A programming logic for $F_\omega$

Poll, E.

Published: 01/01/1992

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Dec. 2018
A programming logic for $F_\omega$

by

Erik Poll 92/25
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

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M. van Hee.
A programming logic for $F_\omega$

Erik Poll*

Abstract. A programming logic is defined in which $F_\omega$ 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 - $\lambda$-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 $\lambda$-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 $\forall x : \rho. \text{pre}(x) \Rightarrow \exists y : \sigma. \text{post}(x, y)$ contains an algorithm which, given a term $x$ and a proof $H_x$ of $\text{pre}(x)$, returns a term $y$ and a proof $H_y$ of $\text{post}(x, y)$. 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 $\forall x : \rho. \text{pre}(x) \Rightarrow \exists y : \sigma. \text{post}(x, y)$ an $F\omega$ program $f$ of type $\rho \rightarrow \sigma$ and a proof $H_f$ of $\forall x : \rho. \text{pre}(x) \Rightarrow \text{post}(x, f(x))$ 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 $\lambda \omega$, Girard's system $F\omega$. 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 $\lambda$-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 $\lambda \omega$, 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. $\lambda \omega$-programs can be annotated in $\lambda \omega_L$. 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 $\sigma$, this annotation consists of a proof $H_f$ that $f$ satisfies some property $\phi$. This property (or specification) $\phi$ is a predicate on the type $\sigma$, and $H_f$ is a proof of the proposition $\langle \phi, f \rangle$. For example, for a program $f$ of type $\texttt{int/ist} \rightarrow \texttt{int/ist}$, the annotation could be a proof of the proposition $(\forall l: \text{int/ist}. \text{sorted}(f(l)) \land \text{permut}(l, f(l)))$.

2 Pure Type Systems

Pure Type Systems, introduced by Berardi and Terlouw, are a generalization of the systems in Barendregt's $\lambda$-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 $(S, A, R)$ with

- $S$ is a set of symbols called the sorts
- $A \subseteq S \times S$, a set of axioms
- $R \subseteq S \times S$, a set of rules

The set of terms $M$ and contexts $\Gamma$ is given by

$$ M ::= x | s | (M M) | (\lambda x : M. M) | (\Pi x : M. M) $$

$$ \Gamma ::= \epsilon | \Gamma, x : M $$

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

- **(axiom)** \[ \epsilon \vdash s_1 : s_2 \] for $s_1 : s_2 \in A$
- **(start)** \[ \Gamma, x : A \vdash b : A \] where $x$ is $\Gamma$-fresh
- **(weakening)** \[ \Gamma \vdash b : B \quad \Gamma \vdash A : s \]
  \[ \Gamma, x : A \vdash b : B \] where $x$ is $\Gamma$-fresh
- **(formation)** \[ \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \]
  \[ \Gamma \vdash (\Pi x : A. B) : s_2 \] for $(s_1, s_2) \in R$

2 In [Bar92] PTS rules are triples $(s_1, s_2, s_3)$, but here this simpler definition suffices.
(abstraction) \[ \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s \]
\[ \Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B) \]

(application) \[ \Gamma \vdash b : (\Pi x : A. B), \Gamma \vdash a : A \]
\[ \Gamma \vdash ba : B[x := a] \]

(conversion) \[ \Gamma \vdash b : B, \Gamma \vdash B' : s, B \simeq B' \]
\[ \Gamma \vdash b : B' \]

where \( s \) ranges over sorts, i.e. \( s \in S \), and \( \simeq \) is \( \beta \)-equality.

**Notation.** \( \Gamma \vdash A : B : C \) is short for \( \Gamma \vdash A : B \) and \( \Gamma \vdash B : C \). We write \( \rightarrow_{\beta} \) for \( \beta \)-reduction.

