Types and consistency in combinatory algebras

Author(s):
Otth, Daniel

Publication Date:
1992

Permanent Link:

https://doi.org/10.3929/ethz-a-000646491 →

Diss. ETH No. 9800

Types and Consistency in Combinatory Algebras

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Mathematics

presented by
DANIEL OTTH
Dipl. Math. ETH
born 22. January 1964
citizen of Innertkirchen (BE)

accepted on the recommendation of
Prof. Dr. E. Engeler, examiner
Prof. Dr. H. Läuchli, co-examiner

1992

Abstract

Logic programs are given by a set of rules over a first-order language. If we iterate the rule formation process, this gives rise to cumulative logic programs, *i.e.* in cumulative logic programs we have rules over rules. To program in this more powerful environment we need an adequate *type system* that classifies programs of similar complexity, a *notion of consistency* that distinguishes the meaningful programs and a *notion of entailment* that closes programs under implicit meanings. We need this not only for sets of simple rules but for cumulative logic programs which contain arbitrary nested rules.

The graph model for the combinatory algebra, as it was introduced by Engeler (1981) is the ideal framework to study cumulative logic programs. These programs turn out to be elements of a graph model and are therefore objects of a combinatory algebra. Hence, we can study equations between cumulative logic programs and find solutions to these equations that are themselves programs rather than sets of simple facts as in traditional logic programs. Further, one can compare directly formal descriptions of a function with graphs of functions, *e.g.* we can use in a cumulative logic program the function sin as a symbol and at the same time the graph which maps every element x to $\sin x$.

Types are introduced in combinatory algebras to constrain the application of combinators. So we can for example disallow the application of a constant to anything, *i.e.* we ensure that there is really a function on the left side of an application. For cumulative logic programs, or more generallly for graph models, types will say something about how much the rules are nested. There are many ways to construct such type systems, *e.g.* projections, partial equivalence relations, di-domains, ideals etc. We will use a "type as principal ideals" approach (Otth 1991). This has the advantage that we have a 1-1 correspondence between the types and the elements of a graph model. Hence the types themselves form a combinatory algebra and there is a fixpoint operator, a normalizer etc. available. We do not have to go outside of the graph model to work with types and every cumulative logic program can be viewed as a logic program, an element of a combinatory algebra and a type at the same time. Because of this fact we can present as an application of this type system an easy and natural model for the polymorphic second order lambda calculus (Girard 1971) and give a sound and complete system of axioms for this type system.

For the consistency notion we will develop an effective construction that allows us to lift a consistency notion on the base set to the whole graph model in a unique way. Analogously we will lift a notion of entailment.

Abstrakt

Logikprogramme bestehen aus einer Menge von Regeln über einer logischen Sprache erster Ordnung. Wenn wir den Prozess der Regelbieldung iterieren erhalten wir kumulative Logikprogramme, *d.h.* in kumulativen Logikprogrammen haben wir Regeln über Regeln. Um in dieser mächtigeren Sprache zu programmieren, brauchen wir ein adequates *Typen System* welches Programme ähnlicher komplexität klassifiziert, einen *Konsistenzbegriff* der sinnvolle Programme unterscheidet und einen *Folgerungsbegriff* der Programme unter implizitem Wissen abschliesst. Wir brauchen diese Begriffe nicht nur für kumulative Logikprogramme die einfache Regeln enthalten sondern auch für solche die beliebig verschachtelte Regeln enthalten.

Im Graphmodell für die kombinatorische Algebra, das von Engeler (1981) eingeführt wurde, finden wir das ideale Werkzeug um kumulative Logikprogramme zu untersuchen. Wie sich herausstellt sind kumulative Logikprogramme Elemente eines Graphmodells und dadurch auch Objekte einer kombinatorischen Algebra. So können wir Gleichungssysteme zwischen kumulativen Logikprogrammen untersuchen und Lösungen finden die selber wieder Programme sind und nicht nur Mengen von Fakten wie bei herkömmlichen Logikprogrammen. Weiter können wir unmittelbar formale Beschreibungen einer Funktion mit ihrem Graph vergleichen, *d.h.* wir können in einem kumulativen Logikprogramm die Sinus-Funktion als Symbol sin und gleichzeitig als Graph, der jedes Element x nach $\sin x$ abbildet benutzen.

In einer kombinatorischen Algebra werden Typen eingeführt um die Applikation einzuschränken. So kann zum Beispiel die Applikation einer Konstante auf irgendetwas unterbunden werden, *d.h* wir können uns versichern, dass auf der linken Seite einer Applikation immer eine Funktion steht. In Logikprogrammen oder algemeiner in Graphmodellen, sagen die Typen etwas aus über die Verschachtelung der Regeln. Es gibt viele Möglichkeiten solche Typen Systeme einzuführen. Einige traditionelle Ansätze sind zum Beispiel Projektionen, partielle Äquivalenzrelationen, di-Bereiche, Ideale etc. Wir werden den "Typen als Hauptideale" Ansatz (Otth 1991) verfolgen. Dies bringt uns den Vorteil, dass wir eine eindeutige Zuordnung von Typen und Elemente eines Graphmodell haben. So bilden die Typen selber eine kombinatorische Algebra und wir verfügen über Fixpunktoperatoren, Normalisatoren etc. Wir brauchen also das Graphmodell nicht zu verlassen um mit Typen zu arbeiten. In diesem Lichte kann ein kumulatives Logikprogramm als dreierlei gesehen werden; als Logikprogramm, als Objekt einer kombinatorischen Algebra und als Typ. Gerade wegen dieser Eigenschaft können wir auf eine einfache und natührliche Weise ein Modell für den polymorphen lambda Kalkül zweiter Stufe (Girard 1971) präsentieren und ein korrektes und vollständiges Axiomen System angeben.

Für den Konsistenzbegriff entwickeln wir eine effektive Konstruktion, welche uns erlaubt

einen vorgegebenen Konsistenzbegriff auf der Basismenge eines Graphmodell in einer eindeutigen Weise auf das ganze Graphmodell hochzuziehen. Auf analoge Weise entwickeln wir einen Folgerungsbegriff.