Record concatenation with intersection types

Citation for published version (APA):

Document status and date:
Published: 01/01/1995

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.
Record concatenation with intersection types

by

Jan Zwanenburg

ISSN 0926-4515

All rights reserved
editors: prof.dr. R.C. Backhouse
    prof.dr. J.C.M. Baeten

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

Computing Science Report 95/34
Eindhoven, December 1995
Abstract

We define an extension of a second-order type system with records and operations on them, including record concatenation. Our aim is to model the most important concepts in object-oriented languages in a type system with subtyping. Such a model needs a form of record concatenation. The novelty in our approach is that intersection types are used for record concatenation. We give examples of how object-oriented concepts can be modelled and show how the system can be translated to a type system without subtyping.
1 Introduction

Object-oriented programming (OOP) is concerned with objects, classes and methods. Objects are grouped into classes. Every class has a set of methods that operate on the objects in that class. A class can be extended in such a way that the objects in the new class (the subclass) still can be handled by the methods of the original class (the superclass). This defining property of OOP is called inheritance. It permits easily extensible software systems which is very important for programming in the real.

Our goal is to model the most important concepts of OOP in the framework of the typed lambda-calculus, because it has well known semantics, proof-rules and typing rules, which are all (largely) missing in object-oriented languages. By such a model we hope to gain insight in these issues.

But pure lambda calculus is not rich enough to model OOP. To implement objects, records are needed, and to implement some aspects of inheritance, subtyping is needed. A type is a subtype of another type if there is a natural coercion from the former type to the latter type. Even in a calculus with records and subtyping, two problems remain.

One is a consequence of our framework being functional and not imperative. It is not possible to make a function that returns a copy of a record with a certain field changed, working on all records with that field. This is called the polymorphic record update problem. An example is a function that increases the $x$-field of a record by one. We need to write it as a polymorphic function with some demand on the type variable:

$$
\text{up} = \lambda P \text{ "with } x\text{-field of type } \text{Int}". \lambda p : P. \ "p \text{ with } x\text{-increased by one}"
$$

$$
: \forall P \text{ "with } x\text{-field of type } \text{Int}". P \rightarrow \text{P}
$$

$$
\text{up} (\{x: \text{Int}, y: \text{Int} \} \{x=3, y=5\}) = \{x=4, y=5\} : \{x: \text{Int}, y: \text{Int} \}
$$

The demand “with $x$-field of type $\text{Int}$ cannot be expressed by subtyping laws for records.

The other problem is that in order to model inheritance, a form of record concatenation or extension is needed.

Record extension is the extension of a record with one field. By doing this repeatedly, a record can be extended with a certain number of fields. The record concatenation of two records is a new record, that has all the fields that any of the constituents have. This is more flexible than record extension, and is necessary for multiple inheritance (where a class may have more than one superclass).

Wand [Wan 87] gave a system of extensible records in an implicitly typed language. This was later extended to include concatenation and the type-inference problem was solved [Wan 89]. Cardelli and Wegner [CW 85] defined a full second-order, explicitly typed language with records, subtyping and bounded quantification, i.e. an abstraction over a type variable that has to be a subtype of some type.

To solve the polymorphic record update problem, more powerful explicitly typed calculi were defined. Most introduced a form of disjoint concatenation (only new fields may be added), and a way of restricting records (removing a field). Together these operations can provide a polymorphic update. But the ones that include subtyping [CM 91, Car 92] have no full record concatenation, just extension. Alternatives that do have record concatenation [HP 90], do not have subtyping, which is necessary for a natural model of inheritance. Most problems are caused by type variables that represent records.

In this paper, we model concatenation by intersection types, developed by Coppo and Dezani-Ciancaglini [CD 78]. This choice is based on the fact that, given two record types $R$ and $S$, the concatenation of $R$ and $S$ should be a subtype of $R$ and a subtype of $S$. This is one of the defining property of the intersection of $R$ and $S$. More formally, the intersection is defined as the greatest lower bound in the subtype hierarchy:

$$
R \cap S \subseteq R \quad R \cap S \subseteq S \quad T \subseteq R \text{ and } T \subseteq S \Rightarrow T \subseteq R \cap S
$$

All three rules are also valid if one reads record concatenation for $\wedge$, so the interpretation of intersection types for concatenation is correct.

If we permit concatenation of arbitrary record values, the system becomes unsound. Therefore we permit the concatenation of two record values only if the record types have the same type on common fields. This relation between types $R$ and $S$ is denoted by $R \# S$ ( $R$ compatible with $S$). The meaning of a concatenation on values is that for common fields preference
is given to the right value. In this way the polymorphic record update problem is solved in presence of both full record concatenation and subtyping.

