Typed inference systems:

a reference document

by

A.J.J.M. Marcelis

July, 1990
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 or the editor.
Typed inference systems: a reference document

A.J.J.M. Marcelis

Department of Mathematics and Computing science
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven
The Netherlands
e-mail: wsinmar@eutus1.win.tue.nl

March 1990

Abstract

Typed inference systems are introduced as a particular kind of formal deduction system, based on types and expressions, and in which contexts play an important role. Typed inference systems are reminiscent of systems for typed λ-calculi and natural deduction. Typical statements that can be deduced concern the type of an expression (under a context), and propositions, i.e., expressions of type boolean (again, relative to a context).

Contents

1 Introduction 1

2 Typed inference systems 1
   2.1 Inference formulae ........................................... 2
   2.2 Inference rules ............................................... 5
   2.3 Typed inference systems / Properties of derivable formulae 12
   2.4 Syntactical extensions ...................................... 13

3 A structured representation of derivations 13

References 17
1 Introduction

In this note we present a particular kind of formal deduction system, called typed inference systems, consisting of a collection of inference formulae and a collection of inference rules, acting on these formulae.

Typed inference systems are reminiscent of systems of typed λ-calculi, in that the main entities playing a role in the formulae are types and expressions, and λ-abstraction is one of the main expression constructors. The choice of (type and expression) constructs has been motivated by considerations of convenience for our purposes, rather than purity. For a detailed treatment of systems of typed λ-calculi — and the relations between them — we refer to [Bar..]. Some aspects of type structure, pertaining to typed inference systems, can also be found in a paper by Reynolds [Rey85].

Typed inference systems also contain elements from natural deduction, in that the inference rules exhibit a natural deduction-like structure, with introduction and elimination rules occurring pairwise per constructor. For more details on natural deduction, see [DalBO].

Typical statements (or judgements, as we shall call them) that can be deduced within a typed inference system concern the type of an expression, and propositions, i.e., expressions of type boolean. Such judgements are always derived relative to an environment, called a context.

Typed inference systems, as defined in this note, will serve as a basis for the development of a logic for attribute grammars. Such a logic is a means to prove the correctness of an attribute grammar (with respect to a specification) and construct such a grammar (satisfying a specification) in a compositional fashion, similar to the way Hoare's logic is employed for proving and constructing procedural programs (see [Apt81] for a survey paper on Hoare's logic). The first steps towards a logic for attribute grammars are reported on in [Mar90]; the present note will be used for reference purposes in that document (and future ones).

Apart from a fairly detailed description of the formulae and rules of a typed inference system, which is the subject of Section 2, this note also contains a brief description of a particular way of denoting derivations within such a system (Section 3). This technique, which is adopted from [Ned87], is an aid in the structured development of such derivations.

2 Typed inference systems

We use the notion of an inference system in the usual sense, i.e., as a collection of inference formulae and a collection of inference rules for these formulae. An inference rule is a construct

$$
\varphi_1 \ldots \varphi_n \vdash \varphi
$$

where ϕ is an inference formula and ϕ1,..,ϕn is a sequence of inference formulae. Formulae ϕ1,..,ϕn are called the premises of the rule, ϕ is called the conclusion, and we say that ϕ can be inferred from ϕ1,..,ϕn by application of the rule.

An inference rule with an empty sequence of premises is called an axiom; it is simply denoted by ϕ instead of $\varnothing$.

A rule, as above, may be accompanied by a side condition, i.e., a condition in terms of the entities that play a role in the formulae ϕ1,..,ϕn,ϕ. Such a side condition should be viewed as an additional restriction, to be satisfied with each application of the rule.

The inference formulae derivable within the system are defined to be those formulae that can be inferred from the axioms by zero or more applications of inference rules.

Subsections 2.1 and 2.2 are concerned with the formulae and rules of a typed inference system, respectively. Subsection 2.3 defines typed inference systems in terms of the concepts defined earlier, introduces a notation for derivability within such a system and lists some properties of derivable
formulae. In Subsection 2.4, some syntactical extensions to typed inference systems are discussed, aimed at future applications.

2.1 Inference formulae

Before we can state the inference formulae of a typed inference system, we must first dwell on the entities that play a role in such formulae, namely types, expressions, judgements and contexts. This is the purpose of definitions 2.1 through 2.4.

Definition 2.1 (Types)

Let $C_t$, $V$, and $L$ be sets (of type constants, type variables and labels, respectively), with $C_t \cap V = \emptyset$. The set of types over $C_t$, $V$, and $L$, is the smallest set, call it $\mathcal{T}$, satisfying

\[
\mathcal{T} = C_t \mid V \mid \mathcal{T} \to \mathcal{T} \mid \text{prod}(\mathcal{T}, \ldots, \mathcal{T}) \mid \text{sum}(L : \mathcal{T}, \ldots, L : \mathcal{T}) \mid \nu V, \mathcal{T}
\]

The above definition should be read as: $\mathcal{T}$ is the smallest set containing $C_t$, containing $V$, containing all constructs $\mathcal{T}_1 \to \mathcal{T}_2$ for $\mathcal{T}_1, \mathcal{T}_2 \in \mathcal{T}$, and so on. The set of types exhibits the usual constructors for function-, product- and sum-types, as well as the constructor $\nu$ for recursive types.

We have only included constructs that suit our purposes. In particular, the components of a product type $\text{prod}(\mathcal{T}_1, \ldots, \mathcal{T}_n)$ are identified by their relative position, whereas components of a sum type $\text{sum}(l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n)$ are identified by explicit labeling. For a more extensive and complete treatment of types, we refer to [R.ey85].

Definition 2.2 (Expressions)

Let $C_t$, $V$, $C_e$, $V_e$ and $L$ be sets, where $C_t, C_e$ and $V, V_e$ are mutually disjoint. (Elements of these sets are referred to as type constants, type variables, expression constants, expression variables and labels, respectively.) The set of expressions over these components is the smallest set, call it $\mathcal{E}$, satisfying

\[
\mathcal{E} = C_e \mid V_e \mid \lambda V_e : \mathcal{T} \cdot \mathcal{E} \mid \mathcal{E} . \mathcal{E} \mid \mathcal{E}_1 . \ldots . \mathcal{E}_n \mid \pi V, \mathcal{E} \mid [L, \mathcal{E}] \mid \text{case } \mathcal{E} : \mathcal{T} \text{ of } [L, V_e] \text{ then } \mathcal{E}_1, \ldots, [L_n, V_e] \text{ then } \mathcal{E}_n \mid \text{let } V_e : \mathcal{T} = \mathcal{E} \text{ in } \mathcal{E}
\]

wherein $\mathcal{T}$ is the set defined in definition 2.1 and $\mathbb{N}$ denotes the set of natural numbers.

In definition 2.2, constructs are ordered in such a way as to parallel the type constructs of the foregoing definition. Particularly, the third and fourth clause express $\lambda$-abstraction and application (related with function type $\mathcal{T}_1 \to \mathcal{T}_2$), followed by tupling and projection (product type), and injection and case-selection (sum type). $\mu$ is a constructor for recursive expressions, and a construction $\text{let } v : \mathcal{T} = e_1 \text{ in } e_2$ expresses the binding of expression $e_1$ to variable $v$ locally within expression $e_2$.

Again, the decision to include certain constructs is motivated by considerations of convenience for our purposes, rather than purity. In particular, the let-construct is introduced only for ease of notation. Its relation to other constructs can be expressed by stating that $\text{let } v : \mathcal{T} = e_1 \text{ in } e_2$ is equal to $(\lambda v : \mathcal{T} . e_2) . e_1$. (This relation is expressed formally by inference rule 4.6, see definition 2.7.) In addition, a case-expression $\text{case } e : \mathcal{T} \text{ of } [l_1, v_1] \text{ then } e_1, \ldots, [l_n, v_n] \text{ then } e_n$ combines a selection mechanism and its application to expression $e$. The corresponding "pure" select-expression has not been included here; see [R.ey85].

