit{Prf} : \mathrm{CN}\ T][\Pi : \mathsf{Pred}_2 \to \mathsf{Prop}] \\ [\mathcal{S} : \{X : \mathsf{Pred}_2\}\Pi\, X \to \Pi\,(\mathrm{Succ}\ X)] \\ [\mathcal{Z} : \Pi\,\mathrm{Zero}](\mathcal{S}\, T)\ (\mathit{Prf}\ \Pi\, \mathcal{S}\, \mathcal{Z})$$

We use these constructs to define the object type *Point*: we first define

$$\Phi[P : \mathsf{Pred}][N : \mathsf{Pred}] = \\ \mathit{Recordtype} \\ (\mathit{Augment} \\ (\mathit{Augment}\ \mathit{Null\text{-}prototype}\,\mathbf{value}\ \mathit{Int}) \\ \mathbf{move}\,(\mathit{Funtype}\ \mathit{Int}\ P))$$

and then define $\mathit{Point} = \mathit{Object\text{-}type}\ \Phi$, where

$$\mathit{Unroll}\ [N : \mathsf{Pred}_2][\Omega : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}] = \\ N\ ([P : \mathsf{Pred}]\ \Omega\ P\ P)\ (\Omega\ \mathit{True}\,\mathit{False}) \\ \mathit{Object\text{-}type}\ [\Omega : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}] = \\ [\mathit{obj} : \mathsf{Term}]\{T : \mathsf{Pred}_2\}(\mathrm{CN}\ T) \to \\ \mathit{Unroll}\ T\,\Omega\,\mathit{obj}$$

The bindings P and N in Φ mark the *positive* and *negative* occurrences of *Point* in $\{\mathbf{value} : \mathit{Int}, \mathbf{move} : \mathit{Int} \to \mathit{Point}\}$ (here there is only one positive occurrence). The constructor *Object-type* uses a higher order Church numeral to iterate Φ on the "basis" $\Phi\ \mathit{True}\,\mathit{False}$. The "recursive type" of points then gets interpreted as: for *any finite* unwinding of the type declaration, everything is OK.

3.2 Folding and unfolding

Many object oriented type systems have an "unfolding" rule for types. Like our example of contravariance, it would be nice to derive such a rule from logic rather than adding it by fiat. In fact, we can prove the unfolding

$$\{\mathit{obj} : \mathsf{Term}\}(\mathit{Point}\ \mathit{obj}) \to \\ \{d : \mathsf{Term}\}(\mathit{Int}\ d) \to (\mathit{Point}\langle \mathit{obj}\ \mathbf{move}\ d\rangle).$$

The proof depends in a straightforward manner on the higher order Church numerals having successor, since these numerals are used to "unwind" the definitions; we just unwind once, and use conjunction elimination. A curiosity of this framework is that no *generic* construction seems obvious to do unfolding on *all* object types, while it seems clear how to unfold any *particular* such type. This is like a formal system that has proofs of some $\phi(n)$ for each integer n, but no proof of $\forall n.\phi(n)$. Thus a certain amount of reasoning goes on "outside" the system.

3.3 Subtyping recursive types

We can use the higher-order iterators to derive a standard inference rule about subtyping recursive types, written in a vanilla notation as:

$$\frac{\Gamma \cup \{P <: Q\} \vdash \ \Phi(P) <: \Psi(Q)}{\Gamma \vdash \ \mu t.\Phi(t) <: \mu t.\Psi(t)}$$

written in CC as

$$\begin{array}{l}\{\Phi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}\}\\ \quad \{\Psi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}\}\\ \quad\quad (\{P : \mathsf{Pred}\}\{Q : \mathsf{Pred}\}(\mathit{Subtype}\ P\, Q) \to\\ \quad\quad\quad (\mathit{Subtype}\ (\Phi\ P\, P)\,(\Psi\ Q\, Q)))\\ \quad \to \mathit{Subtype}\ (\mathit{Object\text{-}type}\ \Phi)\ (\mathit{Object\text{-}type}\ \Psi)\end{array}$$

The CC term with this type is a straightforward proof by induction on the higher order Church numerals. Details of the proof appear in the appendix.

