A programming logic for F_ω

Eindhoven University of Technology

Department of Mathematics and Computing Science

A programming logic for F_ω

by

Erik Poll

92/25

Computing Science Note 92/25
Eindhoven, September 1992

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

A programming logic for F_ω

Erik Poll*

Abstract. A programming logic is defined in which F_ω programs can be specified and proven correct. Both the programming language and the logic are typed lambda calculi and can be defined as Pure Type Systems. Programs and correctness proofs, as well as types and specifications, are strictly separated.

* supported by the Dutch organization for scientific research (NWO).
E-mail erik@win.tue.nl

1 Introduction

Typed lambda calculi can be viewed as typed functional programming languages, or, by the Curry-Howard-de Bruijn isomorphism, as logics. The first interpretation – λ-terms are programs and types are data types – has led to the design of powerful type systems for programming languages (see for instance [Rey85] [Car89]). The second interpretation – types are propositions and λ-terms are proofs – is the basis of several systems for the development and mechanical verification of mathematical proofs, such as AUTOMATH [dB80], Martin-Löf's Type Theory [ML79] and the Calculus of Constructions [CH88].

Such a formalism, in which proofs, programs, types and propositions can be expressed, can be a suitable basis for a programming logic, a system in which we can express properties of programs and prove that certain programs satisfy certain specifications.

Most research on programming logics based on type theory concerns so-called integrated programming logics. Here the Curry-Howard-de Bruijn isomorphism is exploited by identifying the notions of program and proof and the notions of type and specification (to a certain extent). This idea dates back to Heyting's semantics of constructive proofs: a constructive proof H of $\forall x{:}\rho.\ pre(x) \Rightarrow \exists y{:}\sigma.\ post(x,y)$ contains an algorithm which, given a term x and a proof H_x of $pre(x)$, returns a term y and a proof H_y of $post(x,y)$. The type of H gives all the relevant information about H; it is its specification. Sufficiently expressive type systems, such as Martin-Löf's Type Theory and the Calculus of Constructions, can be used as programming logics in this way.

The main problem with this approach is that proofs contain redundant information. Typically, a large part of H is devoted to the computation of H_y, and as programmers we are only interested in y. As a result, H is inefficient and difficult to read.

There are different ways to get rid of these irrelevant computations. For Martin-Löf's Type Theory subset types have been introduced to hide computational information (see [NPS90]). Another possibility is program extraction. [PM89] describes this approach for the Calculus of Constructions: from a proof of
$(\forall x{:}\rho.\ pre(x) \Rightarrow \exists y{:}\sigma.\ post(x,y))$ an F_ω program f of type $\rho \to \sigma$ and a proof H_f of $\forall x{:}\rho.\ pre(x) \Rightarrow post(x, f(x))$ can be extracted. In both solutions the distinction between data types and propositions resurfaces.

Instead, we investigate a programming logic based on type theory where programming language and logic are strictly separated. So instead of first constructing a proof H and then extracting f and H_f, we directly construct the program f and its correctness proof H_f. This approach is also considered in [BM90]. Programs and their types, as well as specifications and proofs, are all terms in a typed lambda calculus.

Apart from avoiding the need for program extraction, an advantage is that the program under construction is clearly visible at all times. Design choices can then be made not only on the basis of the specification, but also on the basis of an operational understanding of the algorithm and efficiency considerations.

We define a $\lambda\omega_L$, consisting of a programming language and a logic. The system

is a refinement of the Calculus of Constructions, in which data types and propositions are distinguished, and the possible dependencies are restricted. In particular, we do not allow programs and their types to depend on propositions or proofs. The system can be compactly defined as a Pure Type System (PTS) (see [Bar92]).

The programming language is the PTS $\boldsymbol{\lambda\omega}$, Girard's system F_ω. The same language is used as the basis for the programming languages Quest [Car89] and LEAP [PL89]. It is the strongest system in Barendregt's λ-cube without term dependent types. This means there are no computations on terms just for the sake of type checking which are irrelevant for computing the output. There are representations for many data types in $\boldsymbol{\lambda\omega}$, including abstract data types (see [BB85] and [MP88]).

Although we have programs and separate correctness proofs, we do want to construct these hand in hand, instead of first constructing a program and afterwards proving its correctness. $\boldsymbol{\lambda\omega}$-programs can be annotated in $\boldsymbol{\lambda\omega}_L$. This annotation gives information about a program that cannot be expressed in its type, but which can be expressed by a proposition. For a program f of type σ, this annotation consists of a proof H_f that f satisfies some property ϕ. This property (or specification) ϕ is a predicate on the type σ, and H_f is a proof of the proposition $(\phi\ f)$. For example, for a program f of type $\mathit{intlist} \rightarrow \mathit{intlist}$, the annotation could be a proof of the proposition $(\forall l : \mathit{intlist}.\ \mathit{sorted}(f(l)) \wedge \mathit{permut}(l, f(l)))$.

2 Pure Type Systems

Pure Type Systems, introduced by Berardi and Terlouw, are a generalization of the systems in Barendregt's λ-cube. See [Bar92] for a discussion of PTS's and their properties.

Definition 1 (Pure Type Systems).
A Pure Type System (PTS) is specified by a triple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ with

- $\boldsymbol{S}$ is a set of symbols called the *sorts*
- $\boldsymbol{A} \subseteq \boldsymbol{S} \times \boldsymbol{S}$, a set of *axioms*
- $\boldsymbol{R} \subseteq \boldsymbol{S} \times \boldsymbol{S}$, a set of *rules* [2]

The set of terms M and contexts Γ is given by

$$M ::= x \mid s \mid (MM) \mid (\lambda x{:}M.\ M) \mid (\Pi x{:}M.\ M)$$
$$\Gamma ::= \epsilon \mid \Gamma, x{:}M$$

where x is a variable and s is a sort. For type judgements of the form $\Gamma \vdash b : B$ - in context Γ term B is the type of term b - we have the following derivation rules

$$(\mathit{axiom}) \quad \epsilon \vdash s_1 : s_2 \qquad \text{for } s_1 : s_2 \in \boldsymbol{A}$$

$$(\mathit{start}) \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad \text{where } x \text{ is } \Gamma\text{-fresh}$$

$$(\mathit{weakening}) \quad \frac{\Gamma \vdash b : B \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B} \qquad \text{where } x \text{ is } \Gamma\text{-fresh}$$

$$(\mathit{formation}) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_2} \qquad \text{for } (s_1, s_2) \in \boldsymbol{R}$$

