Systems for open terms: an overview

Citation for published version (APA):

Document status and date:
Published: 01/01/2001

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 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

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.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Systems for Open Terms: An Overview

by

G.I. Jojgov

ISSN 0926-4515
All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/es

Computer Science Reports 01/03
Eindhoven, April 2001
Systems for Open Terms: An Overview

G. I. Jojgov
(Георги И. Жожгов)
Eindhoven University of Technology
GJojgov@win.tue.nl

March, 2001
Abstract

In this paper we make an overview of some existing systems of open (incomplete) terms including ALF, Typelab, OLEG, $\lambda\Pi_c$, Automath, $\lambda c$ and $\lambda s_c$. 
Chapter 1

1.1 Introduction

The increasing complexity and scope of problems being solved using formal tools combined with the state-space explosion observed in many of them has led to increased interest in computer-assisted interactive reasoning for problems for which full automation is unfeasable or impossible. One particular area of interest is interactive theorem proving. The idea is that a human guides the computer by providing information which cannot be automatically deduced or which will help prune the search tree. The computer may still be allowed to automatically solve easy subproblems for which efficient automation is possible.

When talking about (interactive) theorem proving we always have in mind some formal framework in which we express our goals and the input data. Usually this is some elaborate theory involving notions like formula, deduction, proof, etc. This can be for example propositional logic, predicate calculus, $\lambda$-calculus or type theory. The predominant part of the currently available systems for interactive theorem proving are based on type theory. There we present formulas as types and their proofs as terms of these types using the Curry-Howard isomorphism.

Due to its incremental nature, in the interactive process of proof construction we need to handle incomplete proofs which in a type theory framework correspond to incomplete terms. These are terms that contain "holes" corresponding to (the parts of) the proofs that are not constructed yet. We call such terms also open.

An interactive theorem prover needs to guarantee at least soundness of the process of proof construction. This means that the terms constructed correspond to real proofs in the underlying formal system. The failure of soundness when open terms are being treated in a naive way was the reason why research in the field of open terms was initiated. This theory helps us model the intermediate steps in the process of interactive theorem proving thus allowing us to reason about this process in a formal way. Such a theory also gives theoretical justification to the implementations of different tools which in many cases in the past were based on intuitive ideas only.

Another potential benefit of the theory of open terms is that it may give rise to new applications by giving an abstract overview of the interactive process which is not being obscured by implementation details and decisions.

This report is organized in several parts. First we discuss the points of interest for a system of open terms which can also play the role of classification criteria.

Then we discuss in brief the following systems: ALF, Typelab, OLEG, ΛΠΣ, the context calculus $\lambda c$, Automath and the $\lambda x$-calculus.
We conclude with a discussion and some observations. We assume the reader is acquainted with basic type theory and logic.

1.2 Issues of Interest and Classification Criteria

In this section we briefly discuss what we will be interested in when considering a system for open terms, why a particular issue is of importance and how the systems can be classified according to it. Of course, there may be dependencies and not all the topics discussed below are relevant to all systems.

**Representation** The most important issue is how open terms are represented in the formal system. Generally there are two main approaches: Historically first appeared those based on some kind of ad-hoc datastructures. These can be different kind of trees or graphs, lists of the tactics applied and so on. The advantages of such representations are usually in the field of performance and ease of implementation. However this makes the representations of open terms tool-dependent (in some cases even worse: version-dependent).

On the other hand we have the second approach, which calls for extending the language of terms with so-called metavariables. These variables are intended as placeholders for terms that are not yet constructed. This gives us a tool-independent presentation of open terms which is based on formal rules. As a consequence the developments in a tool can be compared to the theoretical results and expectations which improves the reliability of the tool. Currently metavariables are de facto the standard for representing open terms.

There is however no widely agreed upon way to introduce metavariables in general. There are numerous solutions for various systems, but they are quite different and in most cases incompatible.

**Explicit Substitutions** Furthermore, systems based on metavariables can be subdivided into two classes based on the way they treat computations with open terms. One of them is a large class of systems of metavariables and explicit substitutions. In the other class we find systems which in some way avoid the complications incurred by the cross-influence between metavariables and computation.

**Underlying System** We may classify the systems according to their underlying system (i.e. the system for which we want to introduce open terms). This system can be for example propositional logic, predicate logic, higher-order logic. For calculi based on type theory this classification can be done in terms of the systems of the λ-cube, as PTS specification and so on. There are also adaptable systems which allow the user to specify the formal framework that he or she wants to use.
Metatheoretical Properties For systems based on type theory we can investigate various relevant properties like for example confluence, strong/weak normalization, subject reduction, decidability of conversion, type inference and type checking and these properties can be used to classify the systems accordingly.

Use of de Bruijn indices Very often systems which are intended for implementation purposes differ from the ones that are purely theoretical. One such difference is for example the use of de Bruijn indices instead of named variables or a combination of the two.

1.3 Informal Use of the Metavariabes in Interactive Theorem Proving

Let us illustrate the use of metavariables in an interactive session with several small examples:

1.3.1 Metavariables standing for (proof)terms

Step 1: Initial state We are looking for term of type \( A \rightarrow A \). The unknown term is denoted by \(?_1\):

\[ \vdash ?_1 : A \rightarrow A \]

Step 2: We decide that the unknown term is an abstraction and solve \(?_1\) by \( \lambda x : A.?_2 \) introducing a new metavariable \(?_2\). Note that \(?_2\) is typed in the context \( x : A \)

\[ \vdash \lambda x : A.?_2 : A \rightarrow A \]
\[ x : A \vdash ?_2 : A \]

Step 3: Now the task has become easy, because we have to find a term of type \( A \) in context \( x : A \). Of course we can choose \(?_2 = x\)

\[ \vdash \lambda x : A.x : A \rightarrow A \]
\[ x : A \vdash x : A \]

Step 3': Note that instead of choosing to solve \(?_2\) by \( x\), we could have chosen another path. Alternative solution is for example \(?_2 = ?_3 ?_4\). This nondeterminism is usually imposed by the complexity of the type inhabitation problem, which in most of the cases is undecidable. Another reason is completeness - we want to be able to derive a judgement in any possible way, because in some cases (interactive program refinement for example) the actual term inhabiting the type is of importance.¹

\[ \vdash \lambda x : A.?_3 ?_4 : A \rightarrow A \]

¹As anyone aware of the difference between the quicksort algorithm and the one checking all permutations for a sorted one will agree.
\[ x : A \vdash \ ?_3 : A \]
\[ x : A \vdash \ ?_3 : A \to A \]
\[ x : A \vdash \ ?_4 : A \]

And in several steps the metavariables can be solved by \( ?_3 = \lambda y : A.y, \ ?_4 = x \) or \( ?_3 = \lambda y : A.y, \ ?_4 = x \).

1.3.2 Metavariables as witnesses

Another typical case when we need to use metavariables is when we want to delay the moment when we specify the witness for an existential quantifier. Consider the following arithmetic formula:

\[ \exists x \in N \forall y \in N(\forall z \in N(x + z \leq z)) \implies x \leq y \]

A proof of such a formula involves finding a witness \( t \) for \( x \) and proving \( \forall y(\forall z(t+z \leq z)) \implies t \leq y \). In this case it is easy to see that 0 can be used as a witness, but we want to have the option to delay this to a later moment. This is good for example for didactic reasons, because it usually helps to identify the reasons for choosing the specific witness by making explicit its definitional properties. Delaying the instantiation of an existential quantifier gives us also the opportunity to possibly find the witness automatically at a later stage by means of unification for example. In our case from \( \forall z(x + z \leq z) \) we can deduce \( x \leq 0 \) and this is a clear hint that we can actually use the value 0 for \( x \).

Because a metavariable can stand for a witness in an existential statement, sometimes metavariables are also called existential variables.

The Transitivity Example Note however that using metavariables as means to delay the specification of witnesses occurs not only with existential statements. Consider the transitivity property for a relation \( R \) on a set \( A \):

\[ (\text{transitive } R) : \forall x \forall y \forall z \ x R y \implies y R z \implies x R z \]

Suppose \( a \) and \( c \) are elements of \( A \) and we want to prove \( a R c \) using the transitivity of \( R \). Hence we must instantiate the universally quantified statement by taking \( x := a \) and \( z := c \). But what to take for \( y \)? At this point we may not know that and a sensible strategy would be to postpone this choice for a later moment when we can find a witness \( b \) for \( y \) such that \( a R b \) and \( b R c \). This postponement can be done by introducing a metavariable for \( y \).

1.3.3 Computing with metavariables

In this report we will also be looking at the problem of computations with open terms. Let us give an example when metavariables are involved in a computation. Suppose we want to prove \( \exists f : N \to N \forall n : N(fn = n) \) for some type \( N \). Having forgotten that identity functions exist, we introduce a metavariable \( ?f \)}
as a witness for the existential quantifier:

\[ \vdash ?_1 : \forall n : N(?_1(n) = n) \]
\[ n : N \vdash ?_1 : N \rightarrow N \]

for the sake of the example, we decide to refine \( ?_1 \) by \( \lambda x : N. ?_2 \) introducing a new metavariable:

\[ \vdash ?_1 : \forall n : N((\lambda x : A. ?_2)n = n) \]
\[ n : N, x : A \vdash ?_2 : N \]

but now we have created a redex \( (\lambda x : N. ?_2)n \). If we decide to reduce this redex we obtain \( ?_2\{x := n\} \). Now we see that in order to have \( ?_2\{x := n\} = n \) we can choose either \( ?_2 = x \) or \( ?_2 = n \) because in both cases after the substitution is executed\(^2\) we get \( n \).

This example once again shows the benefit of postponing the moment when we actually specify the witness.

### 1.4 Main Problem to Solve When Introducing Metavariabes

As we saw in the previous section, it is nice to be able to compute with open terms. In systems which have some built-in notion of computation this turns into necessity. Take for example the conversion rule in the Calculus of Constructions. It allows a type to be replaced by an equal one. In this case the introduction of metavariables forces us to define computation on open terms.