Our concatenation doesn't require both records to be disjoint, therefore eliminating the need for record restriction. But it is more stringent than a concatenation without any demands on its arguments, which leads to inconsistencies, when coupled with subtyping in an explicitly typed calculus.

In section 2 we will, starting with a pure second-order lambda calculus, gradually extend the system with records, subtyping, intersection types and compatibility. Section 3 will show how the most important concepts of OOP can be modelled in our language. Again, we will do this in a gradual fashion. The more complex models will need some constructions not formally described in this paper (type constructors, existential types and the fixpoint combinator, all well known). The semantics of our system are described in section 4. This is based on a translation to the second-order lambda calculus with cartesian products.

2 Development

In this section we will present the second-order lambda calculus, and extend it with records, subtyping, the Top type, intersection types and compatibility. From section 2.6 onwards we will only extend the compatibility rules so as to make the record concatenation more flexible.

In each subsection we will only give the new rules, and designate which old rules are redundant. In appendix A the total system is given.

2.1 Second-order lambda calculus

The starting point is \( F \), the second-order lambda calculus [Gir 72]. In this calculus it is possible to make polymorphic functions, i.e. functions that work on several types.

Rules:

Types:
\[
T ::= X \quad \text{type variable}
T \Rightarrow T \quad \text{function type}
\forall X : T \quad \text{quantification}
\]

Terms:
\[
e ::= x \quad \text{variable}
\lambda x : T. e \quad \text{abstraction}
e e \quad \text{application}
\lambda X : T. e \quad \text{type abstraction}
e T \quad \text{type application}
\]

Contexts:
\[
\Gamma ::= \emptyset \quad \text{empty context}
\Gamma, x : T \quad \text{term variable declaration}
\Gamma, X : T \quad \text{type variable declaration}
\]

Judgments:
\[
\Gamma \vdash \text{ok} \quad \text{well-formed context}
\Gamma \vdash T : * \quad \text{well-formed type}
\Gamma \vdash e : T \quad \text{well-typed term}
\]

Meta-variables:
\[
\Gamma \quad \text{for contexts}
X \quad \text{for type variables}
R, S, T, U, V \quad \text{for types}
x \quad \text{for term variables}
e, f \quad \text{for terms}
\]

Derivation rules:
Context formation:

\[ \emptyset \vdash ok \quad (C-Empty) \]
\[ \Gamma \vdash T : * \quad x \notin \text{dom}(\Gamma) \quad (C-Var) \]
\[ \Gamma, x : T \vdash ok \quad (C-Var) \]
\[ \Gamma \vdash ok \quad X \notin \text{dom}(\Gamma) \quad (C-TVar) \]

Type formation:

\[ \Gamma_1, X : *, \Gamma_2 \vdash ok \quad (K-TVar) \]
\[ \Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : * \quad (K-Arrow) \]
\[ \Gamma \vdash T_1 \rightarrow T_2 : * \quad (K-Arrow) \]
\[ \Gamma, X : * \vdash T : * \quad (K-All) \]

Term formation:

\[ \Gamma_1, x : T, \Gamma_2 \vdash ok \quad (T-Var) \]
\[ \Gamma, x : S \vdash e : T \quad (T-Abs) \]
\[ \Gamma \vdash (\lambda x : S. e) : S \rightarrow T \quad (T-Abs) \]
\[ \Gamma \vdash f : S \rightarrow T \quad \Gamma \vdash e : S \quad (T-App) \]
\[ \Gamma \vdash f : T \quad \Gamma \vdash e : T \quad (T-App) \]
\[ \Gamma \vdash (\lambda X : S. e) : \forall X : *, T \quad (T-TAbs) \]
\[ \Gamma \vdash f : \forall X : *, T \quad \Gamma \vdash U : * \quad (T-TApp) \]
\[ \Gamma \vdash f : T \quad \Gamma \vdash U : T[X := U] \]

Reduction rules:

\[ (\lambda x : t. e_1) \ e_2 \rightarrow \ _ \ e_1[x := e_2] \]
\[ (\lambda X : t. e) \ T \rightarrow \ _ \ e[X := T] \]

In examples we will assume there are types \textit{Real}, \textit{Int} and \textit{Char}, and the usual terms of these types. More formally, we will consider \textit{Real:*}, \textit{Int:*}, \textit{Char:*}, \textit{+:Int\rightarrow Int}, \textit{0 : Int}, \textit{1 : Int} etc. to be always part of the context.

Convention: We write \[ \vdash e : T \] for \[ \emptyset \vdash e : T \] etc.

Examples:

1. \[ \vdash (\lambda X : *. \lambda z : X. z) : \forall X : *. X \rightarrow X \] the polymorphic identity
2. \[ \vdash 7.00 : \textit{Real} \]
3. \[ \vdash 7 : \textit{Int} \]
4. \[ \vdash 'a' : \textit{Char} \]

Limitations: There is no tuple datastructure.
2.2 Records

When objects and classes are modelled in a \(\lambda\)-calculus, records are used. Therefore we will first extend our system with basic record operations, i.e. the construction of a record and the selection of a field of a record.