3.4 Inheritance is not subtyping, etc.

Given $\mathit{Number} = \{\mathbf{value} : \mathit{Int}, \mathbf{lesseq} : \mathit{Number} \to \mathit{Bool}\}$ and $\mathit{Order} = \{\mathbf{lesseq} : \mathit{Order} \to \mathit{Bool}\}$, it has been pointed out that *Number* is not a subtype of *Order*. This has usually been demonstrated by manipulating the standard inference rule for recursive types and arguments about contravariance. When this putative subtyping relationship is viewed logically, another argument emerges to show that, quite simply, inheritance (i.e, of an *Order* object that inherits a **value** field) does not logically imply subtyping. From our logical interpretation of the above types, we may easily prove the following three assertions:

$$\begin{array}{l}\{\mathit{obj} : \mathsf{Term}\}(\mathit{Number}\ \mathit{obj}) \to\\ \quad \{y : \mathsf{Term}\}(\mathit{Number}\ y) \to\\ \quad\quad (\mathit{Bool}\ \langle \mathit{obj}\ \mathbf{lesseq}\, y\rangle)\\ \{\mathit{obj} : \mathsf{Term}\}(\mathit{Number}\ \mathit{obj}) \to\ (\mathit{Int}\ \langle \mathit{obj}\ \mathbf{value}\rangle)\\ \{\mathit{obj} : \mathsf{Term}\}(\mathit{Order}\ \mathit{obj}) \to\\ \quad \{y : \mathsf{Term}\}(\mathit{Order}\ y) \to\\ \quad\quad (\mathit{Bool}\ \langle \mathit{obj}\ \mathbf{lesseq}\, y\rangle)\end{array}$$

Were *Number* indeed a subtype of *Order*, then we would have a proof of $\{\mathit{obj} : \mathsf{Term}\}(\mathit{Number}\ \mathit{obj}) \to (\mathit{Order}\ \mathit{obj})$. In contradiction, then, we will define an object $\mathbf{f}$ which is clearly a *Number*, but not an *Order*: consider the axioms $\mathbf{f}\ \mathbf{value} = \mathbf{0}$, $\forall^1 y.\mathbf{f}\ \mathbf{lesseq}\, y = \mathbf{lesseq}_{\mathit{Int}}\ (\mathbf{f}\ \mathbf{value})\ (y\ \mathbf{value})$, where $\mathbf{lesseq}_{\mathit{Int}}$ defines the property on *Int*. Certainly $\mathbf{f}$ is a *Number*. But it is not an *Order*: define $\mathbf{g}$ by the single axiom $\forall^1 z.\mathbf{g}\ \mathbf{lesseq}\, z = \mathbf{false}$. Again, clearly $\mathbf{g}$ is an *Order*, thus $\mathit{Bool}\langle \mathbf{f}\ \mathbf{lesseq}\, \mathbf{g}\rangle$ must be provable if $\mathbf{f}$ is truly an *Order*; an impossibility, since $\mathbf{f}\ \mathbf{lesseq}\, \mathbf{g}$ cannot be reduced by the equational axioms to either **true** or **false**—the object $\mathbf{g}$ is not equipped with a **value** message.

3.5 F-bounded polymorphism

The idea of F-bounded polymorphism proposed in [CCHOM89] is meant to get around such anomalies, in particular to "type" functions like $\forall^1 xy.\mathbf{min}\ x\, y = \mathbf{if}\ (x\ \mathbf{lesseq}\ y)\ x\, y$. The discussion of the previous subsection shows that, even though we could prove

$$\begin{array}{l}\{T : \mathsf{Pred}\}(\mathit{Subtype}\ T\ \mathit{Order}) \to\\ \quad \{x : \mathsf{Term}\}\{y : \mathsf{Term}\}\\ \quad\quad (T x) \to (T y) \to (T\ \langle \mathbf{min}\ x\, y\rangle),\end{array}$$

the proof would be essentially useless. Instead, define

$$\begin{array}{l}F\ [P : \mathsf{Pred}] =\\ \quad [r : \mathsf{Term}]\{s : \mathsf{Term}\}\\ \quad\quad (P s) \to (\mathit{Bool}\langle r\ \mathbf{lesseq}\ s\rangle)\end{array}$$

(Observe that $\mathit{Order} = \mathit{Object\text{-}type}\ F$.) Clearly, $\mathit{Subtype}\ \mathit{Number}\ (F\ \mathit{Number})$ is provable, thus we "type" **min** as

$$\begin{array}{l}\{T : \mathsf{Pred}\}(\mathit{Subtype}\ T\ (F\ T)) \to\\ \quad \{x : \mathsf{Term}\}\{y : \mathsf{Term}\}(T x)\\ \quad\quad \to (T y) \to (T\ \langle \mathbf{min}\ x\, y\rangle),\end{array}$$