2 In [Bar92] PTS rules are triples (s_1, s_2, s_3), but here this simpler definition suffices.

$$(abstraction)\ \frac{\Gamma, x:A \vdash b:B \quad \Gamma \vdash (\Pi x{:}A.\ B):s}{\Gamma \vdash (\lambda x{:}A.\ b):(\Pi x{:}A.\ B)}$$

$$(application)\ \frac{\Gamma \vdash b:(\Pi x{:}A.\ B) \quad \Gamma \vdash a:A}{\Gamma \vdash ba:B[x:=a]}$$

$$(conversion)\ \frac{\Gamma \vdash b:B \quad \Gamma \vdash B':s \quad B \simeq B'}{\Gamma \vdash b:B'}$$

where s ranges over sorts, i.e. $s \in \boldsymbol{S}$, and $\simeq$ is β-equality.

Notation. $\Gamma \vdash A : B : C$ is short for $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$. We write $\twoheadrightarrow_\beta$ for β-reduction.

If x does not occur free in B, $(\Pi x{:}A.\ B)$ is written as $A \to B$. For $(s_1, s_2) \in \boldsymbol{R}$ we simplify the formation rule to

$$\frac{\Gamma \vdash A:s_1 \quad \Gamma \vdash B:s_2}{\Gamma \vdash A \to B:s_2}$$

if there are no terms A and B such that $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$ with x occurring free in B.

Many well-known type systems can be defined as PTS. For example :

Definition2 ($\lambda\to$, $\lambda 2$, $\lambda\omega$, λC).
Church's simply typed λ calculus is the PTS $\lambda\to$

$$\boldsymbol{S} = \{*, \Box\},\ \ \boldsymbol{A} = \{* : \Box\},\ \ \boldsymbol{R} = \{(*,*)\}$$

The second order typed lambda calculus (system F) is the PTS $\lambda 2$

$$\boldsymbol{S} = \{*, \Box\},\ \ \boldsymbol{A} = \{* : \Box\},\ \ \boldsymbol{R} = \{(*,*), (\Box,*)\}$$

Girard's system F_ω is the PTS $\lambda\omega$

$$\boldsymbol{S} = \{*, \Box\},\ \ \boldsymbol{A} = \{* : \Box\},\ \ \boldsymbol{R} = \{(*,*), (\Box,*), (\Box,\Box)\}$$

The Calculus of Constructions is the PTS λC

$$\boldsymbol{S} = \{*, \Box\},\ \ \boldsymbol{A} = \{* : \Box\},\ \ \boldsymbol{R} = \boldsymbol{S}^2$$

3 $\lambda\omega_L$

The system $\lambda\omega_L$ contains two copies of $\lambda\omega$. One is a programming language – terms are programs and types are data types –, the other is a logic – terms are proofs and types are propositions –. First we discuss these two interpretations of $\lambda\omega$. All sorts get a subscript s (for *set*) for the first interpretation, and a subscript p (for *p*roposition) for the second.

3.1 $\lambda\omega$ as a programming language : $\lambda\omega_s$

$\lambda\omega_s$ is the PTS

$$S = \{*_s, \Box_s\},\ \ A = \{*_s : \Box_s\},\ \ R = \{(*_s, *_s), (\Box_s, *_s), (\Box_s, \Box_s)\}$$

Three levels of expressions can be distinguished: kinds, constructors and programs.

The *kinds* are expressions κ such that $\Gamma \vdash \kappa : \Box_s$. The rule $(\Box_s, \Box_s)$ controls the formation of kinds:

$$(\Box_s, \Box_s)\ \ \frac{\Gamma \vdash \kappa_1 : \Box_s \quad \Gamma \vdash \kappa_2 : \Box_s}{\Gamma \vdash \kappa_1 \to \kappa_2 : \Box_s}$$

The kinds are generated by $\kappa ::= *_s \mid \kappa \to \kappa$. This includes $*_s$, the type of data types, $*_s \to *_s$, the type of functions from data types to data types, and $*_s \to *_s \to *_s$, the type of binary functions from data types to data types.

The *constructors* are expressions σ that have a kind as their type, i.e. $\Gamma \vdash \sigma : \kappa : \Box_s$ for some κ. This includes the data types (inhabitants of $*_s$), the functions from data types to data types, e.g. *list*, and the binary functions from data types to data types, e.g. $+$ and $\times$.

The *programs* are expressions m such that $\Gamma \vdash m : \sigma : *_s$ for some data type σ.

There are two rules for the formation of data types. These correspond to the two forms of abstraction in programs, which are

- abstraction over a data type, i.e. $(\lambda x{:}\sigma.\ \ldots)$ where $\sigma : *_s$.
- abstraction over a kind, i.e. $(\lambda \alpha : \kappa.\ \ldots)$ where $\kappa : \Box_s$. Since $*_s : \Box_s$, this includes abstraction over all data types, as for example in the polymorphic identity $(\lambda \alpha : *_s.\ \lambda x{:}\alpha.\ x)$.

The rule $(*_s, *_s)$ allows the formation of function types, used to type the first form of abstraction

$$(*_s, *_s)\ \ \frac{\Gamma \vdash \rho : *_s \quad \Gamma \vdash \sigma : *_s}{\Gamma \vdash \rho \to \sigma : *_s}$$

$\rho \to \sigma$ is the type of functions from ρ to σ.

The rule $(\Box_s, *_s)$ allows the formation of polymorphic data types, used to type the second form of abstraction

$$(\Box_s, *_s)\ \ \frac{\Gamma \vdash \kappa : \Box_s \quad \Gamma, \alpha : \kappa \vdash \sigma : *_s}{\Gamma \vdash (\Pi \alpha : \kappa.\ \sigma) : *_s}$$

For example, $(\Pi \alpha : *_s.\ \alpha \to \alpha)$ is a polymorphic data type (it is the type of the polymorphic identity).

Convention. $\kappa, \kappa_1, \kappa_2$ range over kinds; ρ, σ, τ range over constructors; α, β range over constructor-variables; m, n, f range over programs.

In $\lambda\omega$ there are representations of many data types such as booleans, integers, products, sums, lists and trees. In fact, all free term algebras can be represented (see [BB85]).