We introduce a set of labels \(\mathcal{L}\) (with meta-variables \(l, m\)) and add the following rules to the system:

Rules:

\[
\begin{align*}
\text{all } l, i \in \mathcal{L} \text{ distinct labels} & \quad \text{for } i = 1..n \quad \Gamma \vdash \{i_1:T_1, \ldots, i_n:T_n\} : \ast \\
\text{(K-Rec)}
\end{align*}
\]

\[
\begin{align*}
\text{for } i = 1..n \quad \Gamma \vdash e_i : T_i & \quad \Gamma \vdash \{i_1:T_1, \ldots, i_n:T_n\} : \ast \\
\Gamma \vdash e : \{i_1:T_1, \ldots, i_n:T_n\} & \quad 1 \leq i \leq n \\
\text{(T-Rec)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : T_i & \quad \Gamma \vdash e_i : T_i \\
\text{(T-Scl)}
\end{align*}
\]

Reduction is extended by:

\[
\{i_1=e_1, \ldots, i_n=e_n\} \vdash \eta \triangleright e_i
\]

We do not identify records that differ only in the order of the fields (unlike most other calculi do), because we will model them with unlabelled products.

Examples:

1. \(\Gamma \vdash \lambda p.\{x: \text{Int}, y: \text{Int}\}. \ p.x + p.y : \{x: \text{Int}, y: \text{Int}\} \rightarrow \text{Int}\)
2. \(\Gamma \vdash \lambda p.\{x: \text{Int}, y: \text{Int}\}. \ \{x=p.x + 1, y=p.y\} : \{x: \text{Int}, y: \text{Int}\} \rightarrow \{x: \text{Int}, y: \text{Int}\}\)
3. \(\Gamma \vdash \{l=\{p=7, q='a'\}, m=3.14\} : \{l: \{p: \text{Int}, q: \text{Char}\}, m: \text{Real}\}\)
   (this is called a nested record)
4. \(\{l=6\}. l \triangleright 6\)

Limitations: The typing is very rigid and there is no structure in the record types that can help to overcome this problem. E.g. we cannot apply the function of example 2 to \(\{x=3, y=4, z=5\}\) or to \(\{y=4, x=3\}\).

2.3 Subtyping

The standard way to alleviate the rigid typing is to introduce subtyping, see e.g. [Car 88] for subtyping on records. Subtyping is a relation between types, denoted by \(\leq\). If \(S\) is a subtype of \(T\), it means that a term of type \(S\) can be used at every place where a term of type \(T\) can be used, because there is a natural way of coercing a value of type \(S\) to type \(T\).

Subtyping is used to model one crucial aspect of inheritance, viz. that an object of a subclass may be used in any place where an object of the superclass is used. As objects are modelled by records, and an object of a subclass has more fields than an object of a superclass, the subtyping relationship we need is that a recordtype with more fields is a subtype of a recordtype with less fields. Of course, there is a natural coercion from the first to the latter: forget some fields. This rule for subtyping is formalized by (S-Width) below.

Sometimes it is necessary to demand that a type variable in a polymorphic function is a subtype of some type \(T\). This is called bounded quantification: The type variable is noted with its bound in the abstraction or polymorphic type.

Here we will present the extension of the rules to realize subtyping.

Rules:

Types:

\[
T ::= \forall X \leq T. \ T \quad \text{bounded quantification}
\]

Terms:

\[
e ::= \forall X \leq T. \ e \quad \text{type abstraction}
\]
Contexts:
\[ \Gamma ::= \Gamma, X \leq T \]  
\( \) type variable declaration

Judgments:
\[ \Gamma \vdash S \leq T \]  
\( \) subtype

Derivation rules:

\textit{Context formation:}
\[ \frac{\Gamma \vdash T : * \quad X \notin \text{dom}(\Gamma)}{\Gamma, X \leq T \vdash \text{ok}} \]  
\( \) (C-Var)

\textit{Type formation:}
\[ \frac{\Gamma_1, X \leq T, \Gamma_2 \vdash \text{ok}}{\Gamma_1, X \leq T, \Gamma_2 \vdash X : *} \]  
\( \) (K-Var)
\[ \frac{\Gamma, X \leq T \vdash U : *}{\Gamma \vdash \forall X \leq T. \ U : *} \]  
\( \) (K-All)