The fundamental problem is that we need to compute with something which is not known yet. Let us consider the most common form of computation – \( \beta \)-reduction and look at the following diagram:

\[ \begin{array}{c}
(\lambda x : T. ?_n) t \\
\beta \quad \beta
\end{array} \]

\[ ?_n \]

\[ \downarrow \]

\[ \begin{array}{c}
?_n := x \\
\beta
\end{array} \]

\[ \downarrow \]

\[ \begin{array}{c}
(\lambda x : T. x) t \\
\end{array} \]

\[ x \]

\[ \downarrow \]

\[ t \]

\(^2M\{x := N\} \) is notation for the (meta)operation of substitution of the free occurrences of \( x \) in \( M \) by the term \( N \). Here the expression \( ?_n\{x := N\} \) can only be given meaning after \( ?_n \) is instantiated by a term. Later we will see systems which give meaning to this expression before the instantiation.
We see that ignoring the effects of substitutions on open terms leads to loss of confluence or other strongly desirable properties (as subject reduction in some cases, see section 2.4).

Similar problems occur if the system has local definitions. Then computations involve expansion of definitions and elimination of dummy definitions.
Chapter 2

Open Term Systems for Theorem Proving

In this chapter we will discuss several systems that are specifically designed to facilitate interactive theorem proving.

2.1 ALF

ALF [18] is an interactive proof editor based on Martin-Löf's monomorphic type theory. It employs placeholders and explicit substitutions in order to represent open terms. The theory of open terms of ALF was developed in order to support the main feature of the tool which is allowing the user to directly work with the proof objects. This theory reduces the process of editing proof-objects to typechecking of incomplete terms.

2.1.1 Syntax of the Calculus

The terms of the target calculus (the one to which the completely constructed term should belong to) are given by the following grammar:

\[
\text{Term ::= Var|Const|[|Var|Term|(Term Term)|Term Subst}
\]

Note that in an abstraction term \([x].M\) the abstracted variable \(x\) is not given a type and thus terms do not depend on types. As we see, in a term we may have explicitly given substitutions which have the following syntax:

\[
\text{Subst ::= {}|(Subst, Var := Term)|Subst Subst}
\]

Substitutions are composed as follows:

\[
\{\} \beta = \beta \\
\{ \gamma, x := e \} \beta = \{ \gamma \beta, x := (e \beta) \}
\]
On the level of types we distinguish between types and families of types which are given as follows:

\[
\text{Type ::= } \text{Set} | \text{Type} \rightarrow [\text{Var}] | \text{Family} | \text{Family} \text{ Term} | \text{Type} \text{ Subst}
\]

\[
\text{Family ::= } E | [\text{Var}] | \text{Type} | \text{Family} \text{ Subst}
\]

So a type is either \text{Set} or a function type or an element of a family or a type with a substitution applied to it. Note that when we select a member of a family, we do this by application to a term. Therefore types may depend on terms and hence we are in a dependently typed setting.

Contexts are lists of declarations:

\[
\text{Context ::= } [] | [\text{Context}, \text{Var} : \text{Type}]
\]

The calculus has judgements dealing with terms, types, contexts and substitutions. Each of these categories may have either of the following kinds of judgements: formation judgements, typing judgements and equality judgements. Contexts as lists of declarations are used to type substitutions which are lists of variable assignments: \(\Delta \vdash \gamma : \Gamma\). The equality judgements internalize the different notions of reductions present in the system. Take for example the \(\beta\)-conversion on types. The corresponding rule is:

\[
\frac{\Gamma, x : \alpha' \vdash \alpha : \text{Type} \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash a : \alpha' \gamma}{\Delta \vdash (\{x\} \alpha \gamma)a = \alpha\{\gamma, x := a\} : \text{Type}} \quad \text{\(\beta\)-Subst}
\]

Note the presence of the substitution \(\gamma\) just outside the scope of the abstraction binder. The rule has this form because according to the rules for substitutions, a substitution cannot be distributed over abstraction and therefore substitutions generated by other redexes tend to accumulate at abstractions. As a side effect of this approach we do not have to be concerned about \(\alpha\)-conversion because terms containing free variables in a substitution will never go into the scope of possibly conflicting binders\(^1\).

2.1.2 Adding incomplete (open) terms

ALF supports incomplete terms by introducing the so-called placeholders. These correspond to metavariables in other systems. The placeholders are denoted by identifiers starting with '?'\(^\star\). So incomplete terms are terms containing placeholders:

\[
\text{Term ::= } \ldots | ?\alpha
\]

Incomplete types are types containing incomplete terms and incomplete contexts are contexts containing incomplete types.

Each placeholder has an unique expected type (possibly incomplete) and an unique local context (also possibly incomplete). This means that the placeholder

\(^1\)Nevertheless, there are some other problems with \(\alpha\)-conversion, discussed in section 4.4 of [18]
may be refined by a term whose free variables are from the local context and in this context this term should have a type identical to the expected type of the placeholder.

The typechecking in this system poses specific problems, some of which are not encountered in other systems because of the nature of the underlying type system. One of them is caused by the lack of type annotations of the bound variables in λ-abstraction. Consider for example the term

\[ [x]x \]

It can be inferred using the derivation rules that in the empty context this term is both of type \( \text{Set} \rightarrow \text{Set} \) and \( \text{Bool} \rightarrow \text{Bool} \) for example. Therefore the following problem

\[ f(\lambda_1, \lambda_1) : \text{Set} \]

where

\[ f : (\text{Set} \rightarrow \text{Set}) \rightarrow (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Set} \]

can be solved by instantiating \( \lambda_1 \) with the term \( [x]x \). But then we see that the first occurrence of \( \lambda \) has expected type \( \text{Set} \rightarrow \text{Set} \) and the second \( \text{Bool} \rightarrow \text{Bool} \) which contradicts the condition for uniqueness of the expected type imposed on the placeholders.

Similar problem occurs if we are not careful about the local contexts. Consider for example

\[ g : (A \rightarrow A) \rightarrow A \rightarrow \text{Set} \]

and the problem

\[ g([x]\lambda_2, \lambda_2) : \text{Set} \]

in the empty context. Considering the first occurrence of \( \lambda_2 \) which is in context \([x : A]\), we may decide to solve \( \lambda_2 \) by \( x \) obtaining

\[ g([x]x, x) : \text{Set} \]

which is not typable in the empty context, because \( x \) occurs as a free variable.

ALF solves these problems by disallowing the introduction of multiple occurrences of the same placeholder by the user. Possible duplications that may appear in the process of unification will then refer to the "major" occurrence when the placeholder was introduced and must therefore have the same type and same context (or extension of it).

The typechecking of complete terms is done by an algorithm which has two main phases [18]. In the first part a list of equations is being generated and in the second we check if these equations hold by simplifying them. These equations involve two terms of given type and in given context. Magnusson [18] shows that if \( \alpha \) is a proper type in a context \( \Gamma \) then \( \Gamma \vdash e : \alpha \) holds if and only if the generated list of equations \( GE(e, \alpha, \Gamma) \) holds. The second part of the typechecking algorithm takes a list of equations and simplifies them, eliminating the trivial ones. It turns out that the list of equations holds if and only if they
can all be eliminated by this procedure and this gives us a criterion whether or
not the judgement we wanted to typecheck holds.

The list of term equations is generated in two steps. First we generate a list of
type equations which are then simplified into term equations in the second step.
In both cases simplification means application of the appropriate conversion
rules of the calculus. Note that this requires the calculus to be normalizing.

We have to point out that as result of the lack of type annotations of the
bound variables, terms containing a β-redex cannot be typechecked as we have
to be able to compute the type of the head of an application. This is impossible
since the type inference problem in this system is not decidable. Therefore the
typechecking algorithm works only for terms which are in normal form. This
is not a big theoretical problem, since the calculus is (assumed to be) weakly
normalizing, but it may affect the performance of an implementation.

The reason why the typechecking algorithm is split in two phases and the
reason to use lists of equations becomes evident when we try to
typecheck incomplete terms. In this case the first phase produces a list of declarations and
typing constraints which are of the form

\[ ?_n : \alpha_n \Gamma_n \]
\[ \alpha_1 = \alpha_2 \Gamma \]

and the equations may involve placeholders. A placeholder declared in the
list can be used in the declaration of other placeholders or in equations which
appear after the declaration of the placeholder. This ensures non-circularity in
the dependencies between placeholders.

There are no special derivation rules for placeholders. Instead, when first a
placeholder is encountered, its context and expected type are computed and a
declaration for it is output in the list. If later, due to dependencies, the type of
the same placeholder needs to be determined, it can be looked up in the list.

This list constitutes a high-order typed unification problem and as we are in
dependently-typed setting there is no algorithm to solve it in general. Neverthe­
less, we may try to find partial solution. The equations that are left unsolved
are constraints on the future instantiations of the placeholders.

The set of placeholder declarations in a list is closely related to the signatures
in Allc (see section 2.4) or proof-problems in TypeLab (see section 2.2).

The corresponding correctness property for incomplete terms states that if
\( \mathcal{C} \) is the list of equations generated for \( \Gamma \vdash e : \alpha \) then for any instantiation \( \sigma \) of
the placeholders of \( \mathcal{C} \) by complete terms

\[ \Gamma \sigma \vdash e \sigma \vdash \alpha \sigma \text{ if and only if } \mathcal{C} \sigma \text{ holds} \]

This guarantees that the resulting term at the end of the session will be type-
correct and we need not typecheck it.

The meta-theoretical properties of the system are not studied in [18] and
the results obtained are relative to some meta-theory assumptions such as that
constructors are one-to-one, that equal ground terms have the same outermost
constructor and that computation on provably equal typable terms terminates
and leads to equal normal forms.
2.1.3 The Transitivity Example

Let us consider how the transitivity example can be represented in ALF. Suppose we have defined $A$, $R$, $a$ and $c$ of their appropriate types and that we have a term $\text{trans}$ of type $(x,y,z:A) \rightarrow (R x y) \rightarrow (R y z) \rightarrow (R x z)$. Then our goal would be to create a term of type $(R \ a \ c)$:

$\text{theorem-aRb} :: \ (R \ a \ c) = ?$

We specify that we want to use $\text{trans}$:

$\text{theorem-aRb} :: \ (R \ a \ c) = \text{trans} \ ?2 \ c \ ?4 \ ?5$

Now we have 3 goals: to find a witness $?2$ of type $A$ and to prove that $(R \ a \ ?2)$ and $(R \ ?2 \ c)$. At this point we may choose to provide the witness directly, or try to prove the two other goals. This may be a good idea if we cannot directly see a suitable candidate for $?2$. Imagine for example that somehow we manage to prove $(R \ a \ b)$ for some $b$. Then we can instantiate $?2$ and only the goal $(R \ b \ c)$ will remain.