If \( x \) does not occur free in \( B \), \( (\Pi x : A. B) \) is written as \( A \rightarrow B \). For \( (s_1, s_2) \in R \) we simplify the formation rule to
\[ \frac{\Gamma \vdash A : s_1, \Gamma \vdash B : s_2}{\Gamma \vdash A \rightarrow B : s_2} \]
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:

**Definition 2** \((\lambda \rightarrow, \lambda \mathbb{A}, \lambda \omega, \lambda C)\). Church’s simply typed \( \lambda \)-calculus is the PTS \( \lambda \rightarrow \)
\[ S = \{\ast, \square\}, \quad A = \{\ast : \square\}, \quad R = \{\ast, \ast\} \]
The second order typed lambda calculus (system \( F \)) is the PTS \( \lambda \mathbb{A} \)
\[ S = \{\ast, \square\}, \quad A = \{\ast : \square\}, \quad R = \{\ast, \ast\} \]
Girard’s system \( F_\omega \) is the PTS \( \lambda \omega \)
\[ S = \{\ast, \square\}, \quad A = \{\ast : \square\}, \quad R = \{\ast, \ast\} \]
The Calculus of Constructions is the PTS \( \lambda C \)
\[ S = \{\ast, \square\}, \quad A = \{\ast : \square\}, \quad R = S^2 \]

3 \( \lambda \mathbb{A}_L \)

The system \( \lambda \mathbb{A}_L \) 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 proposition) for the second.
3.1 \( \lambda \omega \) as a programming language: \( \lambda \omega \)

\( \lambda \omega \) is the PTS

\[
S = \{ \ast_\ast, \square_\ast \}, \quad A = \{ \ast_\ast : \square_\ast \}, \quad R = \{ ((\ast_\ast, \ast_\ast), (\square_\ast, \ast_\ast), (\square_\ast, \square_\ast)) \}
\]

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

The kinds are expressions \( \kappa \) such that \( \Gamma \vdash \kappa : \square_\ast \). The rule \( (\square_\ast, \square_\ast) \) controls the formation of kinds:

\[
\begin{array}{c}
(\square_\ast, \square_\ast) \\
\hline
\Gamma \vdash \kappa_1 : \square_\ast, \Gamma \vdash \kappa_2 : \square_\ast \\
\hline
\Gamma \vdash \kappa_1 \rightarrow \kappa_2 : \square_\ast
\end{array}
\]

The kinds are generated by \( \kappa ::= \ast_\ast | \kappa \rightarrow \kappa \). This includes \( \ast_\ast \), the type of data types, \( \ast_\ast \rightarrow \ast_\ast \), the type of functions from data types to data types, and \( \ast_\ast \rightarrow \ast_\ast \rightarrow \ast_\ast \), the type of binary functions from data types to data types.

The constructors are expressions \( \sigma \) that have a kind as their type, i.e. \( \Gamma \vdash \sigma : \kappa : \square_\ast \) for some \( \kappa \). This includes the data types (inhabitants of \( \ast_\ast \)), the functions from data types to data types, e.g. \textit{list}, and the binary functions from data types to data types, e.g. \texttt{+} and \texttt{x}.

The programs are expressions \( m \) such that \( \Gamma \vdash m : \sigma : \ast_\ast \) for some data type \( \sigma \).

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 \( \sigma : \ast_\ast \),
- abstraction over a kind, i.e. \( \lambda \kappa : \kappa \ldots \) where \( \kappa : \square_\ast \). Since \( \ast_\ast : \square_\ast \), this includes abstraction over all data types, as for example in the polymorphic identity \( \lambda \alpha : \ast_\ast, \lambda z : \alpha . z \).

The rule \( (\ast_\ast, \ast_\ast) \) allows the formation of function types, used to type the first form of abstraction

\[
\begin{array}{c}
(\ast_\ast, \ast_\ast) \\
\hline
\Gamma \vdash \rho : \ast_\ast, \Gamma \vdash \sigma : \ast_\ast \\
\hline
\Gamma \vdash \rho \rightarrow \sigma : \ast_\ast
\end{array}
\]

\( \rho \rightarrow \sigma \) is the type of functions from \( \rho \) to \( \sigma \).

The rule \( (\square_\ast, \ast_\ast) \) allows the formation of polymorphic data types, used to type the second form of abstraction

\[
\begin{array}{c}
(\square_\ast, \ast_\ast) \\
\hline
\Gamma \vdash \kappa : \square_\ast, \Gamma \vdash \alpha : \kappa \vdash \sigma : \ast_\ast \\
\hline
\Gamma \vdash (\Pi \alpha : \kappa . \sigma) : \ast_\ast
\end{array}
\]

For example, \( (\Pi \alpha : \ast_\ast, \alpha \rightarrow \alpha) \) is a polymorphic data type (it is the type of the polymorphic identity).

\textit{Convention.} \( \kappa, \kappa_1, \kappa_2 \) range over kinds; \( \rho, \sigma, \tau \) range over constructors; \( \alpha, \beta \) range over constructor-variables; \( m, n, f \) range over programs.