\textit{Subtyping:}
\[ \frac{\Gamma \vdash T : *}{\Gamma \vdash T \leq T} \]  
\( \) (S-Refl)
\[ \frac{\Gamma \vdash S \leq T \quad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U} \]  
\( \) (S-Trans)
\[ \frac{\Gamma_1, X \leq T, \Gamma_2 \vdash \text{ok}}{\Gamma_1, X \leq T, \Gamma_2 \vdash X \leq T} \]  
\( \) (S-Var)
\[ \frac{\Gamma \vdash S_1 \leq S_2 \quad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \]  
\( \) (S-Arrow)
\[ \frac{\Gamma, X \leq U \vdash S \leq T}{\Gamma \vdash \forall X \leq U. \ S \leq \forall X \leq U. \ T} \]  
\( \) (S-All)
\[ \frac{\Gamma \vdash \{ l_1 : T_1, \ldots, l_n : T_n \} : * \quad n \geq m}{\Gamma \vdash \{ l_1 : T_1, \ldots, l_m : T_m \}} \]  
\( \) (S-Width)
\[ \frac{\{ \text{for } i = 1..n \quad \Gamma \vdash S_i \leq T_i \}}{\Gamma \vdash \{ l_1 : T_1, \ldots, l_n : T_n \}} \]  
\( \) (S-Depth)
\[ \frac{f : 1..n \rightarrow 1..n \text{ bijection}}{\Gamma \vdash \{ l_1 : T_1, \ldots, l_n : T_n \} \leq \{ l_{R(1)} : T_{R(1)}, \ldots, l_{R(n)} : T_{R(n)} \}} \]  
\( \) (S-Order)

\textit{Term formation:}
\[ \frac{\Gamma, X \leq S \vdash e : T}{\Gamma \vdash \lambda X \leq S. \ e : \forall X \leq S. \ T} \]  
\( \) (T-Tabs)
\[ \frac{\Gamma \vdash f : \forall X \leq S. \ T \quad \Gamma \vdash U \leq S}{\Gamma \vdash f \ U : T[X := U]} \]  
\( \) (T-TApp)
\[ \frac{\Gamma \vdash e : S \quad \Gamma \vdash S \leq T}{\Gamma \vdash e : T} \]  
\( \) (T-Subsumption)
\[ \frac{\Gamma \vdash e \ : \ \{ l_i : T_i \}}{\Gamma \vdash e l_i : T} \]  
\( \) (T-Sel)

Equality:
\( (\lambda x : t. \ e_1) \ e_2 = e_1[x := e_2] \)
\( (\lambda X : S. \ e) = e[X := T] \)
\( (\lambda X \leq T_1. \ e) \ T_2 = e[X := T_2] \)
\( \{ l_1 = e_1, \ldots, l_n = e_n \} \ l_i = e_i \)
Notation:
\[ \Gamma \vdash S \sim T \] is an abbreviation for \[ \Gamma \vdash S \leq T \text{ and } \Gamma \vdash T \leq S. \]
This relation is called equivalence.

Discussion:
All rules except (S-All) and the ones for records, viz. (S-Width), (S-Depth), (S-Order) and (T-Sel) are standard.

There has been a long debate about the (S-All) rule (for a summary, see [SP 94]). The following is considered the most natural rule:

\[
\frac{\Gamma \vdash V \leq U \quad \Gamma, X \leq V \vdash S \leq T}{\Gamma \vdash \forall X \leq U, S \leq \forall X \leq V. T} \quad \text{(S-All')}
\]

Unfortunately, the subtyping problem of the system with this rule is undecidable [Pie 94], so we stick to the (S-All) rule.

Rule (T-Sel) is a short rule for the selection of a field of a record in a calculus of subtyping and makes the old (T-Sel) rule redundant. Rule (S-Width) expresses that a record with more fields is a subtype of a record with less fields, and (S-Depth) expresses that a record can be subtyped field-wise. Normally these two rules are presented as one, but we won't do this to make the different aspects of subtyping for records clear. In other calculi rule (S-Order) is not present: records are considered modulo permutation of the fields. Here records with fields in a different order are equivalent, but not the same. From an implementational point of view, where records are modelled as unlabelled tuples this is the most natural. The computer has to apply a coercion, i.e. a permutation of the fields, but we don't want the programmer having to write this coercion explicitly.

In examples we will consider the declaration of the integers in the context to be \( \text{Int} \leq \text{Real} \).