On the level of the theory of ALF the first state is represented by a list of equations which is roughly

...  
[?0 : (R a c) f]

where $\Gamma$ is a context declaring the appropriate variables. When we refine $?0$ by $\text{trans}$ and use the witnesses $a$ and $c$, we get approximately

...  
[?2 : A f]  
[?4 : (R a ?2) f]  
[?5 : (R ?2 c) f]  
[?0 = \text{trans} \ a \ ?2 \ c \ ?4 \ f \ ?5]  

At a later stage, if we succeed in proving $p : (R \ a \ b)$ for some $b$, we can solve $?4$ by $p$:

...  
[?2 : A f]  
[?5 : (R ?2 c) f]  
[?0 = \text{trans} \ a \ ?2 \ c \ p \ ?5 \ f]  
[(R a ?2) = (R a b) f]

which leads to adding the last condition. By simplifying it we can (automatically) infer $?2=b$.

2.1.4 Classification Summary: ALF

<table>
<thead>
<tr>
<th>Properties</th>
<th>MTA</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>MTA</td>
<td>Uses MV</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>?</td>
<td>Explicit Subst.</td>
</tr>
<tr>
<td>Typechecking</td>
<td>decidable</td>
<td>yes</td>
</tr>
<tr>
<td>Type inference</td>
<td>not decidable</td>
<td>yes</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>MTA</td>
<td>Based on</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>no</td>
<td>Martin-Löf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>monomorphic TT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>de Bruijn Ind.</td>
</tr>
</tbody>
</table>

(MTA – Meta-Theory Assumption)
2.2 Typelab

The Typelab system is a software tool based on the theory described in [25, 24]. It was designed as a tool aimed at specification and refinement of programs. Specifications are represented in type theory as dependently typed Σ-types and finding programs meeting these specifications is reduced to finding inhabitants of the corresponding Σ-type.

The target system in Typelab is the Extended Calculus of Constructions or ECC [17]. ECC is an extension of the Calculus of Constructions [7] with dependent Σ-types and universes. This means that Typelab employs a dependently-typed language with metavariables, explicit substitutions and dependent product and Σ-types. It has an infinite hierarchy of type universes \( \text{Prop} : \text{Type}_0 : \text{Type}_1 : \ldots : \text{Type}_n : \ldots \).

2.2.1 Description of the Calculus

The terms of the language are given by the following grammar:

\[
\begin{align*}
\text{Term} &::= \text{Var} \mid \text{Prop} \mid \text{Type} \\
&\mid \text{IVar} : \text{Term} \mid \lambda \text{Var} : \text{Term} \mid \text{Term} \\
&\mid \Sigma \text{Var} : \text{Term} \mid (\text{Term} \cdot \text{Term}) \\
&\mid \text{pair} \text{Term}(\text{Term}, \text{Term}) \mid \pi_1(\text{Term}) \mid \pi_2(\text{Term}) \\
&\mid \text{MVar} \text{Subst} \\
\text{Subst} &::= \text{[[Var := Term] :: Subst}
\end{align*}
\]

where \(\text{Var}\) is the set of variables, \(\text{MVar}\) is the set of metavariables and :: denotes concatenation.

Note that in contrast to other systems based on simpler type theories, here we do not distinguish between terms and types on syntax level.

In Typelab open terms are represented by means of terms which contain metavariables. These metavariables are intended to be placeholders or "holes" for terms that will be constructed at a later stage. Each metavariable has attached to it explicit substitution \(\sigma\). Actually, metavariables without substitutions are not valid terms, but we will identify a metavariable with an empty substitution with the metavariable itself.

We will also assume that we are given a function \(\text{svars} : \text{MVar} \rightarrow \text{List(Var)}\) which assigns a list of variables to each metavariable. The intended meaning is that \(\text{svars}(?n)\) is the list of variables which may be substituted in \(?n\). In other words, this is the list of the variables that may occur free in a term that will instantiate the metavariable. So, if \(\text{svars}(?n) = \langle x, y \rangle\), then \(\lambda z.x\) is eligible to be considered as a candidate to instantiate \(?n\), but \(\lambda z.zx\) is not.

In order to prevent uniqueness problems for the normal forms, we assume a linear order on the variables as given by \(\text{svars}\) and require that the variables in an explicit substitution be ordered respectively. For example, if \(\text{svars}(?n) = \langle x, y \rangle\), then \(?n[x := M, y := N]\) is a well-formed term, but \(?n[y := N, x := M]\) is not. Note that the order may be different for different metavariables.
As we have already mentioned, Typelab is using dependent types and therefore according to the syntax of the calculus metavariables are allowed to appear in types. But then they may be involved in computations by the conversion rule:

\[
\begin{align*}
\Gamma \vdash M : A & \quad \Gamma \vdash B : Type_i & \quad A \equiv B \\
\Gamma \vdash M : B
\end{align*}
\]

(conv)

As we have already noticed, computations with open terms are not so easy:

\[
\begin{align*}
\beta
\end{align*}
\]

It is clear that the substitution generated by the \(\beta\)-reduction should not be ignored, since this leads to a term \(x\) in which the variable \(x\) has escaped its scope and is now free. The solution to this problem is the introduction of explicit substitutions attached to the metavariables: \(?n[x := t_1, y := v]\). This requires additional clauses in the definition of normal (external) substitutions dealing with the case of metavariables.

We still have the usual definition of \(\beta\)-reduction in terms of (external) substitution denoted by curly brackets (\(\sigma = \{x := s\}\)). We need to extend the definition of this metaoperation with the case when it is applied to a metavariable. So \(?n[y_1 := t_1, \ldots, y_k := t_k]\{x := s\}\) is defined as:

\[
\begin{align*}
\text{If } x \in \text{svars}(?n), x \notin \{y_1, \ldots, y_k\} \text{ and } x \text{ is between } y_i \text{ and } y_{i+1} \text{ in } \text{svars}(?n):\
?n[y_1 := t_1\sigma, \ldots, y_i := t_i\sigma, x := s, y_{i+1} := t_{i+1}\sigma, \ldots, y_k := t_k\sigma]
\end{align*}
\]

\[
\begin{align*}
\text{If } x \notin \text{svars}(?n):
?n[y_1 := t_1\sigma, \ldots, y_k := t_k\sigma]
\end{align*}
\]

This means that in effect computations are not completely internalized. The meta-part corresponds to the usual meta-operation of substitution and the explicit substitutions record the metasubstitutions which reach metavariables.

The calculus has the following two base reduction relations:

\(\beta\)-reduction:

\[
(\lambda x : T.M)N \rightarrow_\beta M\{x := N\}
\]

\(\pi\)-reduction:

\[
\pi_i(pair_T(M_1, M_2)) \rightarrow_\pi M_i
\]

The compatible, reflexive and transitive closure of these two relations is denoted by \(\rightarrow\). This relation is shown to have the Church-Rosser property. It gives rise to a corresponding covertibility = relation, which is further extended to the cumulativity relation \(\leq\) which is transitive, symmetric and furthermore:

\[^2\text{The case } x \in \{y_1, \ldots, y_k\} \text{ is not possible if we conform with the usual variable conventions.}\]
Prop \leq T_{type_0} and T_{type_j} \leq T_{type_{j+1}}

If \( A_1 = A_2 \) and \( B_1 \leq B_2 \) then \( \Pi x : A_1 B_1 \leq \Pi x : A_2 B_2 \)

If \( A_1 \leq A_2 \) and \( B_1 \leq B_2 \) then \( \Sigma x : A_1 B_1 \leq \Sigma x : A_2 B_2 \)

The typing rules of the calculus are an extension of those of ECC and include three new rules for typing expressions with metavariables. When we consider typing of metavariables, we always have in mind some proof problem.

A proof problem \( P \) is a triple \( \langle M_P, ctx_P, type_P \rangle \) where \( M_P \) is a finite set of metavariables, \( ctx_P \) is a function which for every \(?n\) in \( M_P \) produces a context such that \( dom(ctx(?n)) = svars(?n) \). The function \( type_P \) assigns a term to each metavariable in \( M_P \). We will drop the index \( P \) when this does not lead to confusion.

Put differently, \( P = \langle \{?1, \ldots, ?k\}, \{\Gamma_1, \ldots, \Gamma_k\}, \{T_1, \ldots, T_k\} \rangle \) means that we are looking for \( k \) terms \(?1, \ldots, ?k\) and each \(?i\) in context \( \Gamma_i \) should have type \( T_i \). It is allowed for some \( T_i \) or \( \Gamma_i \) to contain \(?j\) in case this does not lead to circular dependencies between the metavariables. More precisely, the relation defined by \(?i <?j\) if and only if \(?i, \in MV(T_j) \cup (\Gamma_j)\) should be an irreflexive partial order. For a fixed proof problem \( P \) this is a finite relation and consequently this question is decidable.

The typing rules involving metavariables in Typelab are:

\[
\begin{align*}
\text{ctxt} (?n) &\vdash type (?n) : T_{type_j} & \text{(MV-base)} \\
\text{ctxt} (?n) &\vdash ?n_1 : type (?n) \\
\end{align*}
\]

If \( z \notin FV(\Gamma) \cup FV(\Delta) \cup dom(\sigma) \):

\[
\begin{align*}
\Gamma &\vdash T : T_{type_j} & \text{(MV-weak)} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, z : T, \Delta \vdash ?n \sigma : N \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, \Delta \vdash ?n \sigma : N \\
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash t : T & \text{MV-\beta-Red} \\
\Gamma, x : T, \Delta \vdash ?n \sigma : N \\
\end{align*}
\]

For this typing system one can show that the following properties hold:

**Principal Type** If \( M \) is well-typed in a context \( \Gamma \) then there exists a type \( A \) (principal type) such that \( \Gamma \vdash M : A \) and for every \( A' \)

\[
\Gamma \vdash M : A' \text{ if and only if } A \leq A' \text{ and } A' \text{ is type in } \Gamma
\]

**Subject Reduction** If \( \Gamma \vdash M : A \) and \( M \rightarrow N \) then \( \Gamma \vdash N : A \)

**Decidability** Type inference and hence type checking (using the Principal Type property) are decidable.

**Strong normalization** The calculus is strongly normalizing.

The proof construction proceeds by creating metavariables for unknown terms and solving them by means of instantiations. Instantiations are defined in such way that guarantees the well-typedness of the resulting proof problem. This on its behalf is the basis for the soundness of the proof method.
2.2.2 The Transitivity Example

Our transitivity example can be expressed also in TypeLab. Suppose \( \Gamma \) is a context containing the declarations for a set \( A \), a binary relation \( R \), two elements \( a \) and \( c \) of \( A \) and the assumption \( \text{trans} \) that \( R \) is transitive:

\[
\Gamma = \langle A : \text{Type}, R : A \rightarrow A \rightarrow \text{Prop}, a : A, c : A, \text{trans} : \forall x, y, z : A. xRy \rightarrow yRz \rightarrow xRz \rangle
\]

In this context we want to prove \( aRc \):

\[
\Gamma \vdash ?_1 : xRy
\]

Using \( \text{trans} \) we obtain three goals\(^3\)

\[
\begin{align*}
\Gamma & \vdash ?_2 : A \\
\Gamma & \vdash ?_3 : aR?_2 \\
\Gamma & \vdash ?_4 : ?_2Rc
\end{align*}
\]

which we are free to solve in any order we want.