3.2 $\lambda\omega$ as a logic : $\lambda\omega_p$

$\lambda\omega_p$ is the PTS

$$\boldsymbol{S} = \{*_p, \Box_p\},\quad \boldsymbol{A} = \{*_p : \Box_p\},\quad \boldsymbol{R} = \{(*_p, *_p), (\Box_p, *_p), (\Box_p, \Box_p)\}$$

The three levels of expressions are called prop-kinds, prop-constructors and proofs.

The *prop-kinds* are expressions Φ such that $\Gamma \vdash \Phi : \Box_p$. The rule $(\Box_p, \Box_p)$ controls the formation of prop-kinds:

$$(\Box_p, \Box_p)\ \frac{\Gamma \vdash \Phi_1 : \Box_p \qquad \Gamma \vdash \Phi_2 : \Box_p}{\Gamma \vdash \Phi_1 \to \Phi_2 : \Box_p}$$

This means the prop-kinds are generated by $\Phi ::= *_p \mid \Phi \to \Phi$. This includes $*_p$, the type of propositions, $*_p \to *_p$, the type of functions from propositions to propositions, and $*_p \to *_p \to *_p$, the type of binary functions from propositions to propositions.

The *prop-constructors* are expressions ϕ that have a prop-kind as their type, i.e. $\Gamma \vdash \phi : \Phi : \Box_p$ for some Φ. This includes the propositions (inhabitants of $*_p$), the functions from propositions to propositions, e.g. $\neg$, and the binary functions from propositions to propositions, e.g. $\vee$ and $\wedge$.

The *proofs* are expressions H such that $\Gamma \vdash H : \phi : *_p$ for some proposition ϕ.

There are two rules for the formation of propositions. The rule $(*_p, *_p)$ allows the formation of implication

$$(*_p, *_p)\ \frac{\Gamma \vdash \phi : *_p \quad \Gamma \vdash \psi : *_p}{\Gamma \vdash \phi \Rightarrow \psi : *_p}$$

The rule $(\Box_p, *_p)$ allows higher order quantification

$$(\Box_p, *_p)\ \frac{\Gamma \vdash \Phi : \Box_p \quad \Gamma, \alpha : \Phi \vdash \phi : *_p}{\Gamma \vdash (\forall \alpha{:}\Phi.\ \phi) : *_p}$$

$\Rightarrow$ and $\forall$ are just other notations for $\to$ and Π. Since $*_p : \Box_p$, the last rule allows universal quantification over all propositions (second order logic). This makes it possible to express for example an introduction rule of $\vee$ inside the system, viz. by the proposition $(\forall \phi{:}*_p.\ \forall \psi{:}*_p.\ \phi \Rightarrow (\phi \vee \psi))$.

Convention. Φ, Φ_1, Φ_2 range over prop-kinds; ϕ, ψ range over prop-constructors; H, H_f range over proofs.

Conjunction and disjunction can be defined in $\lambda\omega_p$. These encodings are exactly those of $+$ and $\times$ in $\lambda\omega_s$. Also, we can define $True =_{def} (\forall \phi{:}*_p.\ \phi \Rightarrow \phi)$, $False =_{def} (\forall \phi{:}*_p.\ \phi)$, and $\neg =_{def} (\lambda \phi{:}*_p.\ \phi \Rightarrow False) : *_p \to *_p$. Then $True$ is provable, as is $(\forall \phi{:}*_p.\ False \Rightarrow \phi)$. There is no reason to restrict ourselves to intuitionistic logic, so the axiom $dbng : (\forall \phi{:}*_p.\ (\neg\neg\phi) \Rightarrow \phi)$ can be assumed.

3.3 $\lambda\omega_L$

The programming logic $\lambda\omega_L$ consists of a programming language – $\lambda\omega_s$ – and a logic, in which properties of programs can be expressed. $\lambda\omega_p$ is part of the logic. $\lambda\omega_p$ alone does not suffice as logic; some PTS-rules have to be added.

To allow universal quantification over a data type and over a kind in propositions, the rules $(*_s, *_p)$ and $(\Box_s, *_p)$ are added.

$$(*_s, *_p)\ \frac{\Gamma \vdash \sigma : *_s \quad \Gamma, x : \sigma \vdash \phi : *_p}{\Gamma \vdash (\forall x{:}\sigma.\ \phi) : *_p}$$

$$(\Box_s, *_p)\ \frac{\Gamma \vdash \kappa : \Box_s \quad \Gamma, \alpha : \kappa \vdash \phi : *_p}{\Gamma \vdash (\forall \alpha{:}\kappa.\ \phi) : *_p}$$

Since $*_s : \Box_s$, the second rule allows for instance quantification over all data types, i.e. $(\forall \alpha{:}*_s.\ \ldots)$

The type of predicates on σ is $\sigma \to *_p$, i.e. predicates are propositional valued functions. For example, the ordering $\leq$ on natural numbers is a binary predicate on the type *nat*, i.e. $\leq : nat \to nat \to *_p$. Equality is a polymorphic binary predicate on data types, $= : (\Pi \alpha{:}*_s.\ \alpha \to \alpha \to *_p)$.

$\sigma \to *_p$, $\sigma \to \sigma \to *_p$ and $(\Pi \alpha{:}*_s.\ \alpha \to \alpha \to *_p)$ are all prop-kinds, i.e. they have type $\Box_p$. The formation of the first two requires the rule

$$(*_s, \Box_p)\ \frac{\Gamma \vdash \sigma : *_s \quad \Gamma \vdash \Phi : \Box_p}{\Gamma \vdash \sigma \to \Phi : \Box_p}$$

The formation of $(\Pi \alpha{:}*_s.\ \alpha \to \alpha \to *_p)$ also requires the rule

$$(\Box_s, \Box_p)\ \frac{\Gamma \vdash \kappa : \Box_s \quad \Gamma, \alpha : \kappa \vdash \Phi : \Box_p}{\Gamma \vdash (\Pi \alpha{:}\kappa.\ \Phi) : \Box_p}$$

The standard second-order logic definitions can now be used to define existential quantifications $(\exists x : \sigma.\ \phi)$ and $(\exists \alpha : *_s.\ \phi)$ in terms of implication and (second order) quantification over propositions. Inductive definitions can be coded as in the Calculus of Constructions (see [PPM90]). For example, Leibniz equality, $=_L : (\Pi \alpha{:}*_s.\ \alpha \to \alpha \to *_p)$, can be defined as

$$\lambda \alpha{:}*_s.\ \lambda x, y{:}\alpha.\ (\forall \phi{:}\alpha \to *_p.\ (\phi x) \Rightarrow (\phi y))$$

$=_L$ will be written infix: instead of $(=_L\ \sigma\ m\ n)$ we write $(m =_\sigma n)$.