Examples:
1. \( \vdash \{k:\text{Int}, l:\text{Int}, m:\text{Int}\} \leq \{k:\text{Int}\} \) (use S-order and S-width)
2. \( \vdash \text{Real} \rightarrow \text{Int} \leq \text{Int} \rightarrow \text{Real} \)
3. \( \vdash (\lambda r:\{l:\text{Int}\}, r.l) \{k=4, l=5, m=6\} : \text{Int} \) (use subtyping derivation from example 1)
4. \( \vdash (\lambda p:\{x:\text{Int}, y:\text{Int}\}, p.x + p.y)\{x=4, y=5, z=6\} : \text{Int} \)
5. \( \vdash (\lambda p:\{x:\text{Int}\}, \{x=p.x + 1\})\{x=4, y=5\} : \{x:\text{Int}\} \)
6. \( \vdash \lambda P. \{x=p.x + 1\} : \forall P \leq \{x:\text{Int}\}. P \rightarrow \{x:\text{Int}\} \)
   As is clearly visible from the typing, this function always delivers a one-field record.
   Effectively, this function is the same as the one of example 5.
7. \( \vdash \{l:\{p:\text{Int}, q:\text{Char}\}, m:\text{Real}\} \leq \{l:\{p:\text{Int}\}\} \)

Discussion:
A major consequence of the introduction of subtyping is that \( \beta \)-reduction doesn't fully describe the operational meaning of a program anymore, since coercions are left implicit. E.g. we would like to have

\[ (\lambda x : \text{Real}. x)7 \Downarrow 7.000 \]

One way to solve this is to place coercions explicitly in the terms (section 4). However, to give the reader an idea of the meaning of terms, we have given equality rules and we will give equality rules for records in the following sections. These rules do not compose a full equational system for our calculus, this is outside the scope of this article. A more indirect way of formally deriving equality is presented in section 4.

Limitations: We would like to make a polymorphic record update, a polymorphic function that changes one field of a record, and leaves all other fields untouched. Examples 5 and 6 try to do this, but don't succeed. In [BL 90] is a semantic argument that such a function cannot be made in calculi like this one, and, more generally, a function of type \( \forall X \leq T. X \rightarrow X \) must be an identity function.
2.4 Top type for \( \leq \)

The type \( \text{Top} \) is introduced to recover an \( F \) polymorphic function as a special case of bounded quantification, viz. with bound \( \text{Top} \). This is done to reduce the number of derivation rules, and make the system more uniform.

We add to our system the type \( \text{Top} \) with derivation rules:

Rules:

\[
\begin{align*}
\Gamma \vdash \text{ok} & \quad \text{(K-Top)} \\
\Gamma \vdash S : \ast & \quad \text{(S-Top)} \\
\Gamma \vdash \text{ok} & \quad \text{(T-Top)}
\end{align*}
\]

The rules (C-TVar), (K-All), (T-TAbs) and (T-TApp) of section 2.1 are redundant now as they can be replaced by their counterparts of section 2.3 with bound set to \( \text{Top} \). \( \text{Top} \) has one element: \( \text{top} \), so a coercion of any value to type \( \text{Top} \) will yield \( \text{top} \). Categorically, \( \text{Top} \) is the terminal element of the set of types.

Examples:

1. \( \vdash \forall X \leq \text{Top}. \lambda x: X. x : \forall X \leq \text{Top}. X \rightarrow X \)

   the polymorphic identity

2.5 Record concatenation

An operation that occurs frequently in OOP is extending the methods of a superclass with some new methods to make a subclass. As the set of methods of a class is represented by a record, this operation has to be represented by record extension: adding fields to a record. When a subclass has more than one superclass, i.e. in the case of multiple inheritance, all records of the superclasses have to be concatenated.

Therefore we will introduce record concatenation operators, \( \wedge \) for record types and \( \text{with} \) for record values. We denote the concatenation with intersection, because the first has the same subtyping properties as the latter: the concatenation of record types \( S \) and \( T \) is a subtype of both \( S \) and \( T \). To prevent confusion, concatenation is only permitted if there are no common fields. This notion is formalized by the relation \( \# \) (pronounce: compatible).

Note: \( R \# S \) expresses that \( R \) and \( S \) are record types and don’t have any fields in common.

Just as we needed bounded quantification when we introduced the relation \( \leq \), we need restricted quantification now. This is a quantification where there is a restriction that the type variable must be compatible to a certain type. This means we have compatibility relations in contexts as well.

If we have record concatenation, we don’t need multi-fields record as a primitive anymore since they can be constructed by repeatedly concatenating single-field records. However, we do need the record with no fields as a primitive. Because of these new primitives, we need other reduction rules for records.

All this is formalized in the following rules, as extension of the previous system. After the rules we will give a discussion, where we show which rules of the previous system are redundant (but still valid).