Throughout, notation $\text{FEV}(e)$ will be used to denote the set of expression variables occurring free in expression $e$. Likewise, $\text{FTV}(f)$ denotes the set of type variables occurring free in an expression or type $f$ (recall that some expressions contain explicit type information). These notions are defined using recursion on the structure of types and expressions. The full definitions are omitted here; we shall merely comment on a few clauses:
\[
\begin{align*}
FTV(\nu \tau.\tau) & = FTV(\tau) \setminus \{t\} \\
FTV(\lambda \nu : \tau. e) & = FTV(\tau) \cup FTV(e) \\
FTV(e_1, e_2) & = FTV(e_1) \cup FTV(e_2) \\
FTV(\text{case } e : \tau \text{ of } [l_1, v_1, \ldots, [l_n, v_n] \text{ then } e_n) & = FTV(e) \cup FTV(\tau) \cup (\bigcup k : 1 \leq k \leq n : FTV(e_k)) \\
FTV(\text{let } v : \tau = e_1 \text{ in } e_2) & = FTV(\tau) \cup FTV(e_1) \cup FTV(e_2) \\
FEV(\mu \nu : \tau. e) & = FEV(e) \setminus \{v\} \\
FEV(\lambda \nu : \tau. e) & = FEV(e) \setminus \{v\} \\
FEV(e_1, e_2) & = FEV(e_1) \cup FEV(e_2) \\
FEV(\text{case } e : \tau \text{ of } [l_1, v_1, \ldots, [l_n, v_n] \text{ then } e_n) & = FEV(e) \cup (\bigcup k : 1 \leq k \leq n : FEV(e_k) \setminus \{v_k\}) \\
FEV(\text{let } v : \tau = e_1 \text{ in } e_2) & = FEV(e_1) \cup (FEV(e_2) \setminus \{v\})
\end{align*}
\]

From these clauses it can be observed that type variable \( t \) does not occur free (hence: is bound) in a recursive type \( \nu \tau.\tau \), and the same holds for expression variable \( v \) in a recursive expression \( \mu \nu : \tau. e \). Next, \( \lambda \)-, case- and let-expressions involve the binding of expression variables. In particular, in a case-expression as above the set \( FEV(e_k) \) — for \( 1 \leq k \leq n \) — contributes to the set of free expression variables, however with the exception of \( v_k \). Thus, indeed, an occurrence of \( v_k \) in \( e_k \) acts as a bound variable for the case-construct as a whole. Finally, observe that the sets of (type and expression) variables occurring free in expressions \( \text{let } v : \tau = e_1 \text{ in } e_2 \) and \( (\lambda \nu : \tau. e_2) \cdot e_1 \) are the same, which is in support of the asserted equivalence of these expressions.

Another important notion is that of substitution of an (expression or type) variable in an expression or type. If \( e_1 \) and \( e_2 \) are expressions and \( v \) is an expression variable, then \( e_1[v \leftarrow e_2] \) denotes \( e_1 \) with every free occurrence of \( v \) replaced by \( e_2 \). Likewise, for an expression or type \( f \), a type \( \sigma \) and type variable \( s \), \( f[s \leftarrow \sigma] \) denotes \( f \) with \( \sigma \) substituted for every free occurrence of \( s \). The operations \( e_1[v \leftarrow e_2] \) and \( f[s \leftarrow \sigma] \) are defined with recursion on the structure of \( e_1 \) and \( f \), respectively, and employ renaming of bound variables (in \( e_1 \) and \( f \)) to prevent name clashes. For more details, we refer to [H&S86].

Next we define the kind of statements (henceforth called judgements) we want to make concerning types and expressions. These come in five forms:

**Definition 2.3 (Judgements)**

Let \( C, V, C_e, V_e \) and \( L \) be as in definition 2.2. The set of judgements over these components, call it \( J \), is defined by

\[
J = T : \ast \mid T \equiv_1 T \mid E : T \mid E \equiv_e E \mid E
\]

wherein \( T \) and \( E \) are as in definitions 2.1 and 2.2, respectively.

A judgement of the form \( \ast : \ast \) is pronounced as "\( \ast \) is a type", a judgement \( \tau_1 =_1 \tau_2 \) as "type \( \tau_1 \) equals type \( \tau_2 \)", a judgement \( e : \tau \) as "expression \( e \) has type \( \tau \)", one of the form \( e_1 =_e e_2 \) as "expression \( e_1 \) equals expression \( e_2 \)", and a judgement \( e \) as "expression \( e \) (holds)".

As the pronunciation suggests, a judgement of the first kind enables us to state the well-definedness of a type. Also, the third kind of judgement serves to state the type of an expression, and as such is a means to express well-formedness of an expression. (For instance, it may turn out that \( 3 + \text{true} \) has no type, whereas \( 3 + 2 \) has type integer — which is in accordance with the common interpretation of 2, 3, + and \text{true}.) Next, the second and fourth kind enable us to express equality of types and expressions, respectively. We already mentioned the equality of expressions...
let \( v : \tau = e_1 \) in \( (\lambda v : \tau. e_2)\cdot e_1 \). Finally, the last kind of judgement allows us to express propositions, i.e., expressions of type boolean (as will be clear from what follows).

However, judgements cannot be stated for a type or expression \( f \) in isolation, but can only be made relative to judgements concerning the free variables occurring in \( f \). For instance, the type of an expression \( e \) depends on the types of the variables in \( \text{FENV}(e) \). This observation leads to the introduction of contexts, where a context is a (possibly empty) sequence of judgements concerning variables. Notice the analogy with the formulae of definition 2.3.

**Definition 2.4 (Contexts)**

Let \( C_t, V_t, C_e, V_e \), and \( L \) be as before. The set of contexts over these components is the smallest set, denoted \( \mathcal{C} \), satisfying

\[
\mathcal{C} = \varepsilon \mid \mathcal{C}, V_t : * \mid \mathcal{C}, V_t =_T \mathcal{C}, V_e : T \mid \mathcal{C}, V_e =_E \mathcal{C}, E
\]

wherein \( T \) and \( E \) are as in definitions 2.1 and 2.2, respectively.

The pronunciation of terms in a context is analogous to the pronunciation of judgements, e.g. for \( t \in V_t \), \( t : * \) is pronounced "variable \( t \) is a type", etc. Symbol \( \varepsilon \) denotes the empty context.

**Definition 2.5 (Inference formulae)**

Let \( C_t, V_t, C_e, V_e \) and \( L \) be as before. The set of inference formulae over these components, denoted \( \mathcal{F} \), is defined by

\[
\mathcal{F} = \mathcal{C} \triangleright \mathcal{J}
\]

wherein \( \mathcal{C} \) and \( \mathcal{J} \) are as in definitions 2.4 and 2.3, respectively. (Hence, an inference formula consists of a context and a judgement, separated by the symbol \( \triangleright \). A formula \( \mathcal{A} \triangleright \mathcal{J} \) is pronounced as "under context \( \mathcal{A} \), judgement \( \mathcal{J} \).")

The entities \( T, E, \mathcal{J}, \mathcal{C} \) and \( \mathcal{F} \) (defined above) are parameterised with components \( C_t, V_t, C_e, V_e \), \( V_e \) and \( L \). For the definition of inference rules, we still need to add another component, namely a function assigning a type to all expression constants. In addition, the specific type constant \( \text{bool} \) is required to be included in \( C_t \). It is useful to invent a name for the six-tuple of components thus obtained. This is the purpose of the following definition.