We have now discussed all the rules of $\lambda\omega_L$:

Definition 3. $\lambda\omega_L$ is PTS

$$\begin{aligned}
\boldsymbol{S} &= \{*_s, \Box_s, *_p, \Box_p\} \\
\boldsymbol{A} &= \{(*_s : \Box_s)\ ,\ (*_p : \Box_p)\} \\
\boldsymbol{R} &= \{(\Box_s, \Box_s), \\
&\quad (\Box_s, *_s),\ (*_s, *_s), \\
&\quad (\Box_s, \Box_p), (*_s, \Box_p),\ (\Box_p, \Box_p), \\
&\quad (\Box_s, *_p),\ (*_s, *_p),\ (\Box_p, *_p),\ (*_p, *_p)\ \}
\end{aligned}$$

So $\lambda\omega_L$ consists of

- $\lambda\omega_s$ for programs and their types: $\{(\Box_s, \Box_s), (\Box_s, *_s), (*_s, *_s)\}$
- $\lambda\omega_p$ for the propositions and their proofs: $\{(\Box_p, \Box_p), (\Box_p, *_p), (*_p, *_p)\}$
- all possible dependencies of propositions and proofs on programs and types: $\{(\Box_s, \Box_p), (\Box_s, *_s), (*_s, \Box_p), (*_s, *_p)\}$

$\lambda\omega_L$ does not have all possible rules. It does not have the rules

- $(*_s, \Box_s)$ – the program/term dependent types.
  This is because we have chosen $\lambda\omega$ and not λC as programming language.
- $(*_p, \Box_p)$ – the proof dependent propositions.
  We are only interested in proving properties of programs (i.e. m with $m : \sigma : *_s$) and not in proving properties of proofs (i.e. H with $H : \phi : *_p$), so there is no need for proof dependent propositions.
- $(?_p, ?_s)$ – the proposition/proof dependent programs/types.

Because we do not have any rules of the form $(?_p, ?_s)$, $\lambda\omega_L$ is a conservative extension of the programming language:

Lemma 4. *Suppose* $\Gamma \vdash_{\lambda\omega_L} a : A$ *with* $\Gamma \vdash_{\lambda\omega_L} A : *_s$, $\Gamma \vdash_{\lambda\omega_L} A : \Box_s$, *or* $A \equiv \Box_s$. *Then* $|\Gamma|_s \vdash_{\lambda\omega_S} a : A$,
where $|\epsilon|_s = \epsilon$, *and* $|\Gamma, x : A|_s = \begin{cases} |\Gamma|_s, x : A & \text{if } \Gamma \vdash A : *_s \text{ or } \Gamma \vdash A : \Box_s \\ |\Gamma|_s & \text{if } \Gamma \vdash A : *_p \text{ or } \Gamma \vdash A : \Box_p \end{cases}$

Proof. Induction on the derivation of $\Gamma \vdash a : A$.

If we forget about the subscripts p and s, it is a subsystem of λC:

Lemma 5. *If* $\Gamma \vdash_{\lambda\omega_L} a : A$ *then* $|\Gamma| \vdash_{\lambda C} |a| : |A|$,
where $|z|$ *is* z *with* $*$ *substituted for* $*_s$ *and* $*_p$, *and* $\Box$ *substituted for* $\Box_s$ *and* $\Box_p$.

Proof. Trivial.

So all $\lambda\omega_L$-terms are strongly normalising, and that $\lambda\omega_L$ is consistent, in the sense that not all propositions are provable ($False$, defined as $(\forall \phi : *_p.\ \phi)$, is not provable).

4 Program and proof development in $\lambda\omega_L$

$\lambda\omega_s$ programs and types can be annotated in $\lambda\omega_L$. For a $\lambda\omega_s$-program this annotation is a proof that the program satisfies a certain property. This property is a predicate on the type of the program. A $\lambda\omega_s$-type is annotated with a predicate on that type (a specification).

$\lambda\omega_s$	$\lambda\omega_L$ annotation
type σ, $\Gamma \vdash \sigma : *_s$	predicate P, $\Gamma \vdash P : \sigma \to *_p$
program m, $\Gamma \vdash m : \sigma : *_s$	proof H_m, $\Gamma \vdash H_m : (P\ m) : *_p$

The task of finding a program that satisfies a given specification is the following:
given : a *type* σ, $\Gamma \vdash \sigma : *_s$, and a *specification* P, $\Gamma \vdash P : \sigma \to *_p$
find : a *program* m, $\Gamma \vdash m : \sigma$, and a *proof* H_m, $\Gamma \vdash H_m : (P\ m)$.

For every derivation rule in $\lambda\omega_s$, there is a corresponding derivation rule in $\lambda\omega_L$ dealing with these annotations.

$\lambda\omega_s$-type derivation rules give the type of a program in terms of the types of its component parts. The corresponding $\lambda\omega_L$-derivation rules give a proof that this program satisfies a specification in terms of proofs that its component parts satisfy certain specifications. These rules can be used to develop a program together with its correctness proof.

The specification may have to be of a certain form in order to apply the corresponding $\lambda\omega_L$-rule. For $\lambda\omega_s$-rules for forming a type there is a corresponding rule in $\lambda\omega_L$ for forming specifications on that type, which yields specifications of the right form.

4.1 Function types

First we investigate these rules for function types and the associated forms of abstraction and application. To distinguish programming language and logic $\lambda\omega_s$-judgements are written in boldface.

Abstraction. Suppose we want a program f and a proof H_f such that

$$\boldsymbol{\Gamma \vdash f : \rho \to \sigma} \qquad \Gamma \vdash H_f : (\forall x{:}\rho.\ P\,x \Rightarrow Q\,x\,(fx))$$

where $\Gamma \vdash \rho \to \sigma : *_s$, $\Gamma \vdash P : \rho \to *_p$ and $\Gamma \vdash Q : \rho \to \sigma \to *_p$.