Rules:

Types:

\[
T ::= \forall X \leq T; X \# T. T \quad \text{restricted quantification} \\
\emptyset \quad \text{empty record type} \\
\{L:T\} \quad \text{one-field record type} \\
T \wedge T \quad \text{record concatenation}
\]
Terms:
\[ e ::= \lambda X{T}; X \neq T. e \quad \text{type abstraction} \]
\[ \{ \} \quad \text{empty record} \]
\[ \{ e \} \quad \text{one-field record} \]
\[ e \{ T \} \quad \text{concatenation} \]
\[ e \cdot e \quad \text{field selection} \]

Contexts:
\[ \Gamma ::= \Gamma, X \leq T; X \neq T \quad \text{type variable declaration} \]

Judgments:
\[ \Gamma \vdash S \# T \quad \text{compatible} \]

Derivation rules:

Context formation:
\[
\Gamma \vdash T : * \quad \Gamma \vdash U : * \quad X \notin \text{dom}(\Gamma) \quad \frac{}{\Gamma, X \leq T; X \neq U \vdash \text{ok}} \quad (\text{C-TVar})
\]

Type formation:
\[
\Gamma, X \leq T; X \neq U, \Gamma_2 \vdash \text{ok} \quad \frac{}{\Gamma, X \leq T; X \neq U, \Gamma_2 \vdash X : *} \quad (\text{K-Var})
\]
\[
\Gamma, X \leq T; X \neq U \vdash S : * \quad \frac{}{\Gamma \vdash (\forall X \leq T; X \neq U. S) : *} \quad (\text{K-All})
\]
\[
\Gamma \vdash \text{ok} \quad \frac{}{\Gamma \vdash \{ \} : *} \quad (\text{K-Empty})
\]
\[
\Gamma \vdash T : * \quad \frac{}{\Gamma \vdash \{ T \} : *} \quad (\text{K-Single})
\]
\[
\Gamma \vdash R : * \quad \Gamma \vdash S : * \quad \frac{}{\Gamma \vdash R \wedge S : *} \quad (\text{K-With})
\]

Subtyping:
\[
\Gamma, X \leq T; X \neq U, \Gamma_2 \vdash \text{ok} \quad \frac{}{\Gamma, X \leq T; X \neq U, \Gamma_2 \vdash X \leq T} \quad (\text{S-TVar})
\]
\[
\Gamma, X \leq U; X \neq V \vdash S \leq T \quad \frac{}{\Gamma \vdash (\forall X \leq U; X \neq V. S) \leq (\forall X \leq U; X \neq V. T) \quad (\text{S-All})}
\]
\[
\Gamma \vdash \{ T \} : * \quad \frac{}{\Gamma \vdash \{ T \} \leq \{ \} \quad (\text{S-Empty})}
\]
\[
\Gamma \vdash R \leq S \quad \frac{}{\Gamma \vdash \{ R \} \leq \{ S \} \quad (\text{S-Depth})}
\]
\[
\Gamma \vdash S_1 \wedge S_2 : * \quad i = 1, 2 \quad \frac{}{\Gamma \vdash S_1 \wedge S_2 \leq S_i \quad (\text{S-Inter-LB})}
\]
\[
\Gamma \vdash R \leq S \quad \Gamma \vdash R \leq T \quad \frac{}{\Gamma \vdash R \leq S \wedge T \quad (\text{S-Inter-G})}
\]

Compatibility rules:
\[
\Gamma, X \leq T; X \neq U, \Gamma_2 \vdash \text{ok} \quad \frac{}{\Gamma, X \leq T; X \neq U, \Gamma_2 \vdash X \neq U} \quad (\# \text{-TVar})
\]
\[
\Gamma \vdash \{ R \} \leq \{ \} \quad \frac{}{\Gamma \vdash \{ R \} \# \{ \} \quad (\# \text{-Empty-L})}
\]
\[
\Gamma \vdash R \leq \{ \} \quad \frac{}{\Gamma \vdash R \# \{ \} \quad (\# \text{-Empty-R})}
\]
Term formation:

\[
\begin{align*}
\Gamma \vdash T : \ast & \quad \Gamma \vdash U : \ast & (\# - \text{Diff}) \\
\Gamma \vdash \{l:T\} \# \{m:U\} & (\# - \text{Inter-L}) \\
\Gamma \vdash R \# T & \quad \Gamma \vdash S \# T & (\# - \text{Inter-R}) \\
\Gamma \vdash R \# S & \quad \Gamma \vdash R \# T & (\# - \text{Inter-L}) \\
\Gamma \vdash R \# S \wedge T & (\# - \text{Inter-R})
\end{align*}
\]

Equations:

\[
\begin{align*}
(\lambda X \leq U; X \# T. e) T & = e[X := T] \\
\{l=e\} & \ast = e \\
r \ast \text{ with } (\{\} \text{ \ast}) s & = r \\
\{\ast \text{ with } (l:T) \text{ \ast}\} & = \{s \ast \text{ with } (l:1)\} \\
\{\ast \text{ with } (S \wedge T) \text{ \ast}\} & = \{s \ast \text{ with } (T \wedge t)\} \\
\{r \ast \text{ with } (S \wedge T) \text{ \ast}\} & = \{r \ast \text{ with } (S \wedge T) \text{ \ast}\} \\
\end{align*}
\]

Discussion: We will discuss the new derivation rules sort by sort, in the context of records and record-operations.