In \( \lambda \omega \) there are representations of many data types such as boolean, 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

\[ S = \{ *_p, \square_p \}, \ A = \{ *_p : \square_p \}, \ R = \{ (*_p, *_p), (\square_p, *_p), (\square_p, \square_p) \} \]

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

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

\[
\begin{array}{c}
\Gamma \vdash \Phi_1 : \square_p \\
\Gamma \vdash \Phi_2 : \square_p
\end{array}
\]

\[ \Gamma \vdash \Phi_1 \rightarrow \Phi_2 : \square_p \]

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

The prop-constructors are expressions \( \phi \) that have a prop-kind as their type, i.e. \( \Gamma \vdash \phi : \Phi : \square_p \) for some \( \Phi \). 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 \( \Gamma \vdash H : *_p \) for some proposition \( \phi \).

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

\[
\begin{array}{c}
\Gamma \vdash \phi : *_p \\
\Gamma \vdash \psi : *_p
\end{array}
\]

\[ \Gamma \vdash \phi \Rightarrow \psi : *_p \]

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

\[
\begin{array}{c}
\Gamma \vdash \Phi : \square_p \\
\Gamma, \alpha : \Phi \vdash \phi : *_p
\end{array}
\]

\[ \Gamma \vdash (\forall \alpha : \Phi : \phi) : *_p \]

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

Convention. \( \Phi, \Phi_1, \Phi_2 \) range over prop-kinds; \( \phi, \psi \) 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 \( \lambda \omega \). Also, we can define \( True =_{def} (\forall \phi : *_p : \phi \Rightarrow \phi) \), \( False =_{def} (\forall \phi : *_p : \phi) \), and \( \neg =_{def} (\lambda \phi : *_p : \phi \Rightarrow False) : *_p \rightarrow *_p \). 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 \( dbng = (\forall \phi : *_p : (\neg \phi) \Rightarrow \phi) \) can be assumed.
3.3 $\lambda \omega_L$

The programming logic $\lambda \omega_L$ consists of a programming language $\lambda \omega_p$ 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 $(\ast_\sigma, \ast_p)$ and $(\Box_\sigma, \ast_p)$ are added.

$$(\ast_\sigma, \ast_p) \quad \frac{\Gamma \vdash \sigma : \ast_\sigma \quad \Gamma, \alpha : \sigma \vdash \phi : \ast_p}{\Gamma \vdash (\forall \alpha : \sigma. \phi) : \ast_p}$$

$$(\Box_\sigma, \ast_p) \quad \frac{\Gamma \vdash \kappa : \Box_\sigma \quad \Gamma, \alpha : \kappa \vdash \phi : \ast_p}{\Gamma \vdash (\forall \alpha : \kappa. \phi) : \ast_p}$$

Since $\ast_\sigma : \Box_\sigma$, the second rule allows for instance quantification over all data types, i.e. $(\forall \alpha : \ast_\sigma \ldots)$

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

$\sigma \rightarrow \ast_p, \sigma \rightarrow \sigma \rightarrow \ast_p$ and $(\Pi \alpha : \ast_\sigma, \alpha \rightarrow \alpha \rightarrow \ast_p)$ are all prop-kinds, i.e. they have type $\Box_\sigma$. The formation of the first two requires the rule

$$(\ast_\sigma, \Box_p) \quad \frac{\Gamma \vdash \sigma : \ast_\sigma \quad \Gamma \vdash \Phi : \Box_p}{\Gamma \vdash \sigma \rightarrow \Phi : \Box_p}$$

The formation of $(\Pi \alpha : \ast_\sigma, \alpha \rightarrow \alpha \rightarrow \ast_p)$ also requires the rule

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

The standard second-order logic definitions can now be used to define existential quantifications $(\exists x : \sigma. \phi)$ and $(\exists \alpha : \ast_\sigma, \phi)$ 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, $=_{L}$: $(\Pi \alpha : \ast_\sigma, \alpha \rightarrow \alpha \rightarrow \ast_p)$, can be defined as

$$\lambda \alpha : \ast_\sigma, \lambda x, y : \alpha. (\forall \phi : \alpha \rightarrow \ast_p. (\phi x) \Rightarrow (\phi y))$$