Here is another example: define $\forall^1 x.\mathbf{translate}\ x = x\ \mathbf{move}\ (\mathbf{succ}\ \mathbf{0})$. If we then define

$$\begin{array}{l}F\ [P : \mathsf{Pred}] =\\ \quad [r : \mathsf{Term}]\{s : \mathsf{Term}\}\,(\mathit{Int}\ s) \to (P\langle r\ \mathbf{move}\ s\rangle),\end{array}$$

we can prove

$$\begin{array}{l}\{T : \mathsf{Pred}\}(\mathit{Subtype}\ T\ (F\ T)) \to\\ \quad \{x : \mathsf{Term}\}(T x) \to (T\ \langle \mathbf{translate}\ x\rangle),\end{array}$$

so we know **translate** works on objects of type *Point*, as well as any type T with a properly defined **move** message.

4 Conclusions

Very standard constructs from higher-order logic can be used to explain computation with records and objects. Just as λ-calculus is recognized as an expressive formalism for representing computation, higher order logic (a mixture of lambda calculus and logic) is an expressive formalism for representing types. The existence of strong normalization and realizability for this logic means that the formalism is ideal for calculation as well as representation: for example, we have "off the shelf" a compiler and interpreter for F-bounded polymorphism. The logical foundation provides a mechanism for reasoning about the programs; the semantics of the logic provides a semantics for the programs. The idea of "multiple subtyping" just means that you can prove more than one theorem about a first-order term.

I have tried to answer "what are types for records and objects?" by appealing to logical syntax rather than denotational semantics. But is this paper merely an exercise in denotational semantics with CC as a metalanguage? Whether or not the "logical approach" is the right way remains to be seen, but it is worth thinking about. The perspective given in this paper, borrowing from categorical terminology, is that CC is a useful (and in some ways, simple) "internal language" to discuss the denotational semantics of record and object calculi.

Constructive logic is both a blessing and a curse: a blessing for the reasons we have given, but also a curse because the formal syntax forces us to say what we mean in excruciating detail. In part, is this always the price paid for formal methods? The same problem of detail holds in principle for denotational semantics, but there, less attention to syntax may provide a "way out" via honest but suggestive hand waving. A research challenge is to make the logical approach more palatable by writing better programs than the ones I have, incorporating good data abstraction at the level of proofs.

Other failings of this approach are that there is no clear delineation of what a "type" actually *is* for a term—is any logical construct admissible?—and that reasoning about unbounded computation seems problematical. The latter, of course, is the price we pay for strong normalization. Our introduction of higher-order iterators eliminates the need for unbounded computation for a host of well-known canonical (read "toy") examples, and appears sufficient for situations where the pieces of code satisfy a sort of "master-slave" relationship: anomalous behavior evidenced by programs without this property have been investigated by Colson [Col91]. It would be interesting to see if the expressiveness of the primitive recursive iterators is sufficient to formalize more sophisticated arguments about subtyping, for instance [AC91]. In reasoning about equational theories where the control is *distributed*, the robustness of the approach is not clear. Even if unbounded type recursion is superior for these programs with distributed control, one imagines that they will run aground for other reasons: not only their complexity, but the tremendous difficulty of proving termination of distributed programs.

Acknowledgements. It was Jeff Sanders who suggested that the approach of some earlier work [Mai91] on reasoning about programs based their on polymorphic data types be extended; I thank him for his encouragement and suggestions. Luca Cardelli has been particularly helpful with constructive comments and copious criticism. I also wish to thank Hassan Ait-Kaci, Andrew Black, Daniel Leivant, Patrick O'Keefe, and Mitchell Wand for their advice and their willingness to serve as listeners and critics.

References

[AC91] R. M. Amadio and L. Cardelli. *Subtyping recursive types.* In **Eighteenth Annual ACM Symposium on Principles of Programming Languages**, 1991, pp. 104–118.

[BB85] C. Böhm and A. Berarducci. *Automatic synthesis of typed λ-programs on term algebras.* **Theoretical Computer Science** 39, pp. 135–154.

[BGS90] V. Breazu-Tannen, C. A. Gunter, A. Scedrov. *Computing with coercions.* **1990 ACM Conference on Lisp and Functional Programming**, pp. 44–60.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. A. Gunter, A. Scedrov. *Inheritance as implicit coercion.* Research Report MS-CIS-89-01, Department of Computer and Information Science, University of Pennsylvania.