We can try a program of the form $(\lambda x{:}\rho.\ m)$ for some m. We then have to find a term m and a proof H_m such that

$$\text{(i) } \boldsymbol{\Gamma, x : \rho \vdash m : \sigma} \qquad \text{(ii) } \Gamma, x : \rho, H_x : P\,x \vdash H_m : Q\,x\,m$$

From (i) it then follows that $(\lambda x{:}\rho.\ m)$ has the right type and from (ii) it follows that it satisfies the specification

$$\begin{aligned} &\boldsymbol{\Gamma \vdash (\lambda x : \rho.\ m) : \rho \to \sigma} \\ &\Gamma \vdash (\lambda x{:}\rho.\ \lambda H_x{:}P\,x.\ H_m) \ : \ (\forall x{:}\rho.\ P\,x \Rightarrow Q\,x\,m) \\ &\qquad\qquad\qquad\qquad \simeq (\forall x{:}\rho.\ P\,x \Rightarrow Q\,x\,((\lambda x{:}\rho.\ m)x)) \end{aligned}$$

Application. Suppose we want a program m and a proof H_m such that

$$\boldsymbol{\Gamma \vdash m : \sigma} \qquad \Gamma \vdash H_m : Q'm$$

where $\Gamma \vdash \sigma : *_s$ and $\Gamma \vdash Q' : \sigma \to *_p$.

We can try a program of the form fn, with n possibly occurring in the specification Q'. Then for some $\rho : *_s$, $P : \rho \to *_p$ and $Q : \rho \to \sigma \to *_p$ such that $(Q\,n) \simeq Q'$ we have to find a program n with a proof H_n such that

$$\text{(i) } \boldsymbol{\Gamma \vdash n : \rho} \qquad \text{(ii) } \Gamma \vdash H_n : P\,n$$

and a program f with a proof H_f such that

$$\text{(iii) } \boldsymbol{\Gamma \vdash f : \rho \to \sigma} \qquad \text{(iv) } \Gamma \vdash H_f : (\forall x{:}\rho.\ P\,x \Rightarrow Q\,x\,(fx))$$

From (i) and (iii) it follows that fn has the right type, and from (ii) and (iv) it follows that fn satisfies the specification Q:

$$\Gamma \vdash \boldsymbol{fn} : \sigma \qquad \Gamma \vdash H_f n H_n \ : \ (Q\, n\, (fn)) \ \simeq \ (Q'(fn))$$

The examples above lead to the derived $\lambda\omega_L$-rules rules for function types and the associated forms of application and abstraction given below. For the formation of function types

$$\frac{\Gamma \vdash \rho : *_s \qquad \Gamma, x : \rho \vdash \sigma : *_s}{\Gamma \vdash \rho \to \sigma : *_s} \qquad \frac{\Gamma \vdash P : \rho \to *_p \qquad \Gamma, x : \rho \vdash Q' : \sigma \to *_p}{\Gamma \vdash (\lambda f{:}\rho \to \sigma.\ \forall x{:}\rho.\ P\, x \Rightarrow Q'(fx)) \ : (\rho \to \sigma) \to *_p}$$

Note that x may occur in Q', whereas x cannot occur in σ. This is made more explicit by using a predicate Q such that $\Gamma \vdash Q : \rho \to \sigma \to *_p$ instead of a predicate Q' such that $\Gamma, x : \rho \vdash Q' : \sigma \to *_p$, and replacing Q' by $(Q\, x)$.

For $\Gamma \vdash \rho : *_s, \sigma : *_s$ and $\Gamma \vdash P : \rho \to *_p, Q : \rho \to \sigma \to *_p$ we have the following coupled derivation rules

$$\frac{\Gamma, x : \rho \vdash m : \sigma}{\Gamma \vdash (\lambda x{:}\rho.\ m) : \rho \to \sigma} \qquad \frac{\Gamma, x : \rho, H_x : P\, x \vdash H_m : Q\, x\, m}{\Gamma \vdash (\lambda x{:}\rho.\ \lambda H_x{:}P\, x.\ H_m) \ : \ (\forall x{:}\rho.\ P\, x \Rightarrow Q\, x\, ((\lambda x{:}\rho.\ m)x))}$$

$$\frac{\Gamma \vdash f : \rho \to \sigma \qquad \Gamma \vdash n : \rho}{\Gamma \vdash fn : \sigma} \qquad \frac{\Gamma \vdash H_f : (\forall x{:}\rho.\ P\, x \Rightarrow Q\, x\, (fx)) \qquad \Gamma \vdash H_n : P\, n}{\Gamma \vdash H_f n H_n : Q\, n\, (fn)}$$

All these rules are derivable in $\lambda\omega_L$.

4.2 Polymorphic types

In addition to function types of the form $\rho \to \sigma$ we also have polymorphic types of the form $(\Pi\alpha{:}\kappa.\ \sigma)$, with $\kappa : \Box_s$. For the $\lambda\omega_s$ derivation rules for these polymorphic types we can also give corresponding $\lambda\omega_L$ rules:

$$\frac{\Gamma, \alpha : \kappa \vdash \sigma : *_s}{\Gamma \vdash (\Pi\alpha{:}\kappa.\ \sigma) : *_s} \qquad \frac{\Gamma, \alpha : \kappa \vdash P : \sigma \to *_p}{\Gamma \vdash (\lambda f{:}(\Pi\alpha{:}\kappa.\ \sigma).\ \forall\alpha{:}\kappa.\ P(f\alpha)) \ : \ (\Pi\alpha{:}\kappa.\ \sigma) \to *_p}$$

$$\frac{\Gamma, \alpha : \kappa \vdash m : \sigma}{\Gamma \vdash (\lambda\alpha{:}\kappa.\ m) : (\Pi\alpha{:}\kappa.\ \sigma)} \qquad \frac{\Gamma, \alpha : \kappa \vdash H_m : (P\, m)}{\Gamma \vdash (\lambda\alpha{:}\kappa.\ H_m) : \ (\forall\alpha{:}\kappa.\ P\, m) \simeq (\forall\alpha{:}\kappa.\ P((\lambda\alpha{:}\kappa.\ m)\alpha))}$$

$$\frac{\Gamma \vdash \rho : \kappa \qquad \Gamma \vdash f : (\Pi\alpha{:}\kappa.\ \sigma)}{\Gamma \vdash f\rho : \sigma[\alpha := \rho]} \qquad \frac{\Gamma \vdash H_f : (\forall\alpha{:}\kappa.\ P(f\alpha))}{\Gamma \vdash H_f \rho : P[\alpha := \rho](f\rho)}$$