$=_{L}$ 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

$$S = \{ \ast_{\sigma}, \Box_{\sigma}, \ast_p, \Box_p \}$$

$$A = \{ (\ast_\sigma : \Box_\sigma), (\ast_p : \Box_p) \}$$

$$R = \{ (\Box_\sigma, \ast_\sigma), (\ast_\sigma, \ast_\sigma), (\ast_\sigma, \ast_p), (\Box_\sigma, \Box_p), (\ast_p, \Box_p), (\ast_\sigma, \ast_\sigma), (\ast_\sigma, \ast_p), (\ast_p, \ast_\sigma), (\ast_p, \ast_p) \}$$
So $\lambda_\omega _L$ consists of

- $\lambda_\omega _L$ for programs and their types: $\{(\square _s, \square _s), (\square _s, *_s), (*_s, *_s)\}$
- $\lambda_\omega _P$ for the propositions and their proofs: $\{(\square _p, \square _p), (\square _p, *_p), (*_p, *_p)\}$
- all possible dependencies of propositions and proofs on programs and types: $\{(\square _s, \square _p), (\square _s, *_s), (*_s, \square _p), (*_s, *_p)\}$

$\lambda_\omega _L$ does not have all possible rules. It does not have the rules

- $(*_s, \square _s)$ – the program/term dependent types.
  This is because we have chosen $\lambda_\omega$ and not $\lambda C$ as programming language.
- $(*_p, \square _p)$ – 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 $\Gamma \vdash _{\lambda_\omega _L} A : *_s$, $\Gamma \vdash _{\lambda_\omega _L} A : \square _s$, or $A \equiv \square _s$.
Then $| \Gamma | := ^{\lambda_\omega _L} a : A$,
where $| e | := \epsilon$, and $| \Gamma, x : A | := \begin{cases} \{ | \Gamma |, x : A \text{ if } \Gamma \vdash A : *_s \text{ or } \Gamma \vdash A : \square _s \} & \text{ if } \Gamma \vdash A : *_s \\ | \Gamma | & \text{ if } \Gamma \vdash A : *_p \text{ or } \Gamma \vdash A : \square _p \end{cases}$

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

If we forget about the subscripts $p$ and $s$, it is a subsystem of $\lambda C$:

Lemma 5. If $\Gamma \vdash _{\lambda_\omega _L} a : A$ then $| \Gamma | := ^{\lambda C} a : | A |$, where $| z |$ is $z$ with $*$ substituted for $*_s$ and $*_p$, and $\square$ substituted for $\square _s$ and $\square _p$.

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 ($\text{False}$, defined as $(\forall \phi : *_p \phi$), is not provable).

4 Program and proof development in $\lambda_\omega _L$

$\lambda_\omega _L$ programs and types can be annotated in $\lambda_\omega _L$. For a $\lambda_\omega _L$-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 $\lambda_\omega _L$-type is annotated with a predicate on that type (a specification).

<table>
<thead>
<tr>
<th>$\lambda_\omega _L$ annotation</th>
<th>$\lambda_\omega _L$ annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \sigma : *_s$</td>
<td>$\Gamma \vdash P : \sigma \rightarrow *_p$</td>
</tr>
<tr>
<td>$\Gamma \vdash m : \sigma : *_s$</td>
<td>$\Gamma \vdash H_m : (P m) : *_p$</td>
</tr>
</tbody>
</table>

The task of finding a program that satisfies a given specification is the following:
given: a type $\sigma$, $\Gamma \vdash \sigma : *_s$, and a specification $P$, $\Gamma \vdash P : \sigma \rightarrow *_p$
find: a program $m$, $\Gamma \vdash m : \sigma$, and a proof $H_m$, $\Gamma \vdash H_m : (P m)$.
For every derivation rule in $\lambda \omega$, there is a corresponding derivation rule in $\lambda \omega_L$ dealing with these annotations.

$\lambda \omega$-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 $\lambda \omega$-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 $\lambda \omega$-judgements are written in boldface.