[Bru92] K. Bruce. *A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics.* Technical Report CS-92-01, Williams College.

[CCHOM89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. *F-bounded polymorphism for object-oriented programming.* In **Fourth ACM Conference on Functional Programming Languages and Computer Architecture,** 1989, pp. 273–280.

[Car88] L. Cardelli. *A semantics of multiple inheritance.* **Information and Computation** 76:2/3, 1988, pp. 138–164.

[Car89] L. Cardelli. *Typeful programming.* Lecture Notes for the IFIP Advanced Seminar on Formal Methods in Programming Language Semantics, Rio de Janeiro, Brazil, 1989. See also SRC Report 45, Digital Equipment Corporation.

[Car91] L. Cardelli. *Extensible records in a pure calculus of subtyping.* Unpublished draft.

[CM91] L. Cardelli and J. C. Mitchell. *Operations on records.* **Mathematical Structures in Computer Science** 1:1, 1991, pp. 3–48.

[CW85] L. Cardelli and P. Wegner. *On understanding types, data abstraction, and polymorphism.* **Computing Surveys** 17(4):471–522, 1985.

[Col91] L. Colson. *About primitive recursive algorithms.* **Theoretical Computer Science** 83 (1991), pp. 57–69.

[CHC90] W. R. Cook, W. L. Hill, and P. S. Canning. *Inheritance is not subtyping.* **Seventeenth Annual ACM Symposium on Principles of Programming Languages**, 1990, pp. 125–135.

[CH88] T. Coquand and G. Huet. *The calculus of constructions.* **Information and Computation** 76:2/3, 1988, pp. 95–120.

[Gir72] J.-Y. Girard, *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. **Proofs and Types.** Cambridge University Press, 1989.

[Hue90] G. Huet. *A uniform approach to type theory.* In **Logical Foundations of Functional Programming**, ed. G. Huet, pp. 337–398. Addison Wesley, 1990.

[Lei90] D. Leivant. *Contracting proofs to programs.* In **Logic and Computer Science**, ed. P. Odifreddi, pp. 279–328. Academic Press, 1990.

[Mai91] H. G. Mairson. *Outline of a proof theory of parametricity.* In **Fifth ACM Conference on Functional Programming Languages and Computer Architecture.** LNCS 523, 1991, pp. 313–327.

[Pie91] B. Pierce. *Bounded quantification is undecidable.* Research Report CMU-CS-91-161, School of Computer Science, Carnegie Mellon University, 1991.

[Rém89] D. Rémy. *Typechecking records and variants in a natural extension of ML.* **Sixteenth Annual ACM Symposium on Principles of Programming Languages**, 1989, pp. 60–76.

[Wan89] M. Wand. *Type inference for record concatenation and simple objects.* **Fourth IEEE Symposium on Logic in Computer Science**, 1989, pp. 92–97.

A Subtyping recursive types

We use the higher-order iterators to derive a standard inference rule about subtyping. Recall the definition

$$\begin{gathered} \mathit{Unroll}\ [N : \mathsf{Pred}_2][\Omega : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}] = \\ N\ ([P : \mathsf{Pred}]\ \Omega\ P\ P)\ (\Omega\ \mathit{True}\ \mathit{False}). \end{gathered}$$

First we prove

$$\begin{aligned} \Theta = \\ &\{\Phi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}\} \\ &\{\Psi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}\} \\ &\{T : (\mathsf{Pred} \to \mathsf{Pred}) \to (\mathsf{Pred} \to \mathsf{Pred})\} \\ &\quad (\mathrm{CN}\ T) \to \\ &\quad (\{P : \mathsf{Pred}\}\{Q : \mathsf{Pred}\}(\mathit{Subtype}\ P\ Q) \to \\ &\qquad (\mathit{Subtype}\ (\Phi P)\,(\Psi Q))) \to \\ &\quad \mathit{Subtype}\ (\mathit{Unroll}\ T\ \Phi)\,(\mathit{Unroll}\ T\ \Psi) \end{aligned}$$

