Verifikation objektorientierter Programme

Martin Steffen

zusammen mit M. Hofmann, W. Naraschewski und T. Stroup

Juni 1998

Gliederung

- Motivation
- Programme & Beweise
- Kodierung: Objekte, Klassen, Vererbung
- Schluß

Motivation

- relativ gut untersucht: Theorie/Semantik von OO-Sprachen,
- Verifikation?
- Herausforderung: reichhaltiges Arsenal an komplexen Sprachmerkmalen

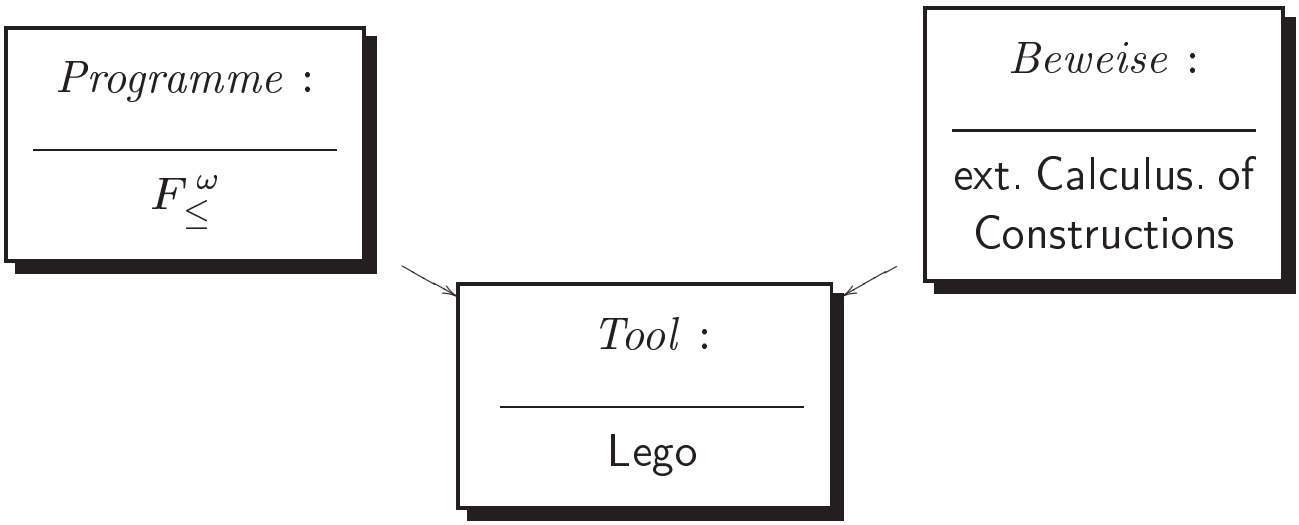
Ziele:

1. formale Verifikation objektorientierte Programme, und zwar unter
2. Ausnutzung der OO-Struktur der Programme $\Rightarrow$
3. Strukturierung der Beweise

Ansatz

mathematischer Rahmen: Typentheorie

- uniforme Theorie für Programme und Beweise
- ausdrucksstarke Logik
- Implementierungen, Beweisprüfer
- Wahl hier:



Basiskalkül für OO-Sprachen

$$F^{\omega}_{\leq}$$

- statisch typisierter λ-Kalkül mit Subtyping
- ausdruckstark $\Rightarrow$ Basiskalkül für typisierte OO-Sprachen
- Modell– und beweistheoretisch gut untersucht
- insbesondere: Objektmodell von Pierce & Turner
  - Klassen als Strukturierungskonzept
  - parametriche Polymorphie
  - Subtypepolymorphie
  - Kapselung von Objekten
  - Klassenvererbung zur Wiederverwendung von Code
  - dynamische Methodenbindung