2.2.3 Classification Summary: Typelab

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>yes</td>
</tr>
<tr>
<td>Type checking</td>
<td>decidable</td>
</tr>
<tr>
<td>Type inference</td>
<td>decidable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Uses MV</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit Subst.</td>
<td>yes</td>
</tr>
<tr>
<td>Based on</td>
<td>Extended Calculus</td>
</tr>
<tr>
<td>de Bruijn Ind.</td>
<td>no</td>
</tr>
</tbody>
</table>

2.3 OLEG

OLEG [19] is a calculus of open terms for the ECC with local definitions but without \( \Sigma \)-types. It consists of two layers - OLEG core and OLEG development calculus. The core is essentially ECC with definitions and the development calculus is wrapped around it adding metavariable support.

OLEG was introduced as a tool to study dependently typed functional programs and is partly implemented as an extension of the proof-assistant LEGO [23].

The OLEG calculus has a couple of features which distinguish it from the rest of the calculi that we consider here. The first one is the introduction of explicit binders for holes which allow a metavariable to be treated as a normal one in many respects. The second is the way OLEG handles computations (reductions) involving metavariables. Instead of using functional coding or explicit substitution, OLEG forbids occurrences of metavariable declarations at trouble spots.

Let us look more closely at the calculus:

\(^3\)To be precise these three goals form the following proof problem: \( P = \langle (?_2, ?_3, ?_4), (\Gamma, \Gamma), (A, aR?_2, ?_2Rc) \rangle \).
2.3.1 The syntax

The syntax and typing of OLEG are given by a number of rules. We will not present all of them here, they can be found in [19]. A core OLEG term can be

- Variable \( x \)
- Application \( MN \)
- Abstraction \( \lambda x : T.M \)
- Dependent Type \( \forall x : T.M \)
- Definition \( !_x = M : T.N \)

There are three contraction schemes:

(β) \( \Gamma \vdash (\lambda x : T.N)M \rightarrow_\beta !_x = M : T.N \)

(δ) \( \Gamma, x = M : T, \Delta \vdash x \rightarrow_\delta M \)

(!) \( \Gamma \vdash !_x = M : T.N \rightarrow_! N \) if \( x \not\in \text{FV}(N) \)

Note that (β) does not involve externally defined substitution, it just converts a \( \beta \)-redex into a \( \delta \)-redex:

\[
(\lambda x : T.N)M \rightarrow_\beta !_x = M : T.N \rightarrow_\delta \ldots \rightarrow_\delta !_x = M.N[x/M] \rightarrow_! N[x/M]
\]

The core calculus enjoys the usual properties of ECC as Church-Rosser, subject reduction, strong normalization and cut (if \( \Gamma, !_x = s : S, \Delta \vdash M : A \) then \( \Gamma, \Delta \{ x := s \} \vdash M \{ x := s \} : A \{ x := s \} \)). The typing rules for the core calculus mimic those for ECC, with substitution replaced by appropriate definitions, for example:

\[
\begin{align*}
(\text{app}) & \quad \Gamma \vdash F : \forall x : S.T & \quad \Gamma \vdash t : S \\
\hline
\Gamma \vdash Ft : !x = t : S.T
\end{align*}
\]

The development calculus is an extension of the core calculus with open terms. The term language is extended with a new binder for metavariables "?". So, if \( M \) is a term, then \( ?x : T.M \) is also a term. The binder declares a metavariable and this metavariable can be used as a normal variable in its scope: \( \lambda f : A.?x : B.fx \). Another extension of the term language is the introduction of "guesses" which represent a term that may possibly be a solution of a metavariable: \(?x \approx P : S.M \). The \( \approx \) sign is used to emphasize that this expression cannot be treated as a definition and \( x \) cannot be replaced by \( P \) in \( M \).

The term construction process goes like this: first we introduce a metavariable of the appropriate type \( ?x : A.M \) then we try to solve \( x \) by \( P_1 \) resulting in \(?x \approx P_1.M \). However, this may be a wrong guess, so we can change it to \( P_2 \) and so on, until we have found the right one \(?x \approx P.M \). Once this has happened, we can turn the guess into local definition \(!x = P.M \). All these operations can be done by means of appropriate admissible rules.
For every metavariable we have to be able to uniquely identify its context and type. The type is explicitly given in its declaration and the context can be determined by the position of its declaration (i.e. the context is formed by all variables in whose scope the declaration of the metavariable).

The treatment of metavariables in such a way is not enough to avoid problems. For example it is quite possible to obtain terms like \( \lambda f : (\forall x : A.x).f \) whose typability is problematic, especially if we want to be able to solve metavariables in arbitrary order and to have commuting hole filling and computation.

McBride in [19] relates those problems to metavariable declarations leaking into types and imposes restrictions on the system which guarantee that declarations of metavariables will not occur in the types involved in a derivable judgement.

These restrictions are strong enough to ensure that the resulting development calculus has the replacement property which states that a guess of given type can safely be replaced by any other of the same type; a property that is very useful in applications (for precise formulation and proof, cf. [19]).

The rules involving metavariable declarations are given below:

(declare) \[
\Delta \vdash valid \quad \Delta \vdash T : Type_j \\
\Delta, ?x : T \vdash valid
\]

(construct) \[
\Delta \vdash p : T \\
\Delta, ?x \approx p : T \vdash valid
\]

(hole) \[
\Delta, ?x : S \vdash p : T \\
\Delta \vdash ?x : S.p : T \quad \text{if} \ x \not\in T
\]

(guess) \[
\Delta, ?x \approx q : S \vdash p : T \\
\Delta \vdash ?x \approx q : S.p : T \quad \text{if} \ x \not\in T
\]

Note the side conditions in (hole) and (guess). They disallow metavariable bounded in a term to occur in its type. If this condition were not present, we would have to accept metavariable declarations in types.

The side conditions in these two typing rules are not enough to guarantee that \(?\)-binders do not appear in types. This is achieved by disallowing applications and \(!\)-bounded values containing \(\exists\)-binders (i.e. if \(\!x = s : S.M \) and \(fs\) are terms, then \(s\) is not allowed to contain \(\exists\)-binders) and removing \(\beta\) and \(\delta\) reduction from the contraction schemes for partial constructions. Using the embedding of core terms into the development calculus the system is still allowed to do computations on expressions with metavariables (declared in the context), but computations on terms with \(\exists\)-binders are disallowed.

The lack of \(\delta\)-reduction on terms with \(\exists\) binders prevents us from getting inconsistencies like:

\[
\begin{array}{l}
\forall x : A. \ ?n : A. \ n \quad \ ?n := x \\
\forall x : A. \ ?n := x
\end{array}
\]

17
Of course, a term may exploit metavariables in the context, but then they are outside the scope of the \( i \)-binder and cause no trouble:

\[ x \text{ is not in the scope of } n \]

\[ \ldots, ?n : B, \ldots \rightarrow \delta t = \vdash _{B, x n} \delta t \]

\[ \ldots, ?n : B, \ldots \rightarrow \bot n \]

Although the restrictions imposed on the use of metavariables in OLEG are very strict, the calculus is still expressive enough to capture all the cases in the intended use of metavariables in the context of [19].

The metatheoretic properties of the core are also present in the development calculus: confluence, Subject Reduction, Cut, Strong Normalization, etc.

### 2.3.2 The Transitivity Example

The declarations we need for introducing the problem, using the notation for states introduced in section 2.4 of [19] are:

\[
\lambda \ A : Type \\
\lambda \ R : A \rightarrow A \rightarrow Prop \\
\lambda \ a, c : A \\
\lambda \ trans : \forall x, y, z : A . (Rxy) \rightarrow (Ryz) \rightarrow (Rxz) \\
?X_{1} : (Rac)
\]

Then refining \( X_{1} \) by \( trans \) and filling in \( a \) and \( c \) for \( x \) and \( z \) we get

\[
\ldots \\
?X_{1} \Rightarrow ?y : A \\
?r_{1} : (R a y) \\
?r_{2} : (R y c) \\
trans a y c r_{1} r_{2} \\
(R a c)
\]

and we are free to solve \( r_{1} \) or \( r_{2} \) first.

### 2.3.3 Classification Summary: OLEG

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>yes</td>
</tr>
<tr>
<td>Typechecking</td>
<td>decidable</td>
</tr>
<tr>
<td>Type inference</td>
<td>decidable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Uses MV</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit Subst.</td>
<td>no</td>
</tr>
<tr>
<td>local definitions</td>
<td>Based on</td>
</tr>
<tr>
<td>Extended Calculus</td>
<td>de Bruijn Ind.</td>
</tr>
<tr>
<td>of Constructions</td>
<td>no</td>
</tr>
</tbody>
</table>

18
2.4 The $\lambda\Pi_E$-calculus

The $\lambda\Pi_E$-calculus is a calculus of explicit substitutions using de Bruijn indices. It has dependent types and the metavariables and the substitutions are first-class objects in the calculus. Its origins can be traced back to $\lambda\sigma$, but it essentially extends it, since adding support for dependent types is not straightforward. $\lambda\Pi_E$ is based on dependently typed lambda calculus $\lambda\Pi$. It has a version for the stronger Calculus of Constructions called $CC_E$, but the solutions to the main problems can be found in $\lambda\Pi_E$.