In context $[\Phi : \mathsf{Pred}_1][\Psi : \mathsf{Pred}_1][T : \mathsf{Pred}_2][\mathcal{N} : \mathrm{CN}\ T]\ [\mathcal{I} : \{P : \mathsf{Pred}\}\{Q : \mathsf{Pred}\}\ (\mathit{Subtype}\ P\,Q) \to (\mathit{Subtype}\ \ (\Phi P)\,(\Psi Q))]$, observe that $\mathcal{N}\ [T' : \mathsf{Pred}_2]\ \mathit{Subtype}(\mathit{Unroll}\ T'\ \ \Phi)\,(\mathit{Unroll}\ T'\,\Psi)$ has type $\Theta_1 \to \Theta_2 \to \mathit{Subtype}\ (T\ \ \Phi\ \mathit{True})\ (T\ \ \Psi\ \mathit{True})$, where Θ_1 is

$$\begin{aligned} &\{X : \mathsf{Pred}_2\}(\mathit{Subtype}\ (\mathit{Unroll}\ X\ \Phi) \\ &\qquad (\mathit{Unroll}\ X\ \Psi)) \to \\ &\quad (\mathit{Subtype}\ (\mathit{Unroll}\ (\mathrm{Succ}\ X)\ \Phi) \\ &\qquad (\mathit{Unroll}\ (\mathrm{Succ}\ X)\ \Psi) \end{aligned}$$

and Θ_2 is $\mathit{Subtype}\ (\mathit{Unroll}\ \mathrm{Zero}\ \Phi)\ (\mathit{Unroll}\ \mathrm{Zero}\ \Psi)$

Assumption Θ_1 has proof

$$[X : \mathsf{Pred}_2]\,\mathcal{I}\ (\mathit{Unroll}\ X\ \Phi)\,(\mathit{Unroll}\ X\ \Psi),$$

and Θ_2 has a trivial proof, recalling the iterative properties of Succ and Zero. Call $\mathcal{C}$ the proof of Θ. Defining *Object-type* **as earlier**, we can prove the standard subtyping rule

$$\{\Phi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}\}$$
$$\{\Psi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}\}$$
$$(\{P : \mathsf{Pred}\}\{Q : \mathsf{Pred}\}(\mathit{Subtype}\ P\, Q) \to (\mathit{Subtype}\ (\Phi\ P\, P)\, (\Psi\ Q\, Q))) \to \mathit{Subtype}\ (\mathit{Object\text{-}type}\ \Phi)\ (\mathit{Object\text{-}type}\ \Psi)$$

The construction with this type is

$$[\Phi : \mathsf{Pred} \to \mathsf{Pred}]$$
$$[\Psi : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}]$$
$$[\mathit{Prf}_1 : \{P : \mathsf{Pred}\}\{Q : \mathsf{Pred}\}(\mathit{Subtype}\ P\, Q) \to (\mathit{Subtype}\ (\Phi P)\, (\Psi Q))]$$
$$[\mathit{obj} : \mathsf{Term}]$$
$$[\mathit{Prf}_2 : \{T : \mathsf{Pred}_2\}(\mathrm{CN}\ T) \to \mathit{Unroll}\ T\, \Phi\, \mathit{obj}]$$
$$[T : \mathsf{Pred}_2][\mathcal{N} : \mathrm{CN}\ T]$$
$$\mathcal{C}\ \Phi\ \Psi\ T\ \mathcal{N}\ \mathit{Prf}_1\ (\mathit{Prf}_2\ T\ \mathcal{N})$$

B Proof of rule (TE7)

To prove

$$\{\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}\}\{\mathcal{P} : \mathit{Proto}\ \Phi\}$$
$$\{\ell : \mathsf{Term}\}\{L : \mathit{Label}\ \ell\}\{A : \mathsf{Pred}\}\{x : \mathsf{Term}\}$$
$$((\mathit{Project}\ (\mathit{Augment}\ \Phi\, \ell\, A)\, \ell)\, x) \ \leftrightarrow\ (A\ x)$$

where

$$\mathit{Project} = [\Phi : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}]\ [\ell : \mathsf{Term}][y : \mathsf{Term}]\ \{\mathit{rec} : \mathsf{Term}\}(Q\ y\, \langle \mathit{rec}\ \ell\rangle) \to (\Phi\ \mathit{rec}\ \ell)$$

observe that

$$\mathit{Project}\ (\mathit{Augment}\ \Phi\, \ell\, A)\, \ell = [y : \mathsf{Term}]\{\mathit{rec} : \mathsf{Term}\}\ (Q\ y\, \langle \mathit{rec}\ \ell\rangle) \to (\textsc{and}\ ((Q\ \ell\, \ell) \to (A\ \langle \mathit{rec}\ \ell\rangle))\ (\textsc{not}\ (Q\ \ell\, \ell) \to \Phi\ \mathit{rec}\, \ell)).$$