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

\[
\Gamma \vdash f : \rho \rightarrow \sigma \quad \Gamma \vdash H_f : (\forall z : \rho. P z \Rightarrow Q x (f z))
\]

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

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

(i) $\Gamma, x : \rho \vdash m : \sigma$ \quad (ii) $\Gamma, x : \rho, H_x : P x \vdash H_m : Q x m$

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

\[
\Gamma \vdash (\lambda z : \rho. m) : \rho \rightarrow \sigma \\
\Gamma \vdash (\lambda z : \rho. \lambda H_x : P x. H_m) : (\forall z : \rho. P z \Rightarrow Q x m) \\
\approx (\forall z : \rho. P z \Rightarrow Q x ((\lambda z : \rho. m) z))
\]

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

\[
\Gamma \vdash m : \sigma \quad \Gamma \vdash H_m : Q' m
\]

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

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

(i) $\Gamma \vdash n : \rho$ \quad (ii) $\Gamma \vdash H_n : P n$

and a program $f$ with a proof $H_f$ such that

(iii) $\Gamma \vdash f : \rho \rightarrow \sigma$ \quad (iv) $\Gamma \vdash H_f : (\forall z : \rho. P z \Rightarrow Q x (f z))$
From (i) and (iii) it follows that \( fn \) has the right type, and from (ii) and (iv) it follows that \( fn \) satisfies the specification \( Q \):

\[
\Gamma \vdash fn : \sigma \quad \Gamma \vdash HfnHn : (Q n (fn)) \simeq (Q'(fn))
\]

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

\[
\begin{align*}
\Gamma \vdash \rho : * \\
\Gamma, x : \rho \vdash \sigma : * \\
\Gamma \vdash \rho \rightarrow \sigma : * \\
\Gamma \vdash (\lambda f : \rho \rightarrow \sigma. \forall x : \rho. P x \Rightarrow Q'(fx)) : (\rho \rightarrow \sigma) \rightarrow *
\end{align*}
\]

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

For \( \Gamma \vdash \rho : * \), \( \sigma : * \), and \( \Gamma \vdash P : \rho \rightarrow * \), \( Q : \rho \rightarrow - \rightarrow * \), we have the following coupled derivation rules

\[
\begin{align*}
\Gamma, x : \rho \vdash m : \sigma & \quad \Gamma, x : \rho, H_\rho : P x \vdash H_m : Q x m \\
\Gamma \vdash (\lambda x : \rho. m) : \rho \rightarrow \sigma & \quad \Gamma \vdash (\lambda x : \rho. \lambda H_\rho : P x. H_m) : (\forall x : \rho. P x \Rightarrow Q x ((\lambda x : \rho. m)x))
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash f : \rho \rightarrow \sigma & \quad \Gamma \vdash H_f : (\forall x : \rho. P x \Rightarrow Q x (fx)) \\
\Gamma \vdash n : \rho & \quad \Gamma \vdash H_n : P n \\
\Gamma \vdash HfnHn : Q n (fn) & \quad \Gamma \vdash HfnHn : Q n (fn)
\end{align*}
\]

All these rules are derivable in \( \lambda \omega_L \).

4.2 Polymorphic types

In addition to function types of the form \( \rho \rightarrow \sigma \) we also have polymorphic types of the form \( (\forall \alpha : \kappa. \sigma) \), with \( \kappa : \Box \). For the \( \lambda \omega \) derivation rules for these polymorphic types we can also give corresponding \( \lambda \omega_L \) rules:

\[
\begin{align*}
\Gamma, \alpha : \kappa \vdash \sigma : * & \quad \Gamma, \alpha : \kappa \vdash P : \sigma \rightarrow * \rho \\
\Gamma \vdash (\Pi \alpha : \kappa. \sigma) : * & \quad \Gamma \vdash (\lambda f : (\Pi \alpha : \kappa. \sigma). \forall \alpha : \kappa. P(\alpha)) : (\Pi \alpha : \kappa. \sigma) \rightarrow * \rho
\end{align*}
\]

\[
\begin{align*}
\Gamma, \alpha : \kappa \vdash m : \sigma & \quad \Gamma, \alpha : \kappa \vdash H_m : (P m) \\
\Gamma \vdash (\lambda \alpha : \kappa. m) : (\Pi \alpha : \kappa. \sigma) & \quad \Gamma \vdash (\lambda \alpha : \kappa. H_m) : (\forall \alpha : \kappa. P m) \\
& \quad \simeq (\forall \alpha : \kappa. P((\lambda \alpha : \kappa. m)\alpha))
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \rho : \kappa & \quad \Gamma \vdash f : (\Pi \alpha : \kappa. \sigma) \\
\Gamma \vdash f \rho : \sigma[\alpha \leftarrow \rho] & \quad \Gamma \vdash H_f : (\forall \alpha : \kappa. P(\alpha)) \\
\Gamma \vdash H_f \rho : P[\alpha \leftarrow \rho](f \rho) & \quad \Gamma \vdash H_f \rho : P[\alpha \leftarrow \rho](f \rho)
\end{align*}
\]