5 Some examples

We need larger building blocks for programs than just abstraction and application. To prove that such programs are correct, corresponding proof rules are then needed. As examples, we consider the booleans and natural numbers. These can be represented in $\lambda\omega_s$ (see for example [BB85]). What these representations are is are not important here. We only have to know there are terms

- $bool : *_s, true : bool, false : bool$ and $if : (\Pi\alpha{:}*_s.\ bool \to \alpha \to \alpha \to \alpha)$ such that $(if\ \alpha\ true\ m\ n) \to\!\!\!\!\to_\beta m$ and $(if\ \alpha\ false\ m\ n) \to\!\!\!\!\to_\beta n$.
  For $(if\ \alpha\ b\ m\ n)$ we write $if\ b\ then\ m\ else\ n$.
- $nat : *_s$, $0 : nat$, $S : nat \to nat$ and $iter : (\Pi\alpha{:}*_s.\ \alpha \to (\alpha \to \alpha) \to (nat \to \alpha))$ such that $(iter\ \alpha\ m\ f\ 0) \to\!\!\!\!\to_\beta a$ and $(iter\ \alpha\ m\ f\ (S\ n)) \to\!\!\!\!\to_\beta f(iter\ \alpha\ m\ f\ n)$.
  Using $iter$ iterative functions on the natural numbers can be defined : $(iter\ \alpha\ m\ f)$ is the function mapping $(S^n 0)$ to $(f^n m)$.
- $+ : *_s \to *_s \to *_s$, $in_i : (\Pi\alpha_1, \alpha_2{:}*_s.\ \alpha_i \to (\alpha_1 + \alpha_2))$ for $i \in \{1, 2\}$ and $out : (\Pi\alpha_1, \alpha_2, \gamma{:}*_s.\ (\alpha_1 \to \gamma) \to (\alpha_2 \to \gamma) \to (\alpha_1 + \alpha_2) \to \gamma)$ such that $(out\alpha_1\alpha_2\gamma f_1 f_2(in_i\alpha_1\alpha_2 a)) \to\!\!\!\!\to_\beta (f_i a)$. For $(out\alpha_1\alpha_2 f_1 f_2)$ we write $[f_1, f_2]$.

Because $\lambda\omega_s$ is strongly normalizing, we may assume the induction principles for these types:

$$\forall\phi{:}bool \to *_p.\ (\phi\ true) \Rightarrow\ (\phi\ false) \Rightarrow\ (\forall b{:}bool.\ (\phi\ b))$$

$$\forall\phi{:}nat \to *_p.\ (\phi\ 0)\ \Rightarrow (\forall x{:}nat.\ (\phi\ x) \Rightarrow (\phi\ (S\ x)))\ \Rightarrow (\forall n{:}nat.\ \ (\phi\ n))$$

$$\forall\alpha, \beta{:}*_s.\ \forall\phi{:}(\alpha + \beta) \to *_p.$$
$$\forall a{:}\alpha.\ \phi(in_1\ \alpha\ \beta\ a) \Rightarrow\ \forall b{:}\beta.\ \phi(in_2\ \alpha\ \beta\ b) \Rightarrow\ (\forall x{:}\alpha + \beta.\ (\phi\ x))$$

For $if\ then\ else$, $iter$ and out proof rules corresponding with the type derivation rules can be found. There are proofs H_{if}, H_{iter} and H_{out} such that

$$\frac{\begin{array}{l}\Gamma \vdash \sigma : *_s \\ \Gamma \vdash b : bool \\ \Gamma \vdash m : \sigma \\ \Gamma \vdash n : \sigma\end{array} \qquad \Gamma \vdash P : bool \to \sigma \to *_p \qquad \begin{array}{l}\Gamma \vdash H_m : (P\ true\ m) \\ \Gamma \vdash H_n : (P\ false\ n)\end{array}}{\begin{array}{l}\Gamma \vdash (H_{if}\ \sigma\ P\ m\ H_m\ n\ H_n\ b)\ : \\ \qquad\qquad P\ b\ (if\ b\ then\ m\ else\ n)\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \sigma : *_s \\ \Gamma \vdash m : \sigma \\ \Gamma \vdash f : \sigma \to \sigma\end{array} \qquad \begin{array}{l}\Gamma \vdash P : nat \to \sigma \to *_p \\ \Gamma \vdash H_m : (P\ 0\ m) \\ \Gamma \vdash H_f : (\forall y{:}\sigma.\ \forall n{:}nat.\ (P\ n\ y) \Rightarrow (P(S\ n)(f\ y)))\end{array}}{\begin{array}{l}\Gamma \vdash (H_{iter}\sigma\ P\ m\ H_m\ f\ H_f)\ : \\ \qquad (\forall n{:}nat.\ \ (P\ n\ (iter\ \sigma\ m\ f\ n)))\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \rho + \sigma : *_s\ ,\ \tau : *_s \\ \Gamma \vdash f : \rho \to \tau \\ \Gamma \vdash g : \sigma \to \tau\end{array} \qquad \begin{array}{l}\Gamma \vdash P : (\rho + \sigma) \to \tau \to *_p \\ \Gamma \vdash H_f : (\forall x{:}\rho.\ (P(in_1\rho\ \sigma\ x)(f\ x))) \\ \Gamma \vdash H_g : (\forall x{:}\sigma.\ (P(in_2\rho\ \sigma\ x)(g\ x)))\end{array}}{\Gamma \vdash (H_{out}\ \rho\ \sigma\ \tau\ P\ f\ H_f\ g\ H_g)\ : (\forall x{:}\rho + \sigma.\ P([f, g]\ x))}$$

So H_{if} is a proof of the following proposition:

$$\begin{array}{l}\forall\alpha{:}*_s.\ \forall P{:}\mathit{bool}\to\alpha\to *_p.\\ \quad\forall m{:}\alpha.\ (P\ \mathit{true}\ m)\Rightarrow\\ \quad\forall n{:}\alpha.\ (P\ \mathit{false}\ n)\Rightarrow\\ \quad\forall b{:}\mathit{bool}.\ (P\ b\ (\mathit{if}\ b\ \mathit{then}\ m\ \mathit{else}\ n))\end{array}$$

The proof terms are getting very long. Fortunately, we can adopt the viewpoint of classical logic: it is important to know that a proposition is true, and not what a proof of the proposition is. Proofs terms could be omitted, abbreviating for example $\Gamma, H_1 : \phi \vdash H_2 : \psi$ to $\Gamma, \phi \vdash \psi$.