Suppose $\mathit{Prf} : (\mathit{Project}\ (\mathit{Augment}\ \Phi\, \ell\, A)\, \ell)\ y$; we would like to construct a proof of $A\ y$. Assume the context contains $[\mathbf{K} : \mathsf{Term}]$ and a familiar axiom,

$$[\text{Def-}\mathbf{K} : \{x : \mathsf{Term}\}\{y : \mathsf{Term}\}\ Q\ x\, \langle \mathbf{K}\ x\, y\rangle].$$

Then

$$\mathit{Prf}\, \langle \mathbf{K}\ y\rangle\ (\text{Def-}\mathbf{K}\ y\, \ell) : (\textsc{and}\ ((Q\ \ell\, \ell) \to (A\ \langle \mathbf{K}\ y\, \ell\rangle))\ ((\textsc{not}\ (Q\ \ell\, \ell)) \to \Phi\ \langle \mathbf{K}\ y\rangle\, \ell))$$

From this construction, it is easy to get a proof of $A\ \langle \mathbf{K}\ y\, \ell\rangle$, and from the definition of Q and Def-$\mathbf{K}$ we can then derive a proof of $A\ y$.

In the other direction, given a proof of $A\ y$, from hypothesis $Q\ y\, \langle \mathit{rec}\ \ell\rangle$ it is easy to prove $A\ \langle \mathit{rec}\ \ell\rangle$, where rec and ℓ are terms. To prove both halves of the conjunction, the first can be proved by introducing (unused!) hypothesis $Q\ \ell\, \ell$, and the second can be proved by assuming $\textsc{not}\ (Q\ \ell\, \ell)$ and using axiom Eq_4: since $Q\ \ell\, \ell$ is certainly provable, from a proof of **true** = **false** we can prove anything, including $\Phi\ \mathit{rec}\ \ell$. Discharging hypotheses yields the result.

C Proof of rule (TE9)

To prove

$$\{R : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}\}$$
$$\{\mathit{rec} : \mathsf{Term}\}\{\mathcal{P} : \mathit{Proto}\ R\}$$
$$((\mathit{Recordtype}\ R)\ \mathit{rec}) \leftrightarrow ((\mathit{Recordtype}\ (\mathit{Augment}\ R\ \ell\ (\mathit{Project}\ R\ \ell)))\ \mathit{rec}),$$

observe that this proposition converts (in the $\to$ direction) to

$$\{R : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}\}$$
$$\{\mathit{rec} : \mathsf{Term}\}\{\mathcal{P} : \mathit{Proto}\ R\}$$
$$(\{x : \mathsf{Term}\}(\mathit{Label}\ x) \to R\ \mathit{rec}\ x) \to$$
$$(\{x : \mathsf{Term}\}(\mathit{Label}\ x) \to (\textsc{and}\ ((Q\ x\, \ell) \to (\mathit{Project}\ R\, \ell)\, \langle \mathit{rec}\ x\rangle)\ ((\textsc{not}\ (Q\ x\, \ell) \to R\ \mathit{rec}\ x)))).$$

Since

$$(\mathit{Project}\ R\ \ell)\ \langle \mathit{rec}\ x\rangle = \{r : \mathsf{Term}\}(Q\ \langle \mathit{rec}\ x\rangle\, \langle r\ \ell\rangle) \to R\ r\, \ell,$$

from precondition $Q\ x\, \ell$ we have

$$(\mathit{Project}\ R\ \ell)\ \langle \mathit{rec}\ \ell\rangle = \{r : \mathsf{Term}\}(Q\ \langle \mathit{rec}\ \ell\rangle\, \langle r\ \ell\rangle) \to R\ r\, \ell.$$

To prove the theorem, then, we need to show $(R\ \mathit{rec}\ \ell) \to (\mathit{Project}\ R\, \ell)\ \langle \mathit{rec}\ \ell\rangle$. This we do *by induction on* R, using the induction axiom $\mathit{Proto}\ R$, i.e.,

$$\mathcal{P}\ [P : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Prop}]\ (R\ \mathit{rec}\ \ell) \to (\mathit{Project}\ R\, \ell)\ \langle \mathit{rec}\ \ell\rangle.$$

The rest of the proof is routine.