10
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 $\lambda \omega$, (see for example [BB85]). What these representations are is are not important here. We only have to know there are terms

- $\text{bool} : \ast$, $\text{true} : \text{bool}$, $\text{false} : \text{bool}$ and $\text{if} : (\Pi \alpha : \ast, \text{bool} \rightarrow \alpha \rightarrow \alpha)$ such that $(\text{if} \alpha \text{true} m) \rightarrow \beta m$ and $(\text{if} \alpha \text{false} m n) \rightarrow \beta n$.
- $\text{nat} : \ast$, $0 : \text{nat}$, $S : \text{nat} \rightarrow \text{nat}$ and $\text{iter} : (\Pi \alpha : \ast, \alpha \rightarrow \alpha \rightarrow (\text{nat} \rightarrow \alpha))$ such that $(\text{iter} \alpha m f 0) \rightarrow \beta a$ and $(\text{iter} \alpha m f (S n)) \rightarrow \beta f(\text{iter} \alpha m f n)$.

Using $\text{iter}$ iterative functions on the natural numbers can be defined: $(\text{iter} \alpha m f)$ is the function mapping $(80)$ to $(I'm)$.

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

$$
\forall \phi : \text{bool} \rightarrow \ast. (\phi \text{true}) \Rightarrow (\phi \text{false}) \Rightarrow (\forall b : \text{bool}. (\phi b))
$$

$$
\forall \phi : \text{nat} \rightarrow \ast. (\phi 0) \Rightarrow (\forall x : \text{nat}. (\phi x) \Rightarrow (\phi (S x))) \Rightarrow (\forall n : \text{nat}. (\phi n))
$$

$$
\forall \alpha, \beta : \ast, \forall \phi : (\alpha + \beta) \rightarrow \ast,
\forall a : \alpha. \phi(in_1 \alpha \beta a) \Rightarrow \forall b : \beta. \phi(in_2 \alpha \beta b) \Rightarrow (\forall x : \alpha + \beta. (\phi x))
$$

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

<table>
<thead>
<tr>
<th>$\Gamma \vdash \sigma : \ast$</th>
<th>$\Gamma \vdash P : \text{bool} \rightarrow \sigma \rightarrow \ast$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash b : \text{bool}$</td>
<td>$\Gamma \vdash H_m : (P \text{true} m)$</td>
</tr>
<tr>
<td>$\Gamma \vdash m : \sigma$</td>
<td>$\Gamma \vdash H_n : (P \text{false} n)$</td>
</tr>
<tr>
<td>$\Gamma \vdash n : \sigma$</td>
<td>$\Gamma \vdash (H_{\text{if}}. P m H_m n H_n) : P b (\text{if} b \text{then} m \text{else} n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\Gamma \vdash \sigma : \ast$</th>
<th>$\Gamma \vdash P : \text{nat} \rightarrow \sigma \rightarrow \ast$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash m : \sigma$</td>
<td>$\Gamma \vdash H_m : (P 0 m)$</td>
</tr>
<tr>
<td>$\Gamma \vdash f : \sigma \rightarrow \sigma$</td>
<td>$\Gamma \vdash H_f : (\forall y : \sigma. \forall n : \text{nat}. (P n (f y)) \Rightarrow (P (S n)(f y)))$</td>
</tr>
<tr>
<td>$\Gamma \vdash (\text{iter} \sigma m f)$</td>
<td>$\Gamma \vdash (H_{\text{iter}}. P m H_m f H_f) : (\forall n : \text{nat}. (P n (\text{iter} \sigma m f n)))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\Gamma \vdash \rho + \sigma : \ast$, $\tau : \ast$</th>
<th>$\Gamma \vdash P : (\rho + \sigma) \rightarrow \tau \rightarrow \ast$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash f : \rho \rightarrow \tau$</td>
<td>$\Gamma \vdash H_f : (\forall z : \rho. (P(\text{in}_1 \rho \sigma z)(f z)))$</td>
</tr>
<tr>
<td>$\Gamma \vdash g : \sigma \rightarrow \tau$</td>
<td>$\Gamma \vdash H_g : (\forall z : \sigma. (P(\text{in}_2 \rho \sigma z)(g z)))$</td>
</tr>
<tr>
<td>$\Gamma \vdash [f, g] : (\rho + \sigma) \rightarrow \tau$</td>
<td>$\Gamma \vdash (H_{\text{out}}. P \rho \sigma \tau P f H_f g H_g) : (\forall z : \rho + \sigma. P([f, g] z))$</td>
</tr>
</tbody>
</table>
So $H_{lf}$ is a proof of the following proposition:

$$\forall \alpha : \ast. \forall P : \text{bool} \rightarrow \alpha \rightarrow \ast.$$  
$$\forall m : \alpha. (P \text{true } m) \Rightarrow$$  
$$\forall n : \alpha. (P \text{false } n) \Rightarrow$$  
$$\forall b : \text{bool}. (P b \ (\text{if } b \text{ then } m \text{ else } n))$$

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

$$\Gamma \vdash \sigma : \ast,$$  
$$\Gamma \vdash b : \text{bool},$$  
$$\Gamma \vdash m : \sigma,$$  
$$\Gamma \vdash n : \sigma,$$  
$$\Gamma \vdash (\text{if } b \text{ then } m \text{ else } n) : \sigma$$  

A possible proof of $Q (\text{if } b \text{ then } m \text{ else } n)$ is

$$(H_{lf} \sigma P(\lambda H : \ldots. H_m)n(\lambda H : \ldots. H_n)\text{beta}_f),$$

where $P : \text{bool} \rightarrow \sigma \rightarrow \ast$ is

$$(\lambda x : \text{bool}. \lambda y : \sigma. (x =_\text{bool} b) \Rightarrow (Q y))$$

and $\text{beta}_f$ is a proof of $b =_\text{bool} b$.

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

$$\Gamma \vdash \sigma : \ast,$$  
$$\Gamma \vdash n : \text{nat},$$  
$$\Gamma \vdash m_1 : \sigma,$$  
$$\Gamma \vdash m_2 : \sigma,$$  
$$\Gamma \vdash (\text{if } \text{zero } n \text{ then } m_1 \text{ else } m_2) : \sigma$$  

A possible proof for $Q (\text{if } \text{zero } n \text{ then } m_1 \text{ else } m_2)$ is

$$(H_{iter} \sigma P(\lambda H : \ldots. H_m) \ldots)$$

$$(\lambda y : \sigma. \lambda x : \text{nat}. \lambda H_2 : (P z y). \lambda H_3 : ((S x) =_\text{nat} n).((\lambda H : \ldots. H_m(\lambda H_2(\lambda H_3(n H_{ref}))))$$

$$(n H_{ref}))$$

where $P : \text{nat} \rightarrow \sigma \rightarrow \ast$ is $\lambda x : \text{nat}. \lambda y : \sigma. (x =_\text{nat} n) \Rightarrow (Q y)$, $H_{nz}$ is a proof of $((S x) =_\text{nat} n) \Rightarrow (n \not=*_\text{nat} 0)$, and $H_{ref}$ is a proof of $n =_\text{nat} n$.

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

We want a program $\text{mod}$ and a proof $H_{\text{mod}}$ such that

$$\Gamma \vdash \text{mod} : \text{nat} \rightarrow \text{nat},$$  
$$H_{\text{mod}} : (\forall n : \text{nat}. (\text{MOD } n (\text{mod } n)))$$

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

$$\Gamma \vdash m_0 : \text{nat},$$
\( H_0 : (MOD 0 m_0) \)
\( \vdash m_S : \text{nat} \rightarrow \text{nat} \).
\( H_S : (\forall m_n : \text{nat}. \forall n : \text{nat}. (MOD n m_n) \Rightarrow (MOD (S n) (m_S m_n))) \)

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

We choose \( m_{S_n} \equiv \text{if zero } m_n \text{ then } m_{\text{odd}} \text{ else } m_{\text{even}} \). We then have to find \( m_{\text{odd}}, m_{\text{even}}, H_{\text{odd}} \) and \( H_{\text{even}} \) such that
\( m_n : \text{nat} \vdash m_{\text{odd}} : \text{nat} \)
\( m_n : \text{nat}, n : \text{nat}, H_{mn} : (MOD n m_n), H : (m_n = \text{nat} 0) \vdash H_{\text{odd}} : MOD (S n) m_{\text{odd}} \)
\( m_n : \text{nat} \vdash m_{\text{even}} : \text{nat} \)
\( m_n : \text{nat}, n : \text{nat}, H_{mn} : (MOD n m_n), H : (m_n \neq \text{nat} 0) \vdash H_{\text{even}} : MOD (S n) m_{\text{even}} \)