**Definition 2.6 (Typed boolean structures)**

A *typed boolean structure* \( B \) is a six-tuple \( B = (C_t, V_t, C_e, V_e, L, \text{TA}) \), with

1. \( C_t, V_t, C_e, V_e \) and \( L \) are sets, such that \( C_t, V_t, C_e \) and \( V_e \) are mutually disjoint (referred to as sets of *type constants*, *type variables*, *expression constants*, *expression variables* and *labels*, respectively).
2. \( \text{TA} \in C_e \rightarrow T \), where \( T \) is as in definition 2.1 (*type assignment function* for expression constants). This function satisfies \( \text{FTV}(\text{TA}(c)) = \emptyset \) for all \( c \in C_e \), i.e., \( \text{TA}(c) \) yields a *closed* type for all \( c \in C_e \).
3. \( C_t \) contains the type constant \( \text{bool} \).
4. \( C_e \) contains the logical constants and operators: \( \text{true}, \text{false}, \neg, \wedge, \vee, \Rightarrow, =, \forall, \exists, \ldots \)
5. \( \text{TA} \) assigns the usual types to the logical symbols of \( C_e \).

\( \Box \)
2.2 Inference rules

The set of inference rules can be subdivided into a collection of rules per kind of judgement, extended with some rules for manipulating contexts. Before enumerating the rules (in definition 2.7), we shall first discuss some typical rules per category and demonstrate the application of these rules by giving examples of derivable formulae. The rules are identified as in definition 2.7 and consecutive Roman letters serve to identify example formulae. Also, with each formula \(f\), in the right margin the main formulae and rules are listed from which \(f\) can be inferred.

In fact, the example formulae here will be concerned with the type of integer lists. To this end, let \(B = (C_t, V_t, C_o, V_o, L, TA)\) be a typed boolean structure such that \(C_t\) contains the constants \(\text{int}\) and \(\text{fl}\), where \(\text{int}\) is the type of the integers and \(\text{fl}\) is the one-element type: the only expression of type \(\text{fl}\) is the expression constant \(w\). In addition, assume that \(\text{list} \in V_t\) and \(\text{nil, cons} \in L\).

1. Rules for deriving formulae of kind \(A \vdash \tau : \ast\) (“under context \(A\), \(\tau\) is a type”)

This set contains one rule per type constructor. The rules are listed in definition 2.7; here we merely give some examples of formulae derivable within the system:

(a) \(t : \ast \vdash \text{prod}(\text{int}, t) : \ast\)

(b) \(t : \ast \vdash \text{sum}(	ext{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t)) : \ast\)

(c) \(\varepsilon \vdash \text{vt.sum}(	ext{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t)) : \ast\)

In general, a formula \(A \vdash \tau : \ast\) can be derived within the system if for each variable \(t\) in \(\text{FTV}(\tau)\) the clause \(t : \ast\) appears in context \(A\). Notice that line (c) above does not conflict with the latter statement: variable \(t\) does not occur free in \(\text{vt.sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t))\).

2. Rules for deriving formulae \(A \vdash \tau_1 \equiv \tau_2\) (“under \(A\), type \(\tau_1\) equals type \(\tau_2\)”)

This set comprises the rule

\[
\frac{A \vdash \tau \equiv \ast}{A, t : \ast, t \equiv \tau \vdash t \equiv \tau}, \quad t \not\in \text{FTV}(\tau)
\]

which allows a context \(A\) — under which \(\tau\) is a type — to be extended with the binding of \(\tau\) to a name \(t\), and, indeed, under the extended context statement “type \(t\) equals type \(\tau\)” is justified.

We also have an unfolding rule for recursive types, expressing the equality of a recursive type \(\nu t.\tau\) and its first unfolding:

\[
\frac{A \vdash \nu t.\tau \equiv \ast}{A \vdash \nu t.\tau =_t \tau[t \mapsto \nu t.\tau]}
\]

wherein \(\tau[t \mapsto \nu t.\tau]\) denotes \(\tau\) with \(\nu t.\tau\) substituted for every free occurrence of \(t\). This rule justifies the interpretation of \(\nu t.\tau\) as a solution of a recursive type equation, or, alternatively, as a fixed point of the function (from types to types) that maps \(t\) to \(\tau\).

This category also contains a rule for changing bound variable \(t\) in a recursive type \(\nu t.\tau\), a rule in support of the view that the order of components in a sum-type is irrelevant and a number of congruence rules, expressing that \(=_t\) is a congruence relation on types (see definition 2.7 for details).

For example, using the rules displayed above we can derive

\[(d) \: \text{list : \ast, list} =_t \nu t.\text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t)) \vdash \text{list} =_t \nu t.\text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t))\]

\[(e) \: \varepsilon \vdash \nu t.\text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t)) =_t \text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, \nu t.\text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t))))\]
(Note: to save writing in what follows, we shall use \( A_0 \) to denote the context of line (d), i.e., the context in which the name \( \text{list} \) is bound to the \( \nu \)-term.)

By applying congruence rules and rules for context extension to (d) and (e), the following is derivable also

\[ A_0 \vdash \text{list} =_\nu \sum(\text{nil} : \Omega, \cons : \prod(\text{int}, \text{list})) \]

Notice that line (e) expresses the same judgement in terms of \( \nu \cdot \sum(\text{nil} : \Omega, \cons : \prod(\text{int}, t)) \) as (f) does in terms of \( \text{list} \). This is made possible by the binding of \( \text{list} \) to the \( \nu \)-term in the context of the latter line. From (f) it follows that \( \text{list} \) (or, alternatively, the \( \nu \)-term it is bound to) solves the type equation \( t =_\nu \sum(\text{nil} : \Omega, \cons : \prod(\text{int}, t)) \). Therefore \( \text{list} \) can be regarded as the type of the lists of integers.

3. Rules for deriving formulae \( A \vdash e : \tau \) ("under \( A \), expression \( e \) has type \( \tau \))

This category contains a rule for each of the "expressions constructors" of definition 2.2, giving the type of an expression formed with that constructor (dependent on the context). For instance, the rule

\[
\begin{array}{c}
\frac{A \vdash e : \tau_k \\
A \vdash \sum(l_1 : \tau_1, \ldots, l_n : \tau_n) : \ast \\
\quad 1 \leq k \leq n}
{A \vdash [l_k, e] : \sum(l_1 : \tau_1, \ldots, l_n : \tau_n)}
\end{array}
\]

expresses that whenever \( e \) has type \( \tau_k \), and the latter appears as a component of a well-defined sum-type, then \( e \) "tagged" with the appropriate label \( l_k \), i.e., \([l_k, e]\), has that sum-type. From rule (3.7) it follows that the type of an expression (under a given context) need not be unique. However, the inference rules are such that — for any context \( A \) and expression \( e \) — if a type can be determined for \( e \) under \( A \), then \( e \) also has a smallest\(^1\) type under \( A \), and the latter type is unique, apart from equality in the sense of \( =_\tau \). (The inclusion of type information in expressions, notably in \( \lambda \)- and case-constructs, is an aid in achieving this property.) In fact, an additional rule in this category expresses that the type of an expression can at best be fixed up to equality of types:

\[
\begin{array}{c}
\frac{A \vdash e : \tau_1}{A \vdash e : \tau_2} \quad \text{if } \tau_1 =_\tau \tau_2
\end{array}
\]

In connection with our running example, it is for instance possible to derive

\begin{align*}
(g) & A_0 \vdash \omega : \Omega \\
(h) & A_0 \vdash [\text{nil}, \omega] : \sum(\text{nil} : \Omega, \cons : \prod(\text{int}, \text{list})) \\
(i) & A_0 \vdash [\text{nil}, \omega] : \text{list} \\
(j) & A_0 \vdash (3, [\text{nil}, \omega]) : \prod(\text{int}, \text{list}) \\
(k) & A_0 \vdash [\cons, (3, [\text{nil}, \omega])] : \sum(\text{nil} : \Omega, \cons : \prod(\text{int}, \text{list})) \\
(l) & A_0 \vdash [\cons, (3, [\text{nil}, \omega])] : \text{list}
\end{align*}

As exemplified by these formulae, the expressions that can be inferred to have type \( \text{list} \) are \([\text{nil}, \omega]\) and all expressions of the form \([\cons, (i, l)]\), wherein \( i \) has type \( \text{int} \) and \( l \) has type \( \text{list} \). Such expressions may well be considered to denote lists of integers, which, then, justifies the interpretation of \( \text{list} \) (or, alternatively, the \( \nu \)-term it is bound to in \( A_0 \)) as the type of lists of integers.