The type formation rule (K-\text{With}) permits the intersection of any two types. In examples we will use it only for recordtypes; maybe the restriction of (K-\text{With}) to recordtypes has better theoretical properties, and should be preferred.

The subtyping rule (S-\text{Inter-LB}) expresses that a concatenation of two records is always a subtype of each of the records, since it has all fields that the constituents have, with the corresponding types. Together with rule (S-\text{Empty}) it replaces the rule (S-\text{Width}) of section 3. Rule (S-\text{Inter-G}) means that a record that is a subtype of two other records, must be a subtype of the concatenation of these two records, since it has all of the fields of the two records, with appropriate types. This rule can be used together with (S-\text{Inter-LB}) to derive that records with their fields in a different order are equivalent, and make rule (S-\text{Order}) of section 3 redundant in this way. Rule (S-\text{Depth}) together with rules (S-\text{Inter-LB}) and (S-\text{Inter-G}) can be used to derive the field-wise subtyping rule for records, and replace the previous (S-\text{Depth}) rule.

A record may always be extended with the empty record, since they have trivially no field in common. This is expressed by rules (\# - \text{Empty-L}) and (\# - \text{Empty-R}). Two single-field records are compatible if their field-names (labels) are different, this is rule (\# - \text{Diff}). Finally, a concatenation of two records has no fields in common with a third record if both of the first two are disjoint with the third (\# - \text{Inter-L} and -R). This completes the formalization of the predicate "no fields in common" into \#. Note that we don't have a symmetry rule. The reasons for this will be given in section 2.7 and in section 4.

Finally, rule (T-\text{With}) permits the concatenation of record values. Note that we decorate the with-operator with the types of the arguments. The values will be coerced to the corresponding types before they are concatenated. The type decorations are necessary because of the subtyping. If we didn't have them every concatenation of records would be permitted because every record is of type (\{\} \ast) and (\{\} \# (\{\} \ast)). This introduction rule for intersection types
seems rather unusual. However, we can see it as the combination of the standard introduction rule and a typing rule for the `with`-construction.

\[
\frac{\Gamma \vdash r_1 : R_1 \quad \Gamma \vdash r_2 : R_2 \quad \Gamma \vdash R_1 \not\equiv R_2 \quad i = 1, 2}{\Gamma \vdash r_1 \mathbin{\text{with}} R_2 r_2 : R_i} \quad (\text{T-With'})
\]

\[
\frac{\Gamma \vdash e : R \quad \Gamma \vdash e : S}{\Gamma \vdash e : R \mathbin{\text{and}} S} \quad (\text{T-Inter'})
\]

However, for practical purposes, the given rule suits us better. The equations we give here seem rather complex, compared to the previous reduction rule for records, but remember that formally these rules are not relevant and serve only to give the reader an intuition about the meaning of concatenation. The rules depend on information in the types concerning presence or absence of fields. If a field is selected, the right-most field of the record is examined (later on the reasons for this will become clear). We will start our explanation of the equations with the third one. A record \( r \) concatenated with a record of the empty type stays the same record \( r \).

A record extended with one field, of which a field is selected, can be equal to two things. If the selected field is in the extension, we can forget the rest of the record. Otherwise, we can forget the extension. The last equation says `with` is associative, so brackets may be moved to the left. This will eventually lead to the case that the last object of concatenation is a single field record, so that the selection of a field of a record will be the value of the field, irrespective of the way the record was constructed.

### Examples:

1. \( \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \not\equiv \{m: \text{Char}\} \quad \{m='a'\} : \{11: \text{Int}\} \land \{m: \text{Char}\} \)

   We have constructed a record with two fields from two single-field records.

2. \( \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{m: \text{Char}\} \not\equiv \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{m='a'\} \quad \{11: \text{Int}\} \land \{m=5\} : \{11: \text{Int}\} \land \{m='a'\} \land \{m=5\} \)

   This is a record with 3 fields.

3. \( \{\} \not\equiv \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{y: \text{Real}\} \quad \{y='a'\} : \{\} \land \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{y='a'\} \)

   So it is possible to concatenate two records of two fields each.

4. \( \lambda r. r \not\equiv R R = \lambda \{\} \{m: \text{Char}\} \quad \{m='a'\} : \{\} \land \{m: \text{Char}\} \)

   A concatenation with the empty record.

5. \( \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \not\equiv \{m: \text{Char}\} \land \{l: \text{Int}\} \quad \text{and vice versa, so} \quad \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \not\equiv \{m: \text{Char}\} \land \{l: \text{Int}\} \)

   Changing the order of the fields results in an equivalent type.