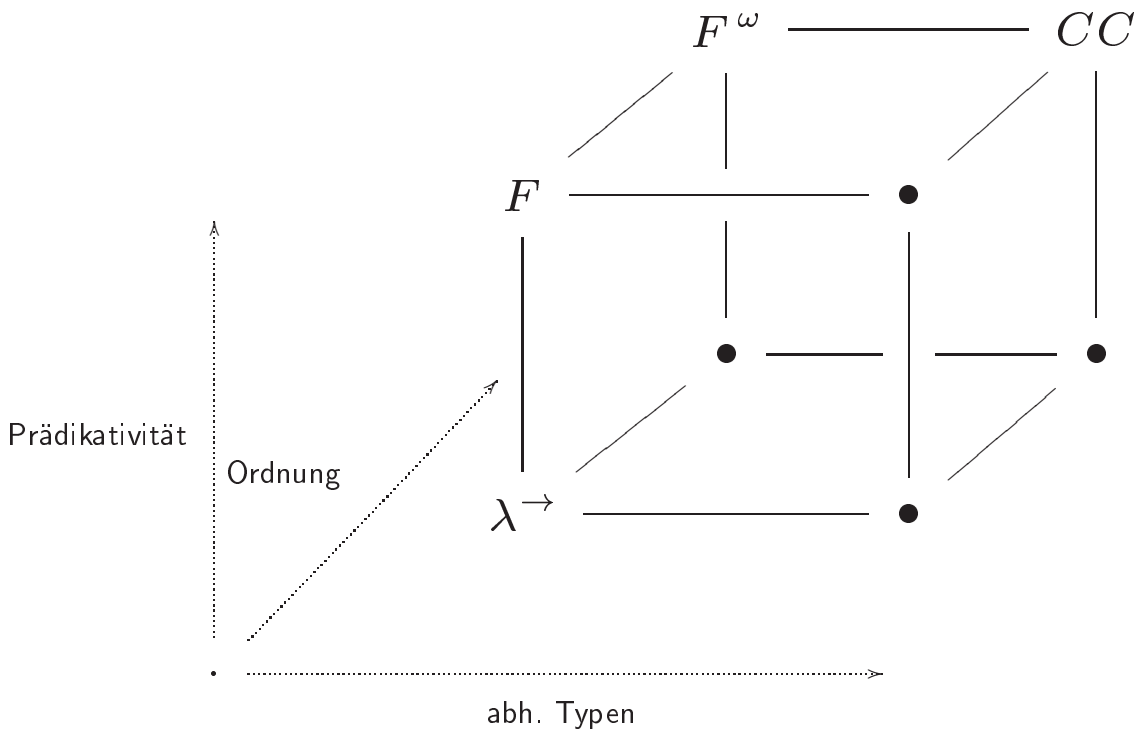
$F^{\omega}_{\leq}$: Syntax

K	$::=$	$Type$	kind of types
	$\mid$	$K \to K$	kind of type operators
T	$::=$	A	type variable
	$\mid$	$Fun(A{:}K)T$	type operator
	$\mid$	$T\ T$	application of a type operator
	$\mid$	$Top(K)$	maximal type
	$\mid$	$T \to T$	function type
	$\mid$	$All(A{\leq}T)T$	universally quantified type
t	$::=$	x	variable
	$\mid$	$fun(x{:}T)t$	abstraction
	$\mid$	$t\ t$	application
	$\mid$	$fun(A{\leq}T)t$	type abstraction
	$\mid$	$t\ T$	type application

Lego

- interaktiver Beweisassistent
- formale Grundlage: (Extended) Calculus of Constructions (ECC) = intuitionistische, imprädikative Logik höherer Ordnung
- Vorteil:

CC = F^{ω}+ abhängige Typen



Isomorphie von Curry & Howard

Systeme nat. Deduktion

Bsp: minimale prop. Logik

$$\frac{\begin{array}{c}[A]\\ \vdots\\ A\end{array}}{A \implies A}$$

λ-Kalküle

Bsp: einfach typ. λ-Kalkül

$$\frac{x{:}A \vdash x{:}A}{\vdash (\lambda x{:}A.x){:}A \to A}$$

- **Satz:** Formel A herleitbar gdw. es ein Term t gibt mit $\vdash t : A$.
- Schlagwort:

Propositions as types

oder

programs as proofs

Beispiel: minimale Logik

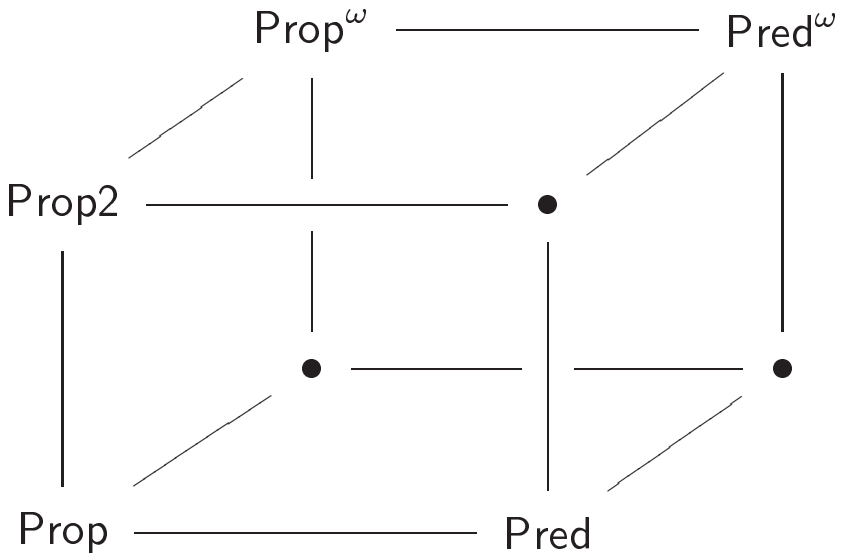
$$\frac{}{A}\ (\text{Ax}) \qquad \frac{}{\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash x{:}T}\ (\text{Ax})$$

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}\ (\to\text{-I}) \qquad \frac{\Gamma,\ x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.t\ \in\ S \to T}\ (\to\text{-I})$$

$$\frac{A \to B \qquad A}{B}\ (\text{MP}) \qquad \frac{\Gamma \vdash s : S \to T \qquad t : S}{\Gamma \vdash s\ t : T}\ (\to\text{-E})$$

Curry/Howard (II)

- Für jeden λ-Kalkül des Würfel: eine entsprechende Logik
-



Objektmodell in $F^{\omega}_{\leq}$

- Objekt:
  - Zustand
  - Methoden
  - Kapselung
  - Instanz einer Klasse
- Klasse: Definition von Objekten
- Klassenhierarchie, Vererbung von Methoden[1]
- späte/dynamische Bindung

[1] Einfachvererbung

Kodierung: Subtyping

- F^ω ist Teilsprache vom CC, also auch von Lego $\Rightarrow$
- einzige Schwierigkeit: Subtyping $\Gamma \vdash S \leq T$.
- Intuition von Subtyping:

  Subsumption

- explizite Darstellung durch Umwandlungsfunktionen (*coercions*), d.h.

  $$S \leq T \text{ durch } get^{S,T} : S \to T.$$

- für funktionalen Update reicht das nicht $\Rightarrow$

  $$put^{S,T} : S \to T \to S.$$

  Codierung von Subtyping:

$$S \leq T \stackrel{\text{def}}{=} [\ get : S \to T,\ put : S \to T \to S,\ gpOK : GetPutLaws\ get\ put\ ]$$

Kodierung: Objekte

- Objekte ohne Beweiskomponente ($F^{\omega}_{\leq}$):
  - Signatur: $Sig : \star \to \star$
  - Kapselung durch Existenzquantor

$$Object(Sig) = \exists\, Rep : \star\, .\, [state : Rep,\, ops : Sig\ Rep]$$

Beispiel 1 (Punkte) Signatur/Schnittstelle $SigPoint$ als Typoperator $\star \to \star$ und $Point$ als Typ der Punkte:

$$SigPoint \stackrel{\text{def}}{=} \lambda\, Rep : \star\, .\, [\ \begin{array}{l} getX : Rep \to nat, \\ setX : Rep \to nat \to Rep, \\ inc1 : Rep \to Rep\ ] \end{array}$$

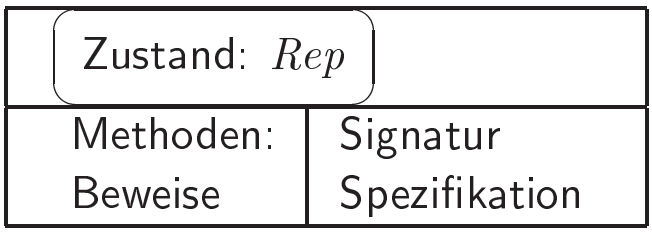
$$Point \stackrel{\text{def}}{=} Object\ \ SigPoint$$