2.4.1 Description of $\lambda\Pi_E$

The well-formed expressions in $\lambda\Pi_E$ are given by the following grammar:

<table>
<thead>
<tr>
<th>natural numbers</th>
<th>$n$</th>
<th>::=</th>
<th>0$n$ + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>metavariables</td>
<td>$M$</td>
<td>::=</td>
<td>$X[Y]$...</td>
</tr>
<tr>
<td>terms</td>
<td>$A, B, M, N$</td>
<td>::=</td>
<td>$\text{Kind}[\text{Type}][1] \lambda_A.M[MN][MN][M][S] M$</td>
</tr>
<tr>
<td>substitutions</td>
<td>$S, T$</td>
<td>::=</td>
<td>$\uparrow^n [M \cdot A S] S \circ T$</td>
</tr>
</tbody>
</table>

It has the following reduction rules:

- **(Beta)**: $(\lambda_A.M)N \rightarrow M[N \cdot A \uparrow^0]$
- **(Lambda)**: $(\lambda_A.M)[S] \rightarrow \lambda_M[S].(M[1 \cdot A (S \circ \uparrow^1)])$
- **(Pi)**: $(\Pi_A.B)[S] \rightarrow \Pi_A[S].(B[1 \cdot A (S \circ \uparrow^1)])$
- **(App)**: $(MN)[S] \rightarrow (M[S]N[S])$
- **(Clos)**: $(M[S])[T] \rightarrow M[S \circ T]$
- **(VarCons)**: $1[M \cdot A S] \rightarrow M$
- **(Id)**: $M[\uparrow^0] \rightarrow M$
- **(Map)**: $(M \cdot A S) \circ T \rightarrow M[T] \cdot (S \circ T)$
- **(IdS)**: $\uparrow^n \circ S \rightarrow S$
- **(ShiftCons)**: $\uparrow^{n+1} o (M \cdot A S) \rightarrow \uparrow^n o S$
- **(ShiftShift)**: $\uparrow^{n+1} o \uparrow^n \rightarrow \uparrow^n o \uparrow^{n+1}$
- **(Shift0)**: $1.A \uparrow^1 \rightarrow \uparrow^0$
- **(ShiftS)**: $1[\uparrow^n]A \uparrow^{n+1} \rightarrow \uparrow^n$
- **(Type)**: $\text{Type}[S] \rightarrow \text{Type}$

In order to be able to type metavariables, judgements in the system are extended to contain signatures (denoted by $\Sigma$) which are lists of metavariable declarations: $\Sigma; \Gamma \vdash M : A$. The judgements expressing the validity of a signature have the form $\vdash \Sigma$, the judgement expressing that a context $\Gamma$ is valid in a signature $\Sigma$ is $\vdash \Sigma; \Gamma$. The type $A_i$ and the context $\Gamma_i$ of a metavariable $X_i$ in a signature $\Sigma = (X_1 : \Gamma_1, A_1, \ldots, X_n : \Gamma_n, A_n)$ may depend only on metavariables with index $j$ where $i < j \leq n$. The rules for the metavariables are the following:

$\vdash \Sigma; \Gamma \vdash A : s$

$\vdash \Sigma; \Gamma \vdash \text{X-fresh}$

$\vdash (X : s \cdot A).\Sigma$

$\vdash \Sigma; \Gamma \vdash X : A$

$\vdash \Sigma; \Gamma \vdash \Delta = \lambda\Pi_E$ $\Gamma$

$\vdash \Sigma; \Gamma \vdash X : A \in \Sigma$
The full set of rules can be found in [21]. Since substitutions by themselves are objects in the calculus, they can be typed and the type of a substitution is a context.

One specific feature of $\lambda\Pi_C$ is that the substitutions are annotated with types. These annotations are created by the rules (Beta), (Lambda) and (Pi) and are preserved during the life-cycle of the substitution.

There are however some subtle details that need attention. Bloo [5] presents an example by Geuvers for a term (namely $(Ax : N.\lambda f : (Pz) \rightarrow N.\lambda y : (Pz).fy)z$ in the context $\Gamma_N = N : Type, P : N \rightarrow Type$ and $z : N$) which is typable, but reduces to a term which is not. This is due to the fact that there might be a difference between the type in the declaration of a variable after the substitution and the type of the variable at the place where it is used, but the substitution is still not executed. This happens if we distribute the substitution over the binder for $f$:

$$\lambda f : (Pz) \rightarrow N. (\lambda y : (Pz).fy)\{x := z\}$$

$f$ expects an argument of type $(Pz)$ but since the substitution is not yet propagated over the binder for $y$, the variable $y$ has type $(Pz)$ which is not convertible to $(Pz)$.

This problem is avoided by introducing the type annotations in substitutions. Take for example the (Lambda) rule. Written using names for the variables it reads

$$(\lambda x : A.M)[S] \rightarrow (\lambda x : A[S].M[x := x : A, S])$$

In this way passing through abstraction the substitution is extended by a "dummy" substitution which records the original type of the variable. In this way using the (Clos) derivation rule

$$\frac{\Sigma ; \Gamma \vdash S : \Delta \quad \Sigma ; \Delta \vdash M : A \quad \Sigma ; \Delta \vdash A : s}{\Sigma ; \Gamma \vdash M[S] : A[S]}$$

the correct types can be computed.

Note that this problem is avoided in Typelab which also has explicit substitutions by allowing them only at metavariables and using the (meta)operation substitution.

The metatheoretic properties of $\lambda\Pi_C$ have been studied and proofs can be found for example in [21]. The calculus is confluent on well-typed terms and has the properties of subject reduction, type uniqueness, weak normalization, etc.

2.4.2 The Transitivity Example

$\lambda\Pi_C$ is strong enough to provide the functionality needed to facilitate the transitivity example. Let$^4 \Gamma$ be the context $trans : \Pi x, y, z : A.(Rx y) \rightarrow$
(Ryz) \rightarrow (Rxz).a : A.c : A.R : A \rightarrow A \rightarrow \text{Type}.A : \text{Type}. Then the goal of proving (Rac) can be expressed by the following judgement:

\( \langle X : \rightarrow (Rac) \rangle; \Gamma \vdash X : (Rac) \)

refining by \textit{trans} results in

\( \langle P_2 : \rightarrow (RYc).P_1 : \rightarrow (RaY).Y \rightarrow A \rangle; \Gamma \vdash \text{trans} a Y c P_1 P_2 : (Rac) \)

2.4.3 Classification Summary: \( \lambda \Pi \mathcal{L} \)

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>yes</td>
</tr>
<tr>
<td>Typechecking</td>
<td>decidable</td>
</tr>
<tr>
<td>Type inference</td>
<td>decidable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>no</td>
</tr>
</tbody>
</table>

Uses MV  yes
Explicit Subst. yes
Based on \( \lambda \Pi \)
\( \text{de Bruijn Ind.} \) yes

21
Chapter 3

Other Calculi of Open Terms

In this chapter we consider several other systems of open terms which are not explicitly developed or suited for theorem proving, but still deserve attention.

3.1 The Context Calculus $\lambda c$

3.1.1 Contexts in untyped $\lambda$-calculus as open terms

In [6] a calculus of (untyped) $\lambda$-calculus contexts is presented. In this section we will use the term context in the sense of this paper. Note that it denotes a different notion, related more to open terms than to contexts as lists of variable declarations.

So, the central notion is context, "an expression with special places called holes, where other expressions can be placed". We will denote the holes by the symbol $\Box$. Examples for contexts are

\[(\lambda x.\Box)z \quad \Box \quad y\Box\]

The main motivation for introducing this calculus is to formalize the notion of context used in many formal discussions just as a notational convention. This formalization has to take into consideration the effect of variable capturing which occurs when the term filling a hole has a free variable which becomes bound by a binder in whose scope the hole is situated. Suppose in the context $(\lambda x.\Box)z$ we want to put the term $xz$. Then the result is

\[(\lambda x.zx)z\]

and the variable $x$ becomes bound.

Another potential difficulty of the communication between a context and the expressions being put in its holes is again the one described in section 1.4.
This means that the informally defined $\lambda$-contexts are not considered modulo $\alpha$-conversion and cannot be subject to $\beta$-reduction which is limiting their use. The $\lambda\sigma$-calculus described below formalizes the notion of context and solves these problems.

3.1.2 Description of the Calculus

In $\lambda\sigma$ we have new constructors: $\Lambda$ for multiple abstraction, $\otimes$ for multiple application, $\circ$ for composition of contexts and also the corresponding multiple (simultaneous) substitution:

$T ::= V$
$\langle \lambda V.T \rangle$
$\langle TT \rangle$
$\langle \Lambda_n V_1, \ldots, V_n.T \rangle$
$\langle \otimes T, T_1, \ldots, T_n \rangle$
$\langle \circ (T, T_1, \ldots, T_n) \rangle$

There are two main operations defined on contexts: hole filling and composition. In the first case we are given a context and we put a term in a hole of the context. This operation is represented by multiple application. In the case of composition we fill the hole with another context.

The system has the following reduction rules:

$(\lambda x.U)V \rightarrow_{\beta} U\{x := V\}$
$\otimes (\Lambda x.U)V \rightarrow_{\beta} U\{x := V\}$

$\circ (\tilde{B}x.U, (\Lambda h^1 Bx^1.V_1), \ldots, (\Lambda h^n Bx^n.V_n)) \rightarrow_{\circ}$
$\rightarrow_{\circ} (\Lambda h^1 Bx^1, \ldots, x^n U \{g_1 := \Lambda h^1.V_1, \ldots, g_n := \Lambda h^n.V_n\})$

where $\tilde{B}s$ are either all sequences of $\lambda$ or all sequences of $\Lambda$ (not necessarily of the same length). Note also the movement of the binders for $\tilde{x}$ outside the scope of the binders for $\tilde{h}$.

Here is an example illustrating the use of the composition rule:

$\circ ((\Lambda g, g'.U), ((\Lambda \tilde{x}h)V), (\Lambda g\Lambda k, k'.W)) \rightarrow \Lambda h, k, k'.U \{g := \Lambda \tilde{x}V, g' := \Lambda g'.W\}$

The main result in [6] states that this calculus is confluent.

It is a framework for representing different kinds of contexts. The exact structure of the contexts can be fine-tuned using pretyping. Here we will discuss the example in [6] where pretyping yielding untyped $\lambda$-contexts is presented.