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

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 \( F_w \)-program and a correctness proof, we directly construct an \( F_w \) 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 [Lu99]), 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 \( \rho \rightarrow \sigma \) only specifications of the form \( \lambda f : \rho \rightarrow \sigma. \forall x : \rho. (P x) \Rightarrow (Q (f x)) \) 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 \(\text{dbg} : (\forall \phi : a, \neg \phi \Rightarrow \phi)\).

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] [CMS91]).

One useful extension of the programming language is a fixed-point combinator 
\(Y : (\Pi \alpha : a, (\alpha \rightarrow a) \rightarrow a)\). 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 
\(\subseteq : (\Pi \alpha : a, \alpha \rightarrow a \rightarrow a)\), a constant \(\bot : (\Pi \alpha : a, \alpha)\) 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 ×, and recursive types \((\mu \alpha : a, \alpha)\) 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


In this series appeared:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>90/2</td>
<td>K.M. van Hee P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/5</td>
<td>J.A. Brzozowski J.C. Ebergen</td>
<td>On the delay-sensitivity of gate networks, p. 23.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P.America F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
<tr>
<td>90/16</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>A fully abstract model for concurrent logic languages, p. 23.</td>
</tr>
<tr>
<td>90/17</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.</td>
</tr>
<tr>
<td>Page</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>91/02</td>
<td>R. P. Nederpelt, H. C. M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if..., then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J. P. Katoen, L. A. M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of P-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K. M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E. Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.</td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem, p. 12.</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built, p. 63.</td>
</tr>
</tbody>
</table>

An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

Transforming Functional Database Schemes to Relational Representations, p. 21.

Transformational Query Solving, p. 35.

Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.


Assertional Data Reification Proofs: Survey and Perspective, p. 18.


Z and high level Petri nets, p. 16.

Formal semantics for BRM with examples, p. 25.

A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.

The GOOD based hypertext reference model, p. 12.

Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.

A compositional proof system for dynamic process creation, p. 24.

Correctness of Acceptor Schemes for Regular Languages, p. 31.

An Algebra for Process Creation, p. 29.

Some algorithms to decide the equivalence of recursive types, p. 26.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p. 45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p. 22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p. 36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Seljée</td>
<td>Integrity checking in deductive databases; an exposition, p. 32.</td>
</tr>
<tr>
<td>92/17</td>
<td>W.M.P. van der Aalst</td>
<td>Interval timed coloured Petri nets and their analysis, p. 20.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>92/18</td>
<td>R.Nederpelt</td>
<td>A unified approach to Type Theory through a refined lambda-calculus, p. 30.</td>
</tr>
<tr>
<td></td>
<td>F. Kamareddine</td>
<td></td>
</tr>
<tr>
<td>92/19</td>
<td>J.C.M.Baeten</td>
<td>Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.</td>
</tr>
<tr>
<td></td>
<td>J.A.Bergstra</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S.A.Smolka</td>
<td></td>
</tr>
<tr>
<td>92/20</td>
<td>F.Kamareddine</td>
<td>Are Types for Natural Language? P. 32.</td>
</tr>
<tr>
<td>92/21</td>
<td>F.Kamareddine</td>
<td>Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.</td>
</tr>
<tr>
<td>92/22</td>
<td>R. Nederpelt</td>
<td>A useful lambda notation, p. 17.</td>
</tr>
<tr>
<td></td>
<td>F.Kamareddine</td>
<td></td>
</tr>
<tr>
<td>92/23</td>
<td>F.Kamareddine</td>
<td>Nominalization, Predication and Type Containment, p. 40.</td>
</tr>
<tr>
<td></td>
<td>E.Klein</td>
<td></td>
</tr>
<tr>
<td>92/24</td>
<td>M.Codish</td>
<td>Bottum-up Abstract Interpretation of Logic Programs, p. 33.</td>
</tr>
<tr>
<td></td>
<td>D.Dams</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Eyal Yardeni</td>
<td></td>
</tr>
<tr>
<td>92/25</td>
<td>E.Poll</td>
<td>A Programming Logic for Fo, p. 15.</td>
</tr>
</tbody>
</table>