\(^1\)The notion of a smallest type for an expression (under a context) can be formalised via the introduction of a subtyping relation on types. We shall not do so here; see [Rey85].
4. Rules for deriving formulae $A 	riangleright e_1 =_e e_2$ ("under $A$, expression $e_1$ equals expression $e_2"$

This set contains conversion rules for (certain combinations of) expressions. For instance, we have the usual $\beta$-conversion rule (in connection with $\lambda$-abstraction and application) and an unfolding rule for recursive expressions. Let us here discuss the conversion rule for case-expressions:

\[
A \triangleright (\text{case } [l_k,e]:\text{of } [I_1,v_1]; \ldots; [I_n,v_n] \text{ then } e_n) : \tau
\]

The premiss of the rule expresses that the case-construct under consideration must be well-defined, in the sense that some type $\tau$, can be derived for it. By rule (3.8), this implies that type $\alpha$ of the expression following the word case is equal to $\text{sum}(\tau_1; \ldots; \tau_n)$ for some $\tau_1; \ldots; \tau_n$, i.e., the latter expression is indeed of the form $[l_k,e]$. With this proviso, the rule's conclusion expresses that the case-construct reduces to expression $e_k$ — corresponding to label $I_k$ of $[l_k,e]$ — with $e$ substituted for each free occurrence of $v_k$.

Using this rule, in connection with our running example we can derive for instance (with context $A_0$ as before)

\[
A_0 \triangleright \text{case } \text{cons}, \{3,[\text{nil, } \omega]\} : \text{list of } [\text{nil, } v_1] \text{ then } 99, [\text{cons, } v_2] \text{ then } \pi_1, v_2
\]

where, in fact, the number 3 is obtained as a result of $(\pi_1, v_2)[v_2 \xrightarrow{e} (3,[\text{nil, } \omega])]$. Thus, the (selection part of the) given case construction applied to expression $\text{cons}, \{3,[\text{nil, } \omega]\}$ of type list reduces to 3. In general, this case construction maps the empty list to the number 99, and maps a non-empty list to the first element of that list. This exemplifies that the case-construct is actually not a "pure" construct, but a combination of selection and application.

This category of rules also includes a number of congruence rules, expressing that $=_e$ is a congruence relation on expressions; see definition 2.7 for more details.

5. Rules for deriving formulae $A \triangleright e$ ("under context $A$, $e$ (holds)"

These serve to derive expressions of type bool. Apart from some special rules, they include the usual natural deduction rules; see definition 2.7 for examples.

Also, given a context $A$ under which a recursive type $vt.\tau$ is well-defined, it is possible to state an induction rule for the expressions of this type (provided that $\tau$’s shape satisfies certain conditions, which we shall not specify here). For instance, for the type list of our running example, if $A$ is a context such that $A \triangleright P : \text{list} \rightarrow \text{bool}$ is derivable, the appropriate rule for structural induction reads

\[
A \triangleright P[\text{nil, } \omega] \\
A, i : \text{int}, l : \text{list, } P.l \triangleright P[\text{cons}, (i, l)] \\
A \triangleright \forall l : \text{list } [P.l]
\]

A general strategy for finding suitable induction rules for recursively defined types will not be presented here.

6. Rules for manipulating contexts

The rules in this category state the conditions under which terms may be added to a context, or removed from it, while preserving the judgements derivable. They need no further explanation.
Definition 2.7 (Inference rules)

Everything in this definition is relative to a typed boolean structure $B = (C_t, V_t, C_e, V_e, L, TA)$ and sets $T, E, J, C$ and $F$ (of types, expressions, judgements, contexts and inference formulae, respectively), defined over $B$.

To denote arbitrary elements of the various sets involved, we use certain fixed identifiers (possibly subscripted), according to the following conventions:

\[
\begin{align*}
  ct & \in C_t & \tau, \sigma & \in T \\
  t, s & \in V_t & e, f & \in E \\
  ce & \in C_e & j & \in J \\
  v, w & \in V_e & A & \in C \\
  l & \in L
\end{align*}
\]

1. Rules for deriving formulae of kind $A \vdash \tau : *$ ("under context $A$, $\tau$ is a type")

1.1. $e \vdash ct : *$

1.2. $\frac{A \vdash true}{A, t:* \vdash t:*} \\
1.3. $\frac{A \vdash \tau_1 : * \quad A \vdash \tau_2 : *}{A \vdash \tau_1 \rightarrow \tau_2 : *} \\
1.4. $\frac{A \vdash \tau_1 : * \quad \ldots \quad A \vdash \tau_n : *}{A \vdash \text{prod}({\tau_1, \ldots, \tau_n}) : *} \\
1.5. $\frac{A \vdash \tau_1 : * \quad \ldots \quad A \vdash \tau_n : *}{A \vdash \text{sum}(l_1: \tau_1, \ldots, l_n: \tau_n) : *}$, $l_1, \ldots, l_n \in L$ (pairwise different)

1.6. $\frac{A, t:* \vdash \tau : *}{A \vdash \nu t. \tau : *}$

2. Rules for deriving formulae $A \vdash \tau_1 =_t \tau_2$ ("under $A$, type $\tau_1$ equals type $\tau_2"\)

2.1. $\frac{A \vdash \tau : *}{A, t:* \vdash t =_t \tau \vdash t =_t \tau}$, $t \notin FTV(\tau)$

2.2. $\frac{A \vdash \nu t. \tau : *}{A \vdash \nu t. \tau =_t \tau[t \mapsto \nu t. \tau]}$

The following rule provides the key to the statement that the order of components in a sum-type is irrelevant (as far as equality under $=_t$ is concerned).

2.3. $\frac{A \vdash \text{sum}(l_1: \tau_1, \ldots, l_n: \tau_n) : *}{A \vdash \text{sum}(l_1: \tau_1, \ldots, l_n: \tau_n) =_t \text{sum}(l_1: \tau_1, \ldots, l_k: \tau_k, l_{k+1}: \tau_{k+1}, \ldots, l_n: \tau_n)}$, $1 \leq k < n$

Rule (2.4) concerns the replacement of bound variable $t$ of a recursive type $\nu t. \tau$:

2.4. $\frac{A \vdash \nu t. \tau : *}{A \vdash \nu t. \tau =_t \nu s.(\tau[t \mapsto s])}$, $s \notin FTV(\tau)$
The remainder of the rules are congruence rules, expressing that \( =_t \) is a congruence relation on types. Apart from rules for reflexivity, symmetry and transitivity, there is one rule per type constructor

\[
(2.5) \quad \frac{A \vdash t : \tau}{A \vdash t =_t \tau}
\]

\[
(2.6) \quad \frac{A \vdash t_1 =_t t_2}{A \vdash t_2 =_t t_1}
\]

\[
(2.7) \quad \frac{A \vdash t_1 =_t t_2 \quad A \vdash t_2 =_t t_3}{A \vdash t_1 =_t t_3}
\]

\[
(2.8) \quad \frac{A \vdash t_1 =_t \sigma_1 \quad A \vdash t_2 =_t \sigma_2}{A \vdash (t_1 \rightarrow t_2) =_t (\sigma_1 \rightarrow \sigma_2)}
\]

\[
(2.9) \quad \frac{A \vdash t_1 =_t \sigma_1 \quad \ldots \quad A \vdash t_n =_t \sigma_n}{A \vdash \text{prod}(t_1, \ldots, t_n) =_t \text{prod}(\sigma_1, \ldots, \sigma_n)}
\]

\[
(2.10) \quad \frac{A \vdash \text{sum}(t_1 : t_1, \ldots, t_n : t_n) =_t \text{sum}(\sigma_1 : \sigma_1, \ldots, \sigma_n : \sigma_n)}{A \vdash t =_t \sigma} \quad (l_1, \ldots, l_n \in \mathcal{L} \text{ pairwise different})
\]

\[
(2.11) \quad \frac{A, t \vdash \tau =_t \sigma}{A \vdash \nu t. \tau =_t \nu t. \sigma}
\]

9. Rules for deriving formulae \( A \vdash e : \tau \) ("under \( A \), expression \( e \) has type \( \tau \))

\[
(3.1) \quad e \vdash ce : TA(ee)
\]

\[
(3.2) \quad \frac{A \vdash \tau : *}{A, v : \tau \vdash v : \tau}
\]

\[
(3.3) \quad \frac{A \vdash t_1 : * \quad A, v : t_1 \vdash e : t_2}{A \vdash (\lambda v : t_1. e) : t_1 \rightarrow t_2}
\]

\[
(3.4) \quad \frac{A \vdash e_1 : t_1 \rightarrow t_2 \quad A \vdash e_2 : t_1}{A \vdash e_1 \cdot e_2 : t_2}
\]

\[
(3.5) \quad \frac{A \vdash e_1 : t_1 \quad \ldots \quad A \vdash e_n : t_n}{A \vdash (e_1, \ldots, e_n) : \text{prod}(t_1, \ldots, t_n)}
\]

\[
(3.6) \quad \frac{A \vdash e : \text{prod}(t_1, \ldots, t_n)}{A \vdash \pi_k e : t_k} \quad , 1 \leq k \leq n
\]

\[
(3.7) \quad \frac{A \vdash e : t_k \quad A \vdash \text{sum}(t_1 : t_1, \ldots, t_n : t_n) : *}{A \vdash [l_k, e] : \text{sum}(t_1 : t_1, \ldots, t_n : t_n) \quad , 1 \leq k \leq n}
\]

\[
A \vdash e : \sigma \quad A \vdash \sigma =_t \text{sum}(t_1 : t_1, \ldots, t_n : t_n)
\]

\[
A, v_1 : t_1 \vdash e_1 : \tau \quad \ldots \quad A, v_n : t_n \vdash e_n : \tau
\]

\[
A \vdash (\text{case } e : \sigma \text{ of } [l_1, v_1] \text{ then } e_1, \ldots, [l_n, v_n] \text{ then } e_n) : \tau
\]

\[
A, v : \tau \vdash e : \tau
\]

\[
A \vdash (\mu v : \tau. e) : \tau
\]
4. Rules for deriving formulae $A ⊢ e₁ =_e e₂$ ("under $A$, expression $e₁$ equals expression $e₂" )$

\[
\frac{A ⊢ e \colon τ}{A, v : τ ⊢ e =_e e} \quad , v \not\in \text{FEV}(e)
\]

\[
\frac{A ⊢ \lambda v : τ₁. e₂ : τ₁ \rightarrow τ₂ \quad A ⊢ e₁ : τ₁}{A ⊢ (\lambda v : τ₁. e₂) \cdot e₁ =_e e₂[v \mapsto e₁]}
\]

\[
\frac{A ⊢ \langle e₁, \ldots , eₙ ⟩ : \text{prod}(τ₁, \ldots , τₙ) \quad 1 \leq k \leq n}{A ⊢ σ_k(e₁, \ldots , eₙ) =_e e_k}
\]

\[
\frac{A ⊢ (\text{case } [I, e] : \sigma \text{ of } [I, u] \text{ then } e₁, \ldots , [Iₙ, uₙ] \text{ then } eₙ) : τ \quad 1 \leq k \leq n}{A ⊢ e[I, e] =_e e[Iₙ, eₙ][e_k \mapsto e_I]}
\]

\[
\frac{A ⊢ (μ v : τ. e) : τ}{A ⊢ μ v : τ. e =_e e[v \mapsto μ v : τ. e]}
\]

\[
\frac{A ⊢ (\text{let } v : τ = e₁ \text{ in } e₂) : τ_2}{A ⊢ (\text{let } v : τ₁ = e₁ \text{ in } e₂) =_e (\lambda v : τ₁. e₂) \cdot e₁}
\]

Relation $=_t$ on types induces another rule for each of the expressions containing explicit type information. For instance, in connection with $\lambda$-abstraction, the appropriate rule reads

\[
\frac{A ⊢ τ₁ =_t τ₂ \quad A ⊢ (λ v : τ₁. e) : τ}{A ⊢ (λ v : τ₁. e) =_e (λ v : τ₂. e)}
\]

Also, this category contains rules for changing bound variables of $\lambda$-, case-, let- and recursive expressions. For example, we have

\[
\frac{A ⊢ (λ v : τ₁. e) : τ}{A ⊢ (λ v : τ₁. e) =_e (λ w : τ₁. e)[v \mapsto w]} \quad , w \not\in \text{FEV}(e)
\]

Finally, this category contains a number of congruence rules, expressing that $=_e$ is a congruence relation on expressions. Apart from rules for reflexivity, symmetry, and transitivity, there is a rule for each of the "expression constructors" of definition 2.2. We have, for example

\[
\frac{A, v : τ \vdash e₁ =_e e₂}{A ⊢ (λ v : τ. e₁) =_e (λ v : τ. e₂)}
\]

\[
\frac{A ⊢ e₁ =_e f₁ \quad \ldots \quad A ⊢ eₙ =_e fₙ}{A ⊢ \langle e₁, \ldots , eₙ ⟩ =_e \langle f₁, \ldots , fₙ ⟩}
\]

We shall not list the rules exhaustively here; they are reminiscent of the rules expressing congruence of $=_t$ for types, cf. point 2. above.
5. Rules for deriving formulae $A \vdash e$ ("under $A$, e (holds")

(5.1) $e \vdash \text{true}$

(5.2) \[
\frac{A \vdash e : \text{bool}}{A, e \vdash e}
\]

(5.3) \[
\frac{A, v : \tau, v = e_1 \vdash e_2}{A \vdash \text{let } v : \tau = e_1 \text{ in } e_2}
\]

(5.4) \[
\frac{A \vdash e_1 \quad A \vdash e_1 = e_2}{A \vdash e_2}
\]

This category also contains the usual natural deduction rules [Dal80]. For the logical constructors $\land$, $\lor$, $\Rightarrow$, $\neg$, $\exists$ the corresponding introduction- and elimination rules are referred to by (in $\land$), (el $\land$), (in $\lor$), (el $\lor$), etc. As an example, let us give the introduction- and elimination rules for $\Rightarrow$ and $\forall$:

(in $\Rightarrow$) \[
\frac{A, e_1 \vdash e_2}{A \vdash e_1 \Rightarrow e_2}
\]

(el $\Rightarrow$) \[
\frac{A \vdash e_1 \quad A \vdash e_1 \Rightarrow e_2}{A \vdash e_2}
\]

(in $\forall$) \[
\frac{A \vdash \forall \nu : \tau. e}{A \vdash \forall \nu : \tau. e}
\]

(el $\forall$) \[
\frac{A \vdash \forall \nu : \tau. e_1 \quad A \vdash e_2 : \tau}{A \vdash e_2 \left[ \nu \mapsto e_2 \right]}
\]

Also, for a recursive type $\nu t. \tau$ (which is well-defined under a certain context $A$, i.e., $A \vdash \nu t. \tau : *$ is derivable) it is possible to include an inference rule stating an induction principle for the expressions of this type. It would carry too far to treat induction rules for recursive types in general here. However, an example — concerning integer lists — can be found earlier in this subsection.

6. Rules for manipulating contexts

(6.1) \[
\frac{A \vdash j}{A, t : * \vdash j}, \; t \notin FTV(j)
\]

(6.2) \[
\frac{A, t : * \vdash j \quad A \vdash \tau : *}{A, t : *, t = \tau \vdash j}, \; \tau \notin FTV(t)
\]

(6.3) \[
\frac{A \vdash j \quad A \vdash \tau : *}{A, v : \tau \vdash j}, \; v \notin FEV(j)
\]

(6.4) \[
\frac{A, v : \tau \vdash j \quad A \vdash e : \tau}{A, v : \tau, v = e \vdash j}, \; v \notin FEV(e)
\]

(6.5) \[
\frac{A \vdash j \quad A \vdash e : \text{bool}}{A, e \vdash j}
\]

(6.6) \[
\frac{A, t : * \vdash j}{A \vdash j}, \; t \notin FTV(j)
\]
Properties of derivable formulae

We define typed inference systems in terms of the notions of subsections 2.1 and 2.2, and introduce a notation for the derivability of formulae within such a system. Next, we list some properties of derivable formulae.

Definition 2.8 (Typed inference system; derivability)

Let $B = (C, V, C_v, V_v, L, TA)$ be a typed boolean structure, cf. definition 2.6. $TIS_B$, the typed inference system over $B$, is the pair $(F, R)$, wherein $F$ is the set of inference formulae over $B$ according to definition 2.5, and $R$ is the set of inference rules over $B$ according to definition 2.7. For $\varphi \in F$, notation

$$B \vdash TIS \varphi$$

stands for "formula $\varphi$ is derivable from the axioms with the rules from $R$".

Without proof, we list the following properties of the formulae derivable within a typed inference system.

1. If $A \vdash j$ is derivable, then for each variable $t \in FTV(j)$ the clause $t: *$ occurs in $A$, and for each $v \in FEV(j)$ there is a type $\tau$ such that $A \vdash \tau : *$ is derivable and $v: \tau$ occurs in $A$.
2. If $A \vdash \tau_1 \equiv \tau_2$ is derivable, then $A \vdash \tau_1 : *$ and $A \vdash \tau_2 : *$ are derivable also.
3. If $A \vdash e : \tau$ is derivable, then so is $A \vdash \tau : *$.
4. If $A \vdash e_1 \equiv e_2$ is derivable, then there exists a type $\tau$ under $A$ (i.e., $A \vdash \tau : *$ is derivable) such that $A \vdash e_1 : \tau$ and $A \vdash e_2 : \tau$ are derivable within the system.
5. If $A \vdash e$ is derivable, then so is $A \vdash e: bool$.

Statements similar to those above hold for (the clauses constituting) the context of a derivable formula. Hereafter, "context" is short for "context of a derivable formula".

6. If $A_1, t =_t \tau, A_2$ is a context, then there exists a context $A_0$ such that $A_1$ equals $A_0, t : *$ and $A_0 \vdash \tau : *$ is a derivable formula. (Hence, by 1., for each $s \in FTV(\tau)$ the clause $s: *$ appears in $A_0$.)
7. If $A_1, v : \tau, A_2$ is a context, then $A_1 \vdash \tau : *$ is a derivable formula.
8. If $A_1, v =_e e, A_2$ is a context, then there exists a context $A_0$ and a type $\tau$ such that $A_1$ equals $A_0, v : \tau$ and $A_0 \vdash e : \tau$ is a derivable formula.
9. If $A_1, e, A_2$ is a context, then $A_1 \vdash e : bool$ is derivable.

Properties 6. through 9., together with 1., also imply the following:
10. If \(A_1, j, A_2\) is a context with \(j\) of one of the forms \(t = t, v = e\) or \(e\), then for each variable \(s \in FTV(j)\) the clause \(s : *\) occurs in \(A_1\), and for each \(w \in FEV(j)\) there is a type \(\sigma\) such that \(A_1 \triangleright \sigma : *\) is derivable and \(w : \sigma\) occurs in \(A_1\). If \(j\) is of the form \(v : T\), then for each \(s \in FTV(T)\) the clause \(s : *\) occurs in \(A_j\).

### 2.4 Syntactical extensions

In future applications of typed inference systems, we shall sometimes employ some straightforward extensions to the syntax introduced up till now, aimed at improving the readability of formulae. In particular, we mention

- Expressions may be denoted using a more free style than prescribed above. In particular, infix notation for function symbols will be allowed, and brackets will be inserted to avoid ambiguities. Also, \(\lambda\)-abstraction over multiple variables will be written \(\lambda v_1 : t_1, \ldots, v_n : t_n : e\) rather than \(\lambda v_1 : t_1, (\ldots (\lambda v_n : t_n : e) \ldots )\), and local binding of multiple variables will be denoted \(\text{let } v_1 : t_1 = e_1, \ldots, v_n : t_n = e_n \text{ in } e\) instead of \(\text{let } v_1 : t_1 = e_1 \text{ in } (\ldots (\text{let } v_n : t_n = e_n \text{ in } e) \ldots )\).

- In writing down a context, the order of its constituent terms may be relaxed somewhat. In particular, the strict succession of declarations of variables and their definitions — viz. \(t = t, t = t\) and \(v : \tau, v = e\), see properties 6. and 8. in subsection 2.3 — will be abandoned, with the proviso that the declaration of a variable still occurs to the left of its definition. Formally, this generalisation requires the introduction of additional inference rules, stating the conditions under which two adjacent context terms \(h_1, h_2\) may be exchanged (roughly, these conditions amount to the statement that the free variables in \(h_1\) and \(h_2\) do not interfere).

- The definition of recursively defined types and expressions will be generalised to \(n\)-tuples. Concerning recursive types, \(t = t \nu v. T\) generalises to \(t_1, \ldots, t_n = t \nu v_1, \ldots, v_n : T_1, \ldots, T_n\) (for simplicity, bound variables in the \(\nu\)-term are chosen equal to the variables being defined). This enables the definition of mutually recursive types, if we allow variables \(v_1, \ldots, v_n\) to occur in each of the types \(T_1, \ldots, T_n\). A term \(t_1, \ldots, t_n = t \nu v_1, \ldots, v_n : T_1, \ldots, T_n\) will be written in a more readable form as

\[
\text{rec}(t_1 = t_1, \ldots, t_n = t_n).
\]

For example, the definition of \textit{list} from subsection 2.2 can now be denoted as

\[
\text{rec}((\text{list} = t \nu v_1, \text{cons} : \text{prod}(\text{int}, \text{list}))).
\]

Analogously, we now write

\[
\text{rec}(v_1 : t_1 = e_1, \ldots, v_n : t_n = e_n)
\]

for \(v_1, \ldots, v_n = e_1, \ldots, v_n = e_n\) which, in its turn, generalises from

\[v = e \mu \nu : \tau. e\].

### 3 A structured representation of derivations

The derivation of an inference formula \(\varphi\) within a typed inference system consists of a sequence of formulae \(A_1 \triangleright j_1, \ldots, A_n \triangleright j_n\), where \(\varphi\) equals \(A_n \triangleright j_n\). In such a derivation, contexts \(A_k\) — for \(1 \leq k \leq n\) — tend to be rather constant: compared with the contexts of preceding formulae, context \(A_k\) may at best be extended with — or reduced by — a few terms at its right hand side. Therefore, to save writing and improve readability, we adopt the convention to collect context terms in boxes ("flag") and to display their lifetime by means of vertical lines in the left margin ("flag-staffs"). Also, in the right margin of each line a hint is given, saying how this line can be inferred from preceding ones by applying an inference rule. With a slight modification, this method has been adopted from [Ned87].
For an arbitrary line in the derivation, where judgement \( j \) is displayed, the corresponding flat formula is \( A \vdash j \), where \( A \) is obtained by "flattening" all context terms currently in force.

**Example 3.1**
In connection with the example of integer lists of subsection 2.2, we present the derivation of

\[
\epsilon \vdash \nu t.\text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, t)) : *
\]

(which is the formula denoted by (c) on page 5), both in a traditional notation and in flag-notation.

In either case, lines are numbered consecutively and a comment in the right margin of a line indicates how the formula in that line is inferred from those in preceding lines by application of an inference rule. For example, the comment (1.4),4,5 denotes inference from lines 4 and 5 by application of rule (1.4).

**Traditional:**

1. \( \epsilon \vdash \Omega : * \)  
2. \( \epsilon \vdash \text{int} : * \)  
3. \( t : * \vdash \Omega : * \)  
4. \( t : * \vdash \text{int} : * \)  
5. \( t : * \vdash t : * \)  
6. \( t : * \vdash \text{prod}(\text{int}, t) : * \)  
7. \( t : * \vdash \text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, t)) : * \)  
8. \( \epsilon \vdash \nu t.\text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, t)) : * \)

**Flag-notation:**

1. \( \Omega : * \)  
2. \( \text{int} : * \)  
3. \( t : * \)  
4. \( \Omega : * \)  
5. \( \text{int} : * \)  
6. \( t : * \)  
7. \( \text{prod}(\text{int}, t) : * \)  
8. \( \text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, t)) : * \)  
9. \( \nu t.\text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, t)) : * \)

Notice that the empty context \( \epsilon \) is not displayed explicitly in the latter derivation (as opposed to the former one), and formulae in the traditional derivation can be obtained by flattening formulae in the derivation with flag-notation.

The way to read — and construct — the derivation of a formula in flag-notation is "outside-in", in the following sense. Suppose formula \( A_1, A_2 \vdash j \) has to be derived. Initially, this is written as
1. \[ A_1, A_2 \]
\[ \vdots \]
2. \[ j \]

where "..." denotes the part of the derivation still to be constructed. Now suppose that, typically, \( A_1, A_2 \vdash j \) occurs as the conclusion of (an instantiation of) some inference rule \( r \) as follows

\[
\frac{A_1 \vdash j_1 \quad A_1, A_2 \vdash j_2 \quad A_1, A_2, A_3 \vdash j_3}{A_1, A_2 \vdash j}
\]

then (*) is refined as

1. \[ A_1 \]
\[ \vdots \]
2. \[ j_1 \]
3. \[ A_2 \]
\[ \vdots \]
4. \[ j_2 \]
5. \[ A_3 \]
\[ \vdots \]
6. \[ j_3 \]
7. \[ j \]

which leaves three (smaller) parts still to be filled up. This way, the construction of a derivation amounts to repeatedly closing (continuously diminishing) gaps. Observe how the scheme (**) clearly indicates that context-parts \( A_1, A_2 \) and \( A_3 \) all pertain to judgement \( j_3 \), whereas \( j_2 \) must be arrived at in context \( A_1, A_2 \). Also, \( j_1 \) must be derived in context \( A_1 \) only, and, in fact, the wish to express this caused \( A_1 \) and \( A_2 \) to be spread over two flags.

Let us illustrate the "outside-in" construction of derivations by way of another example.

**Example 3.2**
Consider again the type of the integer lists of subsection 2.2. Like before, we use \( A_0 \) to denote the context-part binding the name \( \text{list} \) to \( \nu.t.\text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t)) \). More precisely, \( A_0 \) is

\[
\text{list} : \star, \text{list} = \nu.t.\text{sum}(\text{nil}:\Omega, \text{cons} : \text{prod}(\text{int}, t))
\]

Our aim is to derive

\[
A_0 \vdash (\text{case } [\text{cons}, (3, [\text{nil}, \omega])] : \text{list of } [\text{nil}, v_1] \text{ then } 99, [\text{cons}, v_2] \text{ then } \pi_1.v_2) : \text{int}
\]

that is, under \( A_0 \), the case-construct — which was also used in subsection 2.2 — has type \( \text{int} \).

Thus, we write

1. \[ A_0 \]
\[ \vdots \]
2. \[ (\text{case } [\text{cons}, (3, [\text{nil}, \omega])] : \text{list of } [\text{nil}, v_1] \text{ then } 99, [\text{cons}, v_2] \text{ then } \pi_1.v_2) : \text{int} \]
The conclusion of inference rule (3.8) is concerned with the type of a case-construct. Applying this rule — with the sum-type occurring in it instantiated as \( \text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, \text{list})) \) — yields the refined scheme

1. \( A0 \)
2. \( \text{list} =_t \text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, \text{list})) \)
3. \( \text{[cons, (3, [nil, \omega])]} : \text{list} \)
4. \( \upsilon_1 : \Omega \)
5. \( 99 : \text{int} \)
6. \( \upsilon_2 : \text{prod}(\text{int}, \text{list}) \)
7. \( \pi_1, \upsilon_2 : \text{int} \)
8. \( \text{[case} \text{[cons, (3, [nil, \omega])]} : \text{list of} \)
\[ \text{[nil, } \upsilon_1 \text{]} \text{then } 99, \text{[cons, } \upsilon_2 \text{]} \text{then } \pi_1, \upsilon_2 \text{]} : \text{int} \quad (3.8), 2, 3, 5, 7 \)

This scheme contains four gaps still to be closed. Now observe that lines 2 and 3 (under context 1) express the formulae denoted by (f) and (l), respectively, in subsection 2.2. Closing the gaps above these lines essentially amounts to writing down the formulae (a) through (l) again; we shall not do so here.

Concerning the justification of line 5 (under a context constituted by 1 and 4), observe that by rule (3.1) we have \( \epsilon \triangleright 99 : \text{int} \) and, hence, by repeated application of (6.1) and (6.2)

\[ A0, \upsilon_1 : \Omega \triangleright 99 : \text{int} \]

As for line 7 (under context 1 and 6), rule (3.6) does the trick. Thus, we obtain

1. \( 99 : \text{int} \)  \quad (3.1)
2. \( A0 \)
3. \( \text{list} =_t \text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, \text{list})) \)  \quad (f), subsect. 2.2
4. \( \text{[cons, (3, [nil, \omega])]} : \text{list} \)  \quad (l), subsect. 2.2
5. \( \upsilon_1 : \Omega \)
6. \( 99 : \text{int} \)  \quad (6.1), (6.2), 1
7. \( \upsilon_2 : \text{prod}(\text{int}, \text{list}) \)
8. \( \pi_1, \upsilon_2 : \text{int} \)  \quad (3.6), 7
9. \( \text{[case} \text{[cons, (3, [nil, \omega])]} : \text{list of} \)
\[ \text{[nil, } \upsilon_1 \text{]} \text{then } 99, \text{[cons, } \upsilon_2 \text{]} \text{then } \pi_1, \upsilon_2 \text{]} : \text{int} \quad (3.8), 3, 4, 6, 8 \)

In connection with the latter scheme, two remarks are in place. First, notice that the “outside-in” strategy, applied to the judgement in line 6, resulted in the addition of judgement \( 99 : \text{int} \) (under the empty context) in line 1. Second, the justification of lines 3 and 4 in a sense displays a division of labour: the derivations of

\[ A0 \triangleright \text{list} =_t \text{sum}(\text{nil}:\Omega, \text{cons}:\text{prod}(\text{int}, \text{list})) \]
and

\[ A_0 \rightarrow [\text{cons}, \{3, \text{nil}, \omega\}] : \text{list} \]

are regarded as separate concerns, which are carried out elsewhere. This is a common way of importing parts of a derivation.

References


<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits.</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes.</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films.</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate.</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems.</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing</td>
<td>ELISA, A language for formal specification of information systems.</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures.</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben</td>
<td>The partition of an information system in several systems.</td>
</tr>
<tr>
<td>86/05</td>
<td>J.L.G. Dietz</td>
<td>A framework for the conceptual modeling of discrete dynamic systems.</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP.</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth</td>
<td>On proving communication closedness of distributed layers.</td>
</tr>
<tr>
<td>86/08</td>
<td>R. Koymans</td>
<td>Compositional semantics for real-time distributed computing (Inf.&amp;Control 1987).</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language.</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing.</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder’s commentary (IFIP86).</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher</td>
<td>A timed failures model for extended communicating processes.</td>
</tr>
<tr>
<td>Year/Number</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>86/14</td>
<td>R. Koymans</td>
<td>Specifying passing systems requires extending temporal logic.</td>
</tr>
<tr>
<td>87/01</td>
<td>R. Gerth</td>
<td>On the existence of sound and complete axiomatizations of the monitor concept.</td>
</tr>
<tr>
<td>87/02</td>
<td>Simon J. Klaver, Chris F.M. Verberne</td>
<td>Federatieve Databases.</td>
</tr>
<tr>
<td>87/03</td>
<td>G.J. Houben, J.Paredaens</td>
<td>A formal approach to distributed information systems.</td>
</tr>
<tr>
<td>87/04</td>
<td>T. Verhoeff</td>
<td>Delay-insensitive codes - An overview.</td>
</tr>
<tr>
<td>87/05</td>
<td>R. Kuiper</td>
<td>Enforcing non-determinism via linear time temporal logic specification.</td>
</tr>
<tr>
<td>87/06</td>
<td>R. Koymans</td>
<td>Temporele logica specificatie van message passing en real-time systemen (in Dutch).</td>
</tr>
<tr>
<td>87/07</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems with real-time temporal logic.</td>
</tr>
<tr>
<td>87/08</td>
<td>H.M.J.L. Schols</td>
<td>The maximum number of states after projection.</td>
</tr>
<tr>
<td>87/10</td>
<td>T. Verhoeff</td>
<td>Three families of maximally nondeterministic automata.</td>
</tr>
<tr>
<td>87/11</td>
<td>P. Lemmens</td>
<td>Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.</td>
</tr>
<tr>
<td>87/12</td>
<td>K.M. van Hee and A. Lapinski</td>
<td>OR and AI approaches to decision support systems.</td>
</tr>
<tr>
<td>87/13</td>
<td>J.C.S.P. van der Woude</td>
<td>Playing with patterns - searching for strings.</td>
</tr>
<tr>
<td>87/14</td>
<td>J. Hooman</td>
<td>A compositional proof system for an occam-like real-time language.</td>
</tr>
<tr>
<td>87/16</td>
<td>H.M.M. ten Eikelder, J.C.F. Wilmont</td>
<td>Normal forms for a class of formulas.</td>
</tr>
<tr>
<td>87/17</td>
<td>K.M. van Hee, G.-J. Houben, J.L.G. Dietz</td>
<td>Modelling of discrete dynamic systems framework and examples.</td>
</tr>
</tbody>
</table>
87/18  C.W.A.M. van Overveld  An integer algorithm for rendering curved surfaces.
87/19  A.J. Seebregts  Optimalisering van file allocatie in gedistribueerde database systemen.
87/20  G.J. Houben  J. Paredaens  The $R^2$-Algebra: An extension of an algebra for nested relations.
87/21  R. Gerth  M. Codish  Y. Lichtenstein  E. Shapiro  Fully abstract denotational semantics for concurrent PROLOG.
88/01  T. Verhoeff  A Parallel Program That Generates the Möbius Sequence.
88/03  T. Verhoeff  Settling a Question about Pythagorean Triples.
88/04  G.J. Houben  J. Paredaens  D. Tahon  The Nested Relational Algebra: A Tool to Handle Structured Information.
88/06  H.M.J.L. Schols  Notes on Delay-Insensitive Communication.
88/09  A.T.M. Aerts  K.M. van Hee  A Tutorial for Data Modelling.
88/10  J.C. Ebergen  A Formal Approach to Designing Delay Insensitive Circuits.
88/12  A.E. Eiben  Abstract theory of planning.
88/13  A. Bijlsma  A unified approach to sequences, bags, and trees.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>88/14</td>
<td>R.H. Mak</td>
<td>Language theory of a lambda-calculus with recursive types.</td>
</tr>
<tr>
<td>88/15</td>
<td>R. Bos, C. Hemerik</td>
<td>An introduction to the category theoretic solution of recursive domain equations.</td>
</tr>
<tr>
<td>88/16</td>
<td>C. Hemerik, J.P. Katoen</td>
<td>Bottom-up tree acceptors.</td>
</tr>
<tr>
<td>88/17</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable specifications for discrete event systems.</td>
</tr>
<tr>
<td>88/18</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: concepts and basic results.</td>
</tr>
<tr>
<td>88/19</td>
<td>D.K. Hammer, K.M. van Hee</td>
<td>Fasering en documentatie in software engineering.</td>
</tr>
<tr>
<td>88/20</td>
<td>K.M. van Hee, L. Somers, M. Voorhoeve</td>
<td>EXSPECT, the functional part.</td>
</tr>
<tr>
<td>89/1</td>
<td>E.Zs.Lepoeter-Molnar</td>
<td>Reconstruction of a 3-D surface from its normal vectors.</td>
</tr>
<tr>
<td>89/2</td>
<td>R.H. Mak, P. Struik</td>
<td>A systolic design for dynamic programming.</td>
</tr>
<tr>
<td>89/3</td>
<td>H.M.M. Ten Eikelder, C. Hemerik</td>
<td>Some category theoretical properties related to a model for a polymorphic lambda-calculus.</td>
</tr>
<tr>
<td>89/4</td>
<td>J. Zwiers, W.P. de Roever</td>
<td>Compositionality and modularity in process specification and design: A trace-state based approach.</td>
</tr>
<tr>
<td>89/5</td>
<td>Wei Chen, T. Verhoeff, J.T. Udding</td>
<td>Networks of Communicating Processes and their (De-)Composition.</td>
</tr>
<tr>
<td>89/6</td>
<td>T. Verhoeff</td>
<td>Characterizations of Delay-Insensitive Communication Protocols.</td>
</tr>
<tr>
<td>89/7</td>
<td>P. Struik</td>
<td>A systematic design of a parallel program for Dirichlet convolution.</td>
</tr>
<tr>
<td>89/9</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: Dynamic versus static topology.</td>
</tr>
<tr>
<td>89/10</td>
<td>S. Ramesh</td>
<td>A new efficient implementation of CSP with output guards.</td>
</tr>
<tr>
<td>89/11</td>
<td>S. Ramesh</td>
<td>Algebraic specification and implementation of infinite processes.</td>
</tr>
<tr>
<td>Page</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>89/12</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A concise formal framework for data modeling.</td>
</tr>
<tr>
<td>89/14</td>
<td>H.C. Haesen</td>
<td>ELDA, data manipulatie taal.</td>
</tr>
<tr>
<td>89/15</td>
<td>J.S.C.P. van der Woude</td>
<td>Optimal segmentations.</td>
</tr>
<tr>
<td>89/16</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>Towards a framework for comparing data models.</td>
</tr>
<tr>
<td>89/17</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.</td>
</tr>
<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>
</tbody>
</table>