In the untyped $\lambda$-calculus the terms and contexts are given by the following rules:

$M ::= x | \lambda x.M | MM$
$C ::= x | \square | \lambda x.C | CC$

23
with the assumption that each occurrence of $\Box$ is different and the holes are ordered from left to right.

The translation of these terms into $\lambda c$ is just the identity and the translation for contexts is defined as follows:

$$|C| = \Delta h.C[[\Box h_1, x^1], \ldots, (\Box h_n, x^n)]$$

where $C[M_1, \ldots, M_n]$ means the context $C$ with its first hole filled by $M_1$, the second by $M_2$, etc. The vector $x^i$ contains all variables in whose scope is the $i$th hole. This means that a hole $\Box$ in a $\lambda$-context is represented by a hole variable $h$ applied to all the variables that may occur free in the hole $\Box h x_1, \ldots, x_n$. In this way, any subsequent substitution passing through the hole will be “remembered” because the appropriate variable would be substituted in the argument list.

We can extend this translation to the hole-filling operation $hf(C, M)$ which fills the holes of $C$ by the terms $M$ and the composition operation $comp(C, D)$ which fills the holes of $C$ with other contexts from $D$. One has to distinguish between $hf(C, M)$ and $C[M]$ as the first denotes the operation and the second denotes the resulting term. The same holds for $comp(C, D)$ and $C[D]$.

$$|hf(C, D)| = \Box |C|, \Delta x^1, |D_1|, \ldots, \Delta x^n, |D_n|$$

where $x^i$ is the list of bound variables in whose scope is the $i$th hole of $C$. The translation of the composition is not so straightforward, because the holes in $D$ have to be “lifted”:

$$|comp(C, D)| = o(|C|, (\Delta x^1.LIFT(|D_1|, x^1)), \ldots, (\Delta x^n.LIFT(|D_n|, x^n)))$$

$$LIFT(D, x) = \Delta y.(\Box D, y^1, (\Box g_1, x^1), \ldots, (\Box g_n, x^n), y^n)$$

where $y^i$ is the list of the bound variables in whose scope is the $i$th hole of $D$.

The pre-typing rules given in [6] guarantee the correctness of this translation of untyped $\lambda$-contexts into $\lambda c$. The derivation system has only one base type $t$ for terms, hole variables have type $[t]$, contexts have types $[t] X \ldots X [t] = t$ and contexts, which are to be put in the holes of another context have type $t([t] X \ldots X [t] = t)$. The derivation rules impose discipline on the term formations so that the above translation can be implemented. The resulting calculus $\lambda c'$ has the reduction rules of $\lambda c$, but the set of terms is limited to the well-pretyped terms of $\lambda c$.

Since $\lambda c$ was not aimed at theorem proving and the pre-typing is only used to separate the well-formed terms and does not support dependent types, we have no way to express the problem in the transitivity example in $\lambda c$.
3.1.3 Classification Summary: $\lambda c$

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>not applicable</td>
</tr>
<tr>
<td>Typechecking</td>
<td>not applicable</td>
</tr>
<tr>
<td>Type inference</td>
<td>not applicable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>?</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>?</td>
</tr>
<tr>
<td>Uses MV</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit Subst.</td>
<td>Functional encoding</td>
</tr>
<tr>
<td>Based on</td>
<td>Untyped</td>
</tr>
<tr>
<td>de Bruijn Ind.</td>
<td>no</td>
</tr>
</tbody>
</table>

3.2 Automath

3.2.1 The calculus of segments

The calculus was introduced by de Bruijn [9] and later further studied in [2]. It provides facilities to introduce *abbreviations for parts of terms* containing a hole which can be filled with another term. Let us consider a small example:\(^1\):

\[
\lambda y. x
\]

This diagram is a tree representation of the term $\lambda x.\lambda y.\lambda z.\lambda y. x y$. (In the diagram $\delta$ denotes application. Note also the order of its arguments). Using segments we are able to abbreviate expressions like $\lambda x.\lambda y. o$ obtaining:

\[
\lambda y. x
\]

which in $\lambda$-calculus notation would be approximately represented by the term $(\lambda \sigma.\sigma(\sigma(xy)))(\lambda x.\lambda y.\square)$.

There are several things that at first look strange and we need to explain them: First, the expression $\lambda x.\lambda y. \omega$ represents an abbreviation and the symbol $\omega$ denotes the place where we need to plug in a term in order to construct a valid instantiation of the segment. One segment may only have one open end denoted by $\omega$. It can only be at the far right end of the tree representation of the segment also known as the "heart" position [10].

An abbreviated expression may be used by means of a special variable called segment variable (in this case $\sigma$). In contrast to normal variables, segment variables do not appear on the leaves, but inside the tree. Segment variables stand for a segment. The binding between the segment and the segment variable is accomplished by an artificially created redex $(\lambda \sigma \ldots)(\lambda x.\lambda y.\omega)$.

Second, there is a problem with the bound variables: in the original expression the variables $x$ and $y$ are bound and in the abbreviated expression they

\(^1\)In [9] and [2] the system is presented as a namefree calculus, but here we will illustrate it using named variables.
seem to be outside the scope of the binders. This situation is resolved by adding annotations to the segment variables:

\[
\begin{array}{c}
\lambda_x \rightarrow \lambda_y \rightarrow \omega \\
\delta \rightarrow \lambda_x \rightarrow \sigma(x_1,y_1) \rightarrow \sigma(x_2,y_2) \rightleftharpoons \delta \rightarrow x_2
\end{array}
\]

Things now seem to be looking fine, but there is a third problem, hidden in the \(\beta\)-reduction. Of course we want our abbreviations to really behave like ones and not affect \(\beta\)-reduction which means that the unfolding of abbreviations (which is in fact done by a sort of \(\beta\)-reduction) must commute with the execution of substitutions.

The problem lies in the fact that as a result of a \(\beta\)-reduction we may need to apply a substitution to a segment. Then we have to define how the substitution works on the \(\omega\)-terminator at the end of a segment. We adapt an example from [2]:

\[
\begin{array}{c}
\lambda_x \rightarrow \delta \rightarrow \lambda_y \rightarrow \lambda_z \rightarrow \omega \\
\delta \rightarrow \lambda_x \rightarrow \sigma(x,y,z) \rightarrow y
\end{array}
\]

We see that there is a redex \((\lambda y \ldots) u\). If we decide to reduce it, we need to propagate the substitution of \(u\) for \(y\) in \(\lambda z.\omega\). This confronts us with two problems. First, how does the substitution work on \(\omega\) and second, what does the variable \(y\) in the main branch of the tree refer to after the reduction. The naive definition of substitution which does nothing on \(\omega\) leads to:

\[
\begin{array}{c}
\lambda_x \rightarrow \lambda_z \rightarrow \omega \\
\delta \rightarrow \lambda_x \rightarrow \sigma(x,y,z) \rightarrow y
\end{array}
\]

And this is of course unacceptable as we can see after unfolding the abbreviation. Instead, when we reach \(\omega\), we create a redex which “remembers” the substitution, effectively postponing it until the abbreviation is unfolded. This however leads to reordering of the binders in the segment. Consequently we need to reorder the variable references in the segment variable\(^2\):

\[
\begin{array}{c}
\lambda_x \rightarrow \lambda_z \rightarrow \delta \rightarrow \lambda_y \rightarrow \omega \\
\delta \rightarrow \lambda_x \rightarrow \sigma(x,z,y) \rightarrow y
\end{array}
\]

\(^2\)At first sight this means that substitution is not a local operation anymore because we need to update the segment variables. This is however avoided by explicitly recording a permutation of the variables in the \(\omega\) instead of the segment variable occurrence and applying this permutation to the variable list later at the moment of unfolding.
3.2.2 Properties and Extensions of the System

In [2] several properties of the segment calculus are proved. The most important one is that the calculus is Church-Rosser.

A typed version of the calculus also exists. It is an extension of the simply typed lambda calculus in the Church style. A special sort of types, called \( \rho \)-types are introduced in order to type the segments. For the typed version of the calculus it is proved that under some assumptions, the closure property holds. It states that the type of a term is preserved under \( \beta \)-reduction.

3.2.3 Classification Summary: Automath

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>Uses MV</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>Create Seg. variables</td>
</tr>
<tr>
<td>Typechecking</td>
<td>Explicit Subst.</td>
</tr>
<tr>
<td>Type inference</td>
<td>Based on</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>Untyped</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>de Bruijn Ind.</td>
</tr>
</tbody>
</table>

3.3 The \( \lambda s_e \)-calculus

The \( \lambda s_e \)-calculus [14] was introduced as extension of the \( \lambda s \)-calculus [13, 22] with open terms. This is a calculus of open terms with metavariables and explicit substitutions. It employs nameless variables in the style of de Bruijn. Unfortunately no dependently-typed version of this calculus has been developed so far, which prevents its wider use in theorem proving.

The first thing that separates \( \lambda s \) and \( \lambda s_e \) from the other calculi of explicit substitutions in de Bruijn notation is the way the explicit substitutions are introduced. Instead of creating a new syntactic category of substitutions (compare with section 2.4), this calculus internalizes them by means of two families of operators:

- Explicit substitution operators \( \sigma^j, 1 \leq j \). The expression \( a \sigma^j b \) means the term \( a \) where the free occurrences of the variable with index \( j \) are to be replaced by \( b \).

- Updating functions \( \varphi^j_k, 0 \leq k, 1 \leq i \). These functions are used to update the indices of a term subject to substitution. The intuitive meaning is that \( \varphi^j_k i \) is the term \( i \) in which all indices greater than \( k \) are incremented by \( i - 1 \).