Konkrete Implementierung:

$$\begin{array}{rcl} opsPoint & \stackrel{\text{def}}{=} & [\quad getX = \lambda n{:}\, nat\, .\, n, \\ & & \quad\ setX = \lambda n{:}\, nat\, .\, \lambda m{:}\, nat\, .\, m, \\ & & \quad\ inc1 = \lambda n{:}\, nat\, .\, n + 1\ ] \\ & & :\quad SigPoint\ nat\ . \end{array}$$

Objekte mit Beweiskomponente

- Ausnutzung der Curry-Howardschen Isomorphie
- Objekt: nun drei Komponenten:



- Formaler: gegeben $Sig:\ \star \to \star$
  $Spec:\ \forall\, Rep : \star\,.\ (Sig\ Rep) \to \star$
  dann ist der Typ der Objekte:

$$Object \stackrel{\text{def}}{=} \exists\, Rep : \star\,.\ [state : Rep, ops : Sig\ Rep, prfs : Spec\ Rep\ ops]$$

- tech. Nebenbemerkung: die übliche F-Kodierung des $\exists$ ist hier zu schwach

Beispiel

Beispiel 2 (Punkte) Sei die Signatur *SigPoint* gegeben wie gehabt. Sei eine Implementierung $ops : SigPoint\ \ Rep$ angenommen, dann lautet die Spezifikation:

$$\begin{aligned} SpecPoint \stackrel{\text{def}}{=} & \\ ops.getX(ops.setX\ \ r\ \ n) \quad &=_L \quad n \\ ops.getX(ops.inc1\ \ r) \quad &=_L \quad (ops.getX\ \ r) + 1 \\ \ldots & \end{aligned}$$

Der Typ der Punkte entsprechend

$$Point \stackrel{\text{def}}{=} Object\ \ SigPoint\ \ SpecPoint$$

Konstruktion konkreter Punkte:

1. konkreter Zustand
2. Implementierung der Methoden
3. Beweis (= Beweisterm), daß die Implementierung die Spezifikation erfüllt

$$MyPoint \stackrel{\text{def}}{=} ObjectIntro\ \ 3\ \ opsPoint\ \ prfsPoint$$

Beispiel (2)

- zur Abschreckung? Definition des Objektkonstruktors

$$
\begin{array}{l}
ObjectIntro \stackrel{\text{def}}{=} \\
\quad \lambda\, mystate : Rep\ . \\
\quad\quad \lambda\, myops : Sig\ Rep\ . \\
\quad\quad\quad \lambda\, myprfs : Spec\ Rep\ myops\ . \\
\quad\quad\quad\quad pack\ \ (\lambda\, Rep : \star\ .\ [\ state : Rep, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ops : Sig\ Rep, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad prfs : Spec\ Rep\ ops\ ]) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Rep \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [\ state = mystate, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ ops = myops, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ prfs = myprfs\ ] \\
\quad\quad : \ Rep \rightarrow \forall\, myops : Sig\ Rep\ .\ (Spec\ Rep\ myops) \rightarrow \\
\quad\quad\quad Object\ Sig\ Spec
\end{array}
$$

- Aber: der Beweisassistent assistiert!

Was bringt das alles bisher?

Erweiterung der Objekte um Beweiskomponente $\Rightarrow$

- Kapselung von Beweisen
- Schittstelle stellt Beweismethoden zur Verfügung:
- Einheitlicher formaler Rahmen
- Programmextraktion möglich: Projektion des Programmanteils
- Paket aus Programm + Beweis: "Deliverable"
- ähnliche Idee auch: "proof-carrying code" ($\neq$ "trusted code")

Was noch?