Other, more specific, proof rules can be derived. For example, for *if then else*

$$\frac{\begin{array}{l}\Gamma\vdash\sigma:*_s\\ \Gamma\vdash b:\mathit{bool}\\ \Gamma\vdash m:\sigma\\ \Gamma\vdash n:\sigma\end{array}}{\Gamma\vdash(\mathit{if}\ b\ \mathit{then}\ m\ \mathit{else}\ n):\sigma}\qquad\frac{\begin{array}{l}\Gamma\vdash Q:\sigma\to *_p\\ \\ \Gamma,H:(b=_{\mathit{bool}}\mathit{true})\vdash H_m:(Q\,m)\\ \Gamma,H:(b=_{\mathit{bool}}\mathit{false})\vdash H_n:(Q\,n)\end{array}}{\Gamma\vdash\ldots:Q\,(\mathit{if}\ b\ \mathit{then}\ m\ \mathit{else}\ n)}$$

A possible proof of $Q\,(\mathit{if}\ b\ \mathit{then}\ m\ \mathit{else}\ n)$ is
$(H_{if}\sigma P m(\lambda H{:}\ldots.\ H_m)n(\lambda H{:}\ldots.\ H_n)bH_{refl})$, where $P : \mathit{bool}\to\sigma\to *_p$ is $(\lambda x{:}\mathit{bool}.\lambda y{:}\sigma.\ (x =_{\mathit{bool}} b)\Rightarrow(Q\,y))$ and H_{refl} is a proof of $b =_{\mathit{bool}} b$.

A test for zero is a degenerated case of *iter*. We write $\mathit{ifzero}\ n\ \mathit{then}\ m_1\ \mathit{else}\ m_2$ for $(\mathit{iter}\ \sigma\ m_1(\lambda x{:}\sigma.\ m_2)\ n)$ if x does not occur free in m_2. For *ifzero* we have the following rules

$$\frac{\begin{array}{l}\Gamma\vdash\sigma:*_s\\ \Gamma\vdash n:\mathit{nat}\\ \Gamma\vdash m_1:\sigma\\ \Gamma\vdash m_2:\sigma\end{array}}{\Gamma\vdash(\mathit{ifzero}\ n\ \mathit{then}\ m_1\ \mathit{else}\ m_2):\sigma}\qquad\frac{\begin{array}{l}\Gamma\vdash Q:\sigma\to *_p\\ \\ \Gamma,H:(n=_{\mathit{nat}}0)\vdash H_{m1}:(Q\,m_1)\\ \Gamma,H:(n\neq_{\mathit{nat}}0)\vdash H_{m2}:(Q\,m_2)\end{array}}{\Gamma\vdash\ldots:Q\,(\mathit{ifzero}\ n\ \mathit{then}\ m_1\ \mathit{else}\ m_2)}$$

A possible proof for $Q\,(\mathit{ifzero}\ n\ \mathit{then}\ m_1\ \mathit{else}\ m_2)$ is
$(H_{\mathit{iter}}\sigma P(\lambda H{:}\ldots H_{m1})$
$(\lambda y{:}\sigma.\lambda x{:}\mathit{nat}.\lambda H_2{:}(P\,x\,y).\lambda H_3{:}((S\,x)=_{\mathit{nat}}n).((\lambda H{:}\ldots H_{m2})(H_{nz}\,H_3)))$
$n H_{refl})$
where $P : \mathit{nat}\to\sigma\to *_p$ is $\lambda x{:}\mathit{nat}.\ \lambda y{:}\sigma.\ (x =_{\mathit{nat}} n)\Rightarrow(Qy)$, H_{nz} is a proof of $((Sx)=_{\mathit{nat}}n)\Rightarrow(n\neq_{\mathit{nat}}0)$, and H_{refl} is a proof of $n =_{\mathit{nat}} n$.

To illustrate how these rules can be used, we consider the construction of a program that computes n *modulo* 2. We define $+ =_{def} \lambda n{:}\mathit{nat}.\ (\mathit{iter}\ \mathit{nat}\ n\ S)$, and $\mathit{MOD} =_{def} \lambda n, m{:}\mathit{nat}.\ (\exists d{:}\mathit{nat}.\ d+d+m=_{\mathit{nat}}n)\wedge(m=_{\mathit{nat}}0\vee m=_{\mathit{nat}}=1)$.

We want a program *mod* and a proof H_{mod} such that

$$\vdash \mathit{mod} : \mathit{nat}\to\mathit{nat},\quad H_{\mathit{mod}} : (\forall n{:}\mathit{nat}.\ (\mathit{MOD}\,n\,(\mathit{mod}\,n)))$$

We choose $\mathit{mod}\equiv(\mathit{iter}\ \mathit{nat}\ m_0\ m_S)$. We then have to find programs m_0 and m_S and proofs H_0 and H_S such that

$$\vdash m_0 : \mathit{nat},$$

$$
\begin{array}{l}
\quad H_0 : (MOD\,0\,m_0) \\
\vdash \boldsymbol{m_S : nat \to nat}, \\
\quad H_S : (\forall m_n : nat.\ \forall n : nat.\ (MOD\,n\,m_n) \Rightarrow (MOD\,(S\,n)\,(m_S\,m_n)))
\end{array}
$$

For $m_0 \equiv 0$, $(MOD\ 0\ m_0)$ can be proven. This leaves m_S: we choose $m_S \equiv (\lambda m_n : nat.\ m_{Sn})$. We then want a program m_{Sn} and a proof H_{Sn} s.t.

$$
\begin{array}{r}
\boldsymbol{m_n : nat \vdash m_{Sn} : nat} \\
m_n : nat, n : nat, H_{mn} : (MOD\,n\,m_n) \vdash H_{Sn} : MOD\,(S\,n)\,m_{Sn}
\end{array}
$$