The terms of the system are given by the following grammar:

\[
M, N ::= V[N][\lambda M](\lambda \sigma^j M)(\lambda \rho^i_k M)
\]

where \( V \) is the set of metavariables.
\( \lambda s_e \) is a rewrite system with the following reduction rules:

- **\( \sigma\)-generation**  \((\lambda a)b \rightarrow a\sigma^1b\)
- **\( \sigma\)-\( \lambda\)-transition**  \((\lambda a)\sigma^1b \rightarrow \lambda(\sigma^{i+1}b)\)
- **\( \sigma\)-app-transition**  \((a_1a_2)\sigma^1b \rightarrow (a_1\sigma^ib)(a_2\sigma^ib)\)
- **\( \sigma\)-destruction**  \(n\sigma^ib \rightarrow \begin{cases} n-1 & \text{if } i < n \\ n & \text{if } n \leq i \end{cases}\)
- **\( \varphi\)-\( \lambda\)-transition**  \(\varphi_k^i(\lambda a) \rightarrow \lambda(\varphi_{k+i}a)\)
- **\( \varphi\)-app-transition**  \(\varphi_k^i(a_1a_2) \rightarrow \varphi_k^i(a_1)\varphi_k^i(a_2)\)
- **\( \varphi\)-destruction**  \(\varphi_k^n(n) \rightarrow \begin{cases} n+1 & \text{if } n > k \\ n & \text{if } n \leq k \end{cases}\)
- **\( \sigma\)-\( \sigma\)-transition**  \((\sigma^ib)\sigma^jc \rightarrow (\sigma^{i+1}c)\sigma^j(\sigma^{j+i+1}c)\)
- **\( \sigma\)-\( \varphi\)-transition1**  \(\varphi_k^i(a)\sigma^ib \rightarrow \varphi_k^i(a)\varphi_k^{i+1}b\)
- **\( \sigma\)-\( \varphi\)-transition2**  \(\varphi_k^i(a)\sigma^ib \rightarrow \varphi_k^i(a)\varphi_k^{i+1}b\)
- **\( \varphi\)-\( \sigma\)-transition**  \(\varphi_k^i(\sigma^ib) \rightarrow (\varphi_{k+i}^i(\sigma^ib))\)
- **\( \varphi\)-\( \varphi\)-transition1**  \(\varphi_k^i(\varphi_k^j(a)) \rightarrow \varphi_k^i(\varphi_{k+i-j}^j(a))\)
- **\( \varphi\)-\( \varphi\)-transition2**  \(\varphi_k^i(\varphi_k^j(a)) \rightarrow \varphi_i^{j+1}a\)

Considered as a rewrite system, \( \lambda s_e \) has the following properties:

- **\( \beta\)-reduction** It simulates \( \beta\)-reduction, which means that if \( M \) \( \beta\)-reduces in one step to \( N \) then \( M \) reduces to \( N \) using the rules of \( \lambda s_e \).

- **Confluence** The calculus is confluent on open terms.

The calculus does not preserve strong normalization In [11] Guillaume gives an example for a strongly normalizable \( \lambda\)-term which is not \( \lambda s_e\)-strongly normalizable.

The system \( \lambda s_e \) has been used successfully in areas connected to theorem proving, namely higher order unification [12].

Another interesting feature is that it supports explicit substitutions attached to any subterm (like Typelab for example) which would give efficiency advantages to a theorem proving system based on an eventual extension of \( \lambda s_e \) with dependent types. The results obtained so far for \( \lambda s_e\) compare very well to similar results in systems based on the \( \lambda\sigma\)-calculus, which is the basis for \( \lambda\Pi_2\) which we discussed earlier. Therefore it will not be surprising if an extension of \( \lambda s_e\) soon joins the systems of open terms for theorem proving.
3.3.1 Classification Summary: $\lambda_\alpha$

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>not applicable</td>
</tr>
<tr>
<td>Typechecking</td>
<td>not applicable</td>
</tr>
<tr>
<td>Type inference</td>
<td>not applicable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>PSN fails</td>
</tr>
<tr>
<td>Uses MV</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit Subst.</td>
<td>yes</td>
</tr>
<tr>
<td>Based on</td>
<td>Untyped/Simply-typed</td>
</tr>
<tr>
<td>$\lambda$-calculus</td>
<td>de Bruijn Ind.</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>
Chapter 4

Discussion

In this chapter we want to compare the different systems, to discuss the problems they face and the ways to solve them.

4.1 Common features

Although the systems discussed are quite different from each other, there are several issues on which they agree:

- **Computational effects** It seems that all calculi of open terms must in some way deal with the computational issues of open terms. All systems discussed address the issue of performing a substitution on an open term. Although the approaches are different, they all find some answer to this problem.

- **Representation of holes by (meta)variables** A common trend is to represent unknown terms by (meta)variables. A notable agreement between all the systems is that the main unknown object is a term and therefore metavariables are usually introduced to stand for terms. For some calculi having other syntactic categories (substitutions, types, etc.) metavariables can also be introduced for objects of these categories, but not always successfully.

- **Uniqueness of the type and context of the holes** If allowed, multiple occurrences of a metavariable must agree on their types and contexts. We have seen that in examples where this condition was broken (see e.g. 2.1.2), usually the soundness of the instantiation of the metavariables is compromised.

The uniqueness condition brings into the game some additional structure which describes the contexts and the types of the metavariables. In Typelab this is the proof problem, in OLEG these are the metavariable
binders, in $\lambda\Pi_C$ we have signatures, etc. Except for OLEG where this information is internalized, this external structure parameterizes the whole system - note for example that the type inference rules in Typelab depend on a proof problem that is fixed in advance. Another use of such a structure is to contain the additional information as for example constraints or restrictions on the instantiations of the MVs.

4.2 Open Terms and Computation

As we can see from the descriptions of the different systems, the main issue when introducing open terms is how to handle interactions between them and the computation rules of the system. It seems that the two approaches adopted in the systems discussed are either to avoid troublesome computations or to introduce extra facilities which help solve the problem.

Avoiding troublesome computations in dependently-typed systems is done by imposing restrictions on the reduction and/or the typing rules of the system which guarantee behaviour of the open terms close to the case of simply-typed systems avoiding the problems of dependent types.

On the other hand, systems which need to have a more liberal use of metavariables introduce a mechanism to offset the negative effects of computations on open terms. These negative effects may be loss of confluence or non-commuting of metavariable instantiations and computation.

We can identify two main approaches to achieving this goal:

- Explicit substitutions. They are used to delay the execution of a substitution over an open term, so that it is not lost.

- Functional encoding of the scope. The holes in this case are considered as functions applied to the variables which are allowed to be free in the instantiations of a hole. Then a substitution is recorded by its effect on the arguments of the hole. Its effect on the hole variable is ignored because it stands for a closed term and therefore is not affected by substitutions.

4.3 Computations in ALF, Typelab, OLEG and $\lambda\Pi_C$: Comparison

In this section we want to compare directly the ways computations are done in these systems. In all of them the classical definition of $\beta$-reduction

$$(\lambda x : A. M) N \rightarrow_{\beta} M[x/N]$$

where $M[x/N]$ is the term $M$ in which the free occurrences of $x$ are replaced by $N$ is no longer atomic, but is divided into several phases.
Then the second phase formalizes the intermediate steps of distributing the value of $x$ down the term structure (except for Typelab). ALF and $\lambda\Pi_C$ try to mimic the propagation of the (meta)substitution by providing rules for computing with explicit substitutions. Unfortunately this very much complicates these calculi and leads to a loss of termination. On the other hand, OLEG takes an intermediate position between Typelab and ALF (All" converting $iI$-redexes to local definitions. Then using the $\delta$- and $\imath$-contraction schemes it propagates the value of $x$ in $M$.

So the original $\beta$ rule is simulated in the following ways:

<table>
<thead>
<tr>
<th>System</th>
<th>$\beta$-reduction expressed by means of</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALF</td>
<td>Explicit substitutions + substitution rules</td>
</tr>
<tr>
<td>Typelab</td>
<td>(meta)substitution + expl. subst at metavariables</td>
</tr>
<tr>
<td>OLEG</td>
<td>Local definition and a sequence of $\delta$-reductions</td>
</tr>
<tr>
<td>$\lambda\Pi_C$</td>
<td>Explicit substitutions + substitution rules</td>
</tr>
</tbody>
</table>

These approaches are very different and have their own specifics. For example, full-power explicit substitutions seem to cause more problems than the approach adopted in Typelab, because they allow too much freedom in handling the intermediate steps in substitution reductions. Apart from the termination problem, this causes also consistency problems because of the danger of loss of the Subject Reduction property. Keeping the metasubstitution allows us to discard the intermediate steps which are not essential for the purposes of interactive theorem proving at the expense of worsened implementation efficiency.

### 4.4 Explicit Substitutions

The idea of delaying substitutions can be found in many systems, even ones which do not call themselves explicit substitutions calculi. Explicit substitutions have different forms and names - take for example the redex created by a substitution when it passes over an $\omega$ in Automath or the explicit substitutions in $\lambda\omega$, Typelab, etc. Even the definitions in OLEG can be viewed as a tool to delay substitutions (cf. $(\beta)$-rule).

In each different case of introduction of explicit substitutions we have different goals and different tools at our disposal. This has led to the creation of a large variety of calculi of explicit substitutions. This paper does not aim at presenting an overview of those systems, one could consult for example [16]. Instead, we are interested in explicit substitutions only as far as they are related to the problems of open terms.

Nevertheless, there are many choices to be made when using explicit substitutions. Allowing too much liberty leads to the loss of properties as confluence, strong normalization or preservation of strong normalization. It seems that
these problems occur more frequently in systems of open terms and explicit 
substitutions which use de Bruijn indices.

Although using de Bruijn notation leads to some technical difficulties, for 
systems with full-power explicit substitutions, the fundamental problem of com-
putation in open term is translated into the fundamental problem of all explicit 
substitution calculus – namely the interaction between substitutions.

Typelab however shows us that maybe this is not an intrinsic feature of open 
terms since it uses no internal substitution reductions and is still able to perform 
better in terms of properties of the calculus than the other systems. We may 
conclude that most of the problems in systems of open terms with computations 
on explicit substitutions are due to the complexity of the underlying explicit 
substitution calculus without open terms.

4.5 Typechecking, Constraints and Unification

We conclude our discussion with a look at the connection between open terms 
and unification. Let us introduce the problem by the following example:

Suppose

\[ B : \text{Type} \]
\[ ?A : \text{Type} \]
\[ ?X : ?A \rightarrow ?A \]

and consider the instantiation \( ?X = \lambda x : B.x \). The question is do we accept 
this to be a valid instantiation or not.

ALF and ALLE give a positive answer to this question while for OLEG and 
Typelab this is not the case. In general, in order to check the validity of an 
instantiation, we will have to solve an equation \( M = N \) where both \( M \) and \( N \) 
may contain metavariables. In the case of dependently-typed calculi we clearly 
will be confronted with a higher-order unification problem which is known to be 
undecidable in general. But ALF and ALLE have means to delay the checking 
of these kind of equations by using them as constraints on the future instanti-
ations of the metavariables involved. This leads to a natural incorporation 
of unification algorithms solving decidable equations into the typechecker.