- Vererbung von Beweisen
- Beweise über dynamisch-gebundene Methoden (mit $self$)
- dynamische Bindung von Beweisen

Klassen

- Klasse = Code für Objekte
- zwei Operationen auf Klassen möglich
  1. Instantiierung mit new.
  2. Erweiterung/Spezialisierung = Vererbung mit $inherit$[2]
- Klasse: Repräsentierungstyp + Implementierung der Methode und ihrer Beweise (keine Kapselung)

Gegeben: Signatur, Spezifikation sowie ein Repräsentierungstyp $ClassR \Rightarrow$ Typ der Klassen:

$$\begin{aligned} Class \stackrel{\text{def}}{=} \\ \forall\, Rep : \star\,.\ (Rep \leq ClassR) \rightarrow \\ [ops : Sig\ Rep, prfs : Spec\ Rep\ ops] \end{aligned}$$

Beispiel 3 (Punkte) Typ der Klassen für Punkte:

$$PointClass \stackrel{\text{def}}{=} Class\ nat\ SigPoint\ SpecPoint$$

[2] oder $extends$.

self-Methoden und Instantiierung

- wichtiges Sprachmittel von OO: *self*-Methoden
- *self*:[3] dynamische Methodenbindung (späte Bingung, *late binding*): *self* meint die dynamische, aktuelle Instanz, nicht die Klasse
- Standardlösung: *self* als Parameter:

Beispiel 4 (Klasse der Punkte)

$$
\begin{array}{l}
MyPointClass \stackrel{\text{def}}{=} \\
\quad \lambda\, Rep : \star\, .\ \lambda\, gp : Rep \leq nat\, . \\
\quad \lambda\, self : [ops : SigPoint\ Rep, prfs : SpecPoint\ Rep\ ops]. \\
\qquad [ops = opsPointClass, prfs = prfsPointClass]
\end{array}
$$

wobei (Vgl. Folie 12)

$$
\begin{array}{l}
opsPointClass = \\
\quad [\ getX \ = \ \lambda r\colon Rep\, .\ \ gp.get\ \ r, \\
\qquad setX \ = \ \lambda r\colon Rep\, .\ \lambda n\colon nat\, .\ \ gp.put\ \ r\ n, \\
\qquad inc1 \ = \ \lambda r\colon Rep\, .\ \ self.ops.setX\ \ r \\
\qquad\qquad\qquad\qquad (self.ops.getX\ \ r) + 1\ ]
\end{array}
$$

[3] oder *this* o. ä.

self-Methoden und Instantiierung (2)

$$
\begin{array}{l}
Class \stackrel{\text{def}}{=} \\
\quad \forall\, Rep : \star\,.\ (Rep \leq ClassR) \rightarrow \\
\quad\quad [ops : Sig\ Rep, prfs : Spec\ Rep\ ops] \rightarrow \qquad (self) \\
\quad\quad\quad [ops : Sig\ Rep, prfs : Spec\ Rep\ ops]
\end{array}
$$

- $new =$ im Prinzip eine Funktion

$$new : Klasse \rightarrow Objekt$$

- im Detail: etwas aufwendiger:

Fixpunktauflösung & Kapselung

Vererbungsoperator *inherit*

- im Prinzip: Funktion

$$inherit : Oberklasse \rightarrow Unterklasse$$

- im Detail: etwas aufwendiger $\Rightarrow$

Assume implicitly a representation type $SuperR : \star$, a signature $SuperSig : \star \rightarrow \star$, and a specification $SuperSpec$ of the superclass, which itself has type $\forall\, Rep : \star\,.\ (SuperSig\ Rep) \rightarrow \star$. In addition, assume for the subclass a representation type $SubR$, a signature $SubSig$, and a specification $SubSpec$ correspondingly. Finally, assume a proof $gp_{SubR \leq SuperR}$ of type $SubR \leq SuperR$ and two coercion functions $co\_sig : \forall\, Rep \mid \star\,.\ (SubSig\ Rep) \rightarrow (SuperSig\ Rep)$ and co_spec of type $\forall\, Rep \mid \star\,.\ \forall\, ops \mid SubSig\ Rep\,.\ (SubSpec\ Rep\ ops) \rightarrow (SuperSpec\ Rep\ (co\_sig\ ops))$. The *inheritance operator* is then defined as follows:

$$
\begin{array}{l}
inherit \stackrel{\text{def}}{=} \\
\quad \lambda\, SuperClass : Class\ SuperR\ SuperSig\ SuperSpec\ . \\
\quad \lambda\, build : \ \forall\, Rep : \star\ .\ (Rep \leq SubR) \rightarrow \\
\qquad\qquad [ops : SuperSig\ Rep, \\
\qquad\qquad\ prfs : SuperSpec\ Rep\ ops] \rightarrow \qquad (super) \\
\qquad\qquad [ops : SubSig\ Rep, \\
\qquad\qquad\ prfs : SubSpec\ Rep\ ops] \rightarrow \qquad (self) \\
\qquad\qquad [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops]. \\
\quad (\ \ \lambda\, Rep : \star\ .\ \lambda\, gp_{Rep \leq SubR} : Rep \leq SubR\ . \\
\qquad \lambda\, self : [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops]. \\
\qquad build\ \ Rep\ gp_{Rep \leq SubR} \\
\qquad\qquad (SuperClass\ \ Rep \\
\qquad\qquad\qquad\qquad (trans_{\leq}\ \ gp_{Rep \leq SubR} \\
\qquad\qquad\qquad\qquad\qquad\ gp_{SubR \leq SuperR}) \\
\qquad\qquad\qquad\qquad [\ ops\ \ =\ \ co\_sig\ \ self.ops, \\
\qquad\qquad\qquad\qquad\ \ prfs\ \ =\ \ co\_spec\ \ self.prfs\ ]) \\
\qquad\qquad self\ \ ) \\
: \ (Class\ SuperR\ SuperSig\ SuperSpec) \rightarrow \\
\quad (\forall\, Rep : \star\ .\ (Rep \leq SubR) \rightarrow \\
\qquad [ops : SuperSig\ Rep, prfs : SuperSpec\ Rep\ ops] \rightarrow \\
\qquad [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops] \rightarrow \\
\qquad [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops]) \rightarrow \\
\quad (Class\ SubR\ SubSig\ SubSpec)
\end{array}
$$

inherit als Taktik

- Den Beweisterm *inherit* bekommt man nicht zu Gesicht
- als Taktik angewandt auf eine zu implementierende Klasse liefert *inherit* (grob) als Beweisverpflichtung:
  1. die Oberklasse
  2. Zusammenhang zwischen Oberklasse und Unterklasse:
     - Beweis der Untertypbeziehung zwischen den Repräsentierungen
     - Beweis der Untertypbeziehung zwischen den Signaturen
     - Implementierung der Unterklasse; dabei ist Verwendung
       * der Methoden
       * der Beweise

       der Oberklasse möglich

  $\Rightarrow$

  Vererbung von Beweisen über *super*

Resumée

- OO-Strukturierung der Verifikation::
  - Kapselung
  - Wiederverwendung

  von Beweisen
- Beweise über dynamisch gebundene Methoden möglich
- abstrakte ("virtuelle") Beweise möglich

Ausblick

- Mehrfachvererbung: aufbauend auf $F^{\omega}_{\wedge}$
- Vermeidung der expliziten get und put-Funktionen für $\leq$ $\Rightarrow$ automatische Generierung für monotone Operatoren möglich
- syntaktischen Zucker: z.B. `protected` oder ähnliches.
- Übertragung auf "Konstruktorklassen" in Haskell
- Beweise über die Gleichheit von Objekten: Bisimulation
- Beweise über Eigenschaften der Kodierung selbst
- andere (komplexere) Objektmodelle (z.B. zur Unterstützung binärer Methoden.)
- Case-studies
- Keine Kodierung, sondern Integration von OO-Features in Beweisechecker selbst

Literatur

Literatur

[Bar92] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and Thomas Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1: Mathematical Structures, pages 117–309. Oxford University Press, 1992.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[HNSS98] Martin Hofmann, Wolfgang Naraschewski, Martin Steffen, and Terry Stroup. Inheritance of proofs. *Theory and Practice of Object Systems (Tapos), Special Issue on Third Workshop on Foundations of Object-Oriented Languages (FOOL 3), July 1996*, 4(1):51–69, January 1998. An extended version appeared as Interner Bericht, Universität Erlangen-Nürnberg, IMMDVII-5/96.

[HP92] Martin Hofmann and Benjamin Pierce. An abstract view of objects and subtyping. Technical Report ECS-LFCS-92-225, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1992.

[Leg98] Lego. The LEGO proof assistant. http://www.dcs.ed.ac.uk/home/lego, 1998.

[LP92] Zhaohui Luo and Randy Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1992.

[McK92] James Hugh McKinna. *Delivarables: A Categorical Approach to Program Development in Type Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[PT94] Benjamin Pierce and David Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".