We choose $m_{Sn} \equiv ifzero\ m_n\ then\ m_{odd}\ else\ m_{even}$. We then have to find m_{odd}, m_{even}, H_{odd} and H_{even} such that

$$
\begin{array}{r}
\boldsymbol{m_n : nat \vdash m_{odd} : nat} \\
m_n : nat, n : nat, H_{mn} : (MOD\,n\,m_n), H : (m_n =_{nat} 0) \vdash H_{odd} : MOD\,(S\,n)\,m_{odd} \\
\boldsymbol{m_n : nat \vdash m_{even} : nat} \\
m_n : nat, n : nat, H_{mn} : (MOD\,n\,m_n), H : (m_n \neq_{nat} 0) \vdash H_{even} : MOD\,(S\,n)\,m_{even}
\end{array}
$$

Finally, for $m_{odd} \equiv 1$ and $m_{even} \equiv 0$ there are such proofs H_{odd} and H_{even}. So the resulting program is $(iter\ nat\ 0\ (\lambda m_n : nat.\ ifzero\ m_n\ then\ 1\ else\ 0))$.

6 Related work

Although our approach is quite different from the one described in [PM89], it is closely related. Instead of constructing a proof in the Calculus of Constructions, and then extracting an F_ω-program and a correctness proof, we directly construct an F_ω program and a correctness proof. In the introduction we already motivated this choice.

In [BM90] a similar approach to program construction is considered. There ECC, the Extended Calculus of Constructions (see [Luo89]), is used as an external programming logic, although there programming and logic language are not separated as in $\lambda\omega_L$. A difference is that the programs live at the predicative level of ECC (i.e. the data types have type $\Box$), whereas our programs live at the impredicative level of Calculus of Constructions (i.e. the data types have type $*$). In [BM90] strong sums are used to form pairs of programs and proofs – called deliverables – and pairs of types and specifications, which can then be manipulated inside the system. However, a disadvantage of this is the restrictions it imposes on the form of programs and specifications. On a type $\rho \to \sigma$ only specifications of the form $\lambda f : \rho \to \sigma.\ \forall x : \rho.\ (P\,x) \Rightarrow (Q\,(f\,x))$ with x not occurring in Q are allowed; to express a relationship between the input and output of a function so-called second order deliverables are required.

Dybjer has studied the use of LTC as a programming logic (see [Dyb90]). Like $\lambda\omega_L$, LTC contains a logic and a separate programming language, but both the logic and programming language are different from ours. The programming language is the untyped lambda calculus, so type information must be expressed in the logic.

7 Conclusions and directions for future research

$\lambda\omega_L$ is a small, homogeneous, system, which can be defined as a single PTS. This means the same algorithms can be used for both (data) type checking and proof checking. We avoid the use of subset types and the need for program extraction.

As illustrated in section 5, programs and their correctness proofs can be constructed together, and proof rules for derived constructs can be obtained in a simple manner. Here it is convenient that higher order quantifications are allowed: the proof rules can be proven correct inside the system.

The rigorous separation between programming language and logic makes it easier to extend the programming language without disturbing the logic, and vice versa. For example, a useful extension of the logic is the axiom $dbng : (\forall \phi{:}*_p.\neg\neg\phi \Rightarrow \phi)$.

Many extensions of $\lambda\omega$ have been proposed, that are useful for $\lambda\omega$ as a programming language, for example subtyping, bounded quantification, records, and recursive types (see for instance [CW85] [Car89] [CM89]).

One useful extension of the programming language is a fixed-point combinator $Y : (\Pi\alpha{:}*_s.\ (\alpha \to \alpha) \to \alpha)$. Programs are then no longer guaranteed to terminate, so we have to be able to reason about termination in the programming logic. To do this the programming logic could be extended in the style of LCF with an ordering $\sqsubseteq : (\Pi\alpha{:}*_s.\ \alpha \to \alpha \to *_p)$, a constant $\bot : (\Pi\alpha{:}*_s.\ \alpha)$ and the associated axioms, including the fixed-point induction rule.

To justify these extensions and prove consistency, a model for the programming language is needed in which data types are interpreted as cpos. Examples of such models are [BM92] and [PHtE99].

Other type constructors such as $+$ and $\times$, and recursive types $(\mu\alpha{:}*_s.\ \sigma)$ can be interpreted directly by the corresponding domain theoretic notions in a $\lambda\omega_L$ model based on [PHtE99]. We hope to discuss this in a forthcoming paper.

References

[Bar92] H. P. Barendregt. Typed lambda calculi. In D. M. Gabbai, S. Abramsky, and T. S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1992.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic Synthesis of Typed λ-Programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BM90] Rod Burstall and James McKinna. Deliverables : an approach to program development in the calculus of constructions. In *Procs. of the first Workshop on Logical Frameworks*, pages 113–121, 1990.

[BM92] Kim B. Bruce and John C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Principles of Programming Languages*, pages 316–327. ACM SIGACT-SIGPLAN, 1992.

[Car89] Luca Cardelli. Typeful programming. Technical Report 45, Digital SRC, 1989.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In M. Main et al, editor, *Fifth International Conference on Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*, pages 22–53, 1989.

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

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[Dyb90] Peter Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79, 1990.

[Luo89] Z. Luo. ECC, the Extended Calculus of Constructions. In *Logic in Computer Science*, pages 386–395. IEEE, 1989.

[ML79] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science*, volume VI, pages 153–175. North Holland, 1979.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Lang. and Syst.*, 10(3):470–502, 1988.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory : an introduction.* Oxford Science Publications, 1990.

[PHtE99] Erik Poll, Kees Hemerik, and Huub ten Eikelder. Cpo-models for second order lambda calculus with recursive types and subtyping. 199+. to appear in Theoretical Informatics.

[PL89] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In F.Orejas J.Diaz, editor, *TAPSOFT'89*, LNCS, pages 345–359. Springer, 1989.

[PM89] Christine Paulin-Mohring. Extracting Fω Programs from Proofs in the Calculus of Constructions. In *Principles of Programming Languages*, pages 89–104. ACM, 1989.

[PPM90] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In *Mathematical Foundations of Programming Languages*, volume 442 of *LNCS*. Springer, 1990.

[Rey85] John C. Reynolds. Three Approaches to Type Structure. In Ehrig et al., editor, *Mathematical Foundations of Software Development*, LNCS, pages 97–138. Springer, 1985.

o

In this series appeared:

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.

91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.

92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.