How does this compare to the situation in OLEG and Typelab – there uni-
fication is not explicitly incorporated into the typechecker, but rather applied 
on typechecked proof problems.

By careful inspection one may conclude that given the same unificational 
power the two approaches are not fundamentally different and the choice of one 
or the other depends on external factors.

4.6 Further Work

As we have already mentioned earlier, the focus of all considered systems is the 
unknown \emph{term} represented by a metavariable. One possible abstraction of this
principle one level up - namely shifting the attention from terms to *judgements* as abstract objects.

It was already pointed out that a metavariable is always considered together with its type and context. This of course means that we are actually given a judgement

\[ \Gamma_M \vdash ?M : A_M \]

which can be viewed either as goal or assumption in an interactive derivation session.

On the other hand the derivation rules can be viewed as relations or, with some assumptions, as functions on the set of judgements. In this way of course we obtain the calculus describing derivability of judgements.

If we view the judgements as basic atomic objects, we may then try to describe what an abstract instantiation is and investigate the problem of axiomatizing it. This axiomatization should of course be adequate with respect to the existing systems of open terms that may act as model for such a theory.

The most interesting point, which may turn out to be also the biggest problem with such a calculus is that it totally hides the computational aspect of the underlying system for open terms. Whether this is an advantage that allows generalization and useful abstraction from the underlying machinery or terminal invalidity is an interesting question.

**Acknowledgements**

The author would like to thank my supervisors for their comments which led to many improvements in this report.
# Appendix A

## Classification Summary

### Technical Issues

<table>
<thead>
<tr>
<th>Uses MV</th>
<th>ALF</th>
<th>Typelab</th>
<th>OLEG</th>
<th>$\lambda\Pi_C$</th>
<th>$\lambda_C$</th>
<th>Automath</th>
<th>$\lambda_\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit Subst.</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>functional encoding</td>
<td>create</td>
<td>yes</td>
</tr>
<tr>
<td>Based on</td>
<td>Martin-Löf</td>
<td>ECC</td>
<td>ECC</td>
<td>$\lambda$</td>
<td>untyped/typed</td>
<td>untyped/typed</td>
<td>untyped/typed</td>
</tr>
<tr>
<td>de Bruijn Ind.</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Properties

<table>
<thead>
<tr>
<th>Confluence</th>
<th>ALF</th>
<th>Typelab</th>
<th>OLEG</th>
<th>$\lambda\Pi_C$</th>
<th>$\lambda_C$</th>
<th>Automath</th>
<th>$\lambda_\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTA</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Subject red.</td>
<td>?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Type checking</td>
<td>decidable</td>
<td>decidable</td>
<td>decidable</td>
<td>decidable</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Type inference</td>
<td>not decidable</td>
<td>decidable</td>
<td>decidable</td>
<td>decidable</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Weak Norm.</td>
<td>MTA</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
<td>?</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Norm.</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>?</td>
<td>?</td>
<td>PSN fails</td>
</tr>
</tbody>
</table>

35
Contents

1.1 Introduction ................................................. 1
1.2 Issues of Interest and Classification Criteria .............. 2
1.3 Informal Use of the Metavariables in Interactive Theorem Proving 3
  1.3.1 Metavariables standing for (proof)terms .................. 3
  1.3.2 Metavariables as witnesses ............................. 4
  1.3.3 Computing with metavariables ........................... 4
1.4 Main Problem to Solve When Introducing Metavariables ...... 5

2 Open Term Systems for Theorem Proving ........................ 7
  2.1 ALF ......................................................... 7
    2.1.1 Syntax of the Calculus ................................ 7
    2.1.2 Adding incomplete (open) terms ....................... 8
    2.1.3 The Transitivity Example ........................... 11
    2.1.4 Classification Summary: ALF ....................... 11
  2.2 Typelab .................................................... 12
    2.2.1 Description of the Calculus ......................... 12
    2.2.2 The Transitivity Example ............................ 15
    2.2.3 Classification Summary: Typelab .................... 15
  2.3 OLEG ......................................................... 15
    2.3.1 The syntax ........................................... 16
    2.3.2 The Transitivity Example ............................ 18
    2.3.3 Classification Summary: OLEG ....................... 18
  2.4 The $\lambda\Pi_2$-calculus .................................. 19
    2.4.1 Description of $\lambda\Pi_2$ ............................. 19
    2.4.2 The Transitivity Example ............................ 20
    2.4.3 Classification Summary: $\lambda\Pi_2$ .................. 21

3 Other Calculi of Open Terms .................................... 22
  3.1 The Context Calculus $\lambda c$ ............................ 22
    3.1.1 Contexts in untyped $\lambda$-calculus as open terms .... 22
    3.1.2 Description of the Calculus ......................... 23
    3.1.3 Classification Summary: $\lambda c$ ................... 25
  3.2 Automath ..................................................... 25
    3.2.1 The calculus of segments ............................. 25
    3.2.2 Properties and Extensions of the System ............ 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 Classification Summary: Automath</td>
<td>27</td>
</tr>
<tr>
<td>3.3 The $\lambda e$-calculus</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1 Classification Summary: $\lambda e$</td>
<td>29</td>
</tr>
<tr>
<td>4 Discussion</td>
<td>30</td>
</tr>
<tr>
<td>4.1 Common features</td>
<td>30</td>
</tr>
<tr>
<td>4.2 Open Terms and Computation</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Computations in ALF, Typelab, OLEG and $\lambda \Pi e$: Comparison</td>
<td>31</td>
</tr>
<tr>
<td>4.4 Explicit Substitutions</td>
<td>32</td>
</tr>
<tr>
<td>4.5 Typechecking, Constraints and Unification</td>
<td>33</td>
</tr>
<tr>
<td>4.6 Further Work</td>
<td>33</td>
</tr>
<tr>
<td>A Classification Summary</td>
<td>35</td>
</tr>
</tbody>
</table>
Bibliography


If you want to receive reports, send an email to: m.m.j.l.philips@tue.nl (we cannot guarantee the availability of the requested reports)

**In this series appeared:**

97/02 J. Hoonan and O. v. Roosmalen  
97/03 J. Blanco and A. v. Deursen  
Basic Conditional Process Algebra, p. 20.
97/04 J.C.M. Baeten and J.A. Bergstra  
97/05 J.C.M. Baeten and J.J. Vereijken  
97/06 M. Franssen  
97/07 J.C.M. Baeten and J.A. Bergstra  
Bounded Stacks, Bags and Queues, p. 15.
97/08 P. Hoogendijk and R.C. Backhouse  
When do datatypes commute? p. 35.
Communication Modeling: The Language/Action Perspective, p. 147.
97/10 P.C.N. v. Gorp, E.J. Luit, D.K. Hammer  
E.H.L. Aarts  
Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11 A. Engels, S. Mauw and M.A. Reniers  
97/12 D. Haasnoot, E. Verheek and W. van der Aalst  
97/13 W.M.P. van der Aalst  
Exploring the Process Dimension of Workflow Management, p. 56.
97/14 J.F. Groote, F. Monin and J. Springintvelde  
A computer checked algebraic verification of a distributed summation algorithm, p. 28.
97/15 M. Franssen  
\(\lambda_t\); A Pure Type System for First Order Logic with Automated Theorem Proving, p. 35.
97/16 W.M.P. van der Aalst  
On the verification of Inter-organizational workflows, p. 23.
97/17 M. Vaceari and R.C. Backhouse  
Calculating a Round-Robin Scheduler, p. 23.
97/18 Werkgeheenschap Informatiewetenschap redactie: P.M.E. De Bra  
Informatica 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisceiplinaire Conferentie Informatiewetenschap, p. 60.
98/01 W. Van der Aalst  
98/02 M. Voorhoeve  
State / Event Net Equivalence, p. 25.
98/03 J.C.M. Baeten and J.A. Bergstra  
Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04 R.C. Backhouse  
98/05 D. Danis  
Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06 G. v.d. Bergen, A. Kaldewaij  
V.J. Duijseten  
Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07 Proceedings of the workshop on Workflow Management:  
Net-based Concepts, Models, Techniques and Tools (WF'M98)  
June 22, 1998 Lisbon, Portugal  
98/08 Informal proceedings of the Workshop on User Interfaces for Theorem Provers.  

T. Basten and J. Hooman Process Algebra in PVS

J. Zwanenburg The Proof-assistant Yarrow, p. 15


V. Bos and J.J.T. Kleijn Structured Operational Semantics of $\chi$, p. 27

H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst Diagnosing Workflow Processes using Woflan, p. 44

R.C. Backhouse and P. Hoogendijk Final Algebras: From Categories to Allegories, p. 26

S. Andova Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81


T. Basten and W. v.d. Aalst Inheritance of Workflows: An Approach to tackling problems related to change, p. 66


T.A.C. Willemse The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed $\mu$ CRL, p. 44


J.C.M. Baeten and J.A. Bergstra A Framework for Component Based Software Architectures, p. 19

A. Engels and S. Mauw Why men (and octopuses) cannot juggle a four ball cascade, p. 10


J.C.M. Baeten and T. Basten Inheritance of Behavior, p.83

J.C.M. Baeten and T. Basten Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79


J.C.M. Baeten and J.A. Bergstra Mode Transfer in process Algebra, p. 14

J.C.M. Baeten Process Algebra with Explicit Termination, p. 17.

S. Mauw and M.A. Reniers A process algebra for interworkings, p. 63.

R. Bloo, J. Hooman and E. de Jong Semantical Aspects of an Architecture for Distributed Embedded Systems*, p. 47.
Algebraic Process Verification, p. 65.
The Parallel Composition of Uniform Processes with Data, p. 19.
Variable Binding Operators in Transition System Specifications, p. 27.
Grammars Compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases, p. 33.
A Formal Development of Distributed Summation, p. 35.
Termination in Timed Process Algebra, p. 36.
Impossible Futures and Determinism, p. 14.
An Interactive Viewer for Mathematical Content based on Type Theory, p. 24.
Characterizing λ-terms with equal reduction behavior, p. 12.
Belief Revision with Explicit Justifications: an Exploration in Type Theory, p. 30.
Parameters in Pure Type Systems, p. 41.
Specifying Internet applications with DiCons, p. 9.
A Rewriting Approach to Binary Decision Diagrams, p. 27.
Interpretations of Automata, p. 41.