6. \( R \leq \{\} \vdash R \sim RA \{\} \)

   Any record type \( R \) is equivalent with \( R \) extended by the empty record.

7. \( \vdash \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \not\equiv \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{m: \text{Char}\} \)

   Depth subtyping of a multifield record.

8. \( \vdash \lambda R \leq T p : R \not\equiv \{y: \text{Int}\}. \quad \lambda r. r : R \not\equiv \{y=0\} : YR \leq T p : R \not\equiv \{y: \text{Int}\}. \quad \text{The function accepts any record type} \ R \quad \text{without} \ y \text{-field, a record of this type, and append} \ y \text{-field with value} \ 0 \text{to this record.} \)

9. \( \vdash \lambda R \leq \{x: \text{Int}\} : R \not\equiv \{y: \text{Int}\}. \quad \lambda r. r \not\equiv \{y=0\} \quad \{y=r.x\} : YR \leq \{x: \text{Int}\} : R \not\equiv \{y: \text{Int}\}. \quad \text{Here the possible use of two demands on the type variable is shown.} \)

10. We have the following equalities

    \[
    \begin{align*}
    (((1 \mathbin{\text{=}} 3) \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{p=7\}).1 & = ((1 \mathbin{\text{=}} 3) \{1 \mathbin{\text{=}} 3\} \{11: \text{Int}\} \land \{p=7\}).1 \\
    & (1 \mathbin{\text{=}} 3) \{11: \text{Int}\} \{n=5\}).1 & = (1 \mathbin{\text{=}} 3) \{11: \text{Int}\} \{n=5\}).1\)
    \end{align*}
    \]
Now it is easy to see this system is a conservative extension of the previous system since each record can be built up from one-field records with the record concatenation. To reduce the clutter of types in big records, we introduce the following abbreviation:

**Abbreviation:**

Let all \( i \) be different labels and let \( e_i \) have type \( T_i \), then

\[
\{l_1:T_1, l_2:T_2, \ldots, l_n:T_n\} := \\
(\{l_1:T_1\} \land \{l_2:T_2\}) \ldots \land \{l_n:T_n\}
\]

\[
\{l_1=e_1, l_2=e_2, \ldots, l_n=e_n\} := \\
(\{l_1=e_1\} \{l_1:T_1\} \text{ with } \{l_2:T_2\} \{l_2=e_2\}) \{l_1:T_1, l_2:T_2\} \ldots \text{ with } \{l_n:T_n\} \{l_n=e_n\}
\]

**Examples:**

1. \( \{l=3, m='a', n=3.5\} : \{l: \text{Int}, m: \text{Char}, n: \text{Real}\} \)
   This is the abbreviation for the term of example 2 above.

2. \( \{l=5\} \{l: \text{Int}\} \text{ with } \{m: \text{Char}\} \{l=6, m='a'\} \).
   \[ l = \{l=5\}.l \]
   The right record value is coerced to type \( \{m: \text{Char}\} \), so there is no conflict.

Note that derivation rules (K-Rec), (T-Rec) and (T-Set) and the reduction rule for records of section 2 are derivable now.

In the following sections we will extend the compatibility rules so `with` becomes a more powerful operation.

**Limitations:** We still have no polymorphic record update

### 2.6 Top type for #

Just as we could see ordinary quantification as a special case of bounded quantification, we can see bounded quantification as a special case of restricted quantification. In this way we can reduce the number of derivation rules again.

We add the derivation rules:

**Rules:**

\[
\frac{}{\Gamma \vdash R : *} \quad (\#-Top-L)
\]

\[
\frac{}{\Gamma \vdash R : * \text{ with } Top \Rightarrow R \# \text{ with } \Gamma \vdash \text{Top} : R} \quad (\#-Top-R)
\]

**Equations:**

\[ e \text{ with } \text{Top} f = e f \text{ with } \text{T} e = e \]

**Discussion:** Now every bounded quantification can be written as restricted quantification with the restriction set to top, e.g. \( \lambda X : \text{Real. } e \) will be written now as \( \lambda X : \text{Real. } X \# \text{Top. } e \). The rules (C-TVar), (K-All), (T-TAbs) and (T-TApp) of section 3 are redundant now as they can be replaced by their counterparts of section 5 with restriction set to \( \text{Top} \). This has as a consequence that it is permitted to concatenate an arbitrary value \( x \) with a value of type \( \text{Top} \), but this will be the same as \( x \).

The attentive reader has remarked that the rules for \( [] \) and \( \text{Top} \) are very similar. Indeed, \( \text{Top} \) can be seen as the intersection of zero types, which is the same as the concatenation of zero fields, i.e. the empty record. There is no need to keep them separated in this calculus, so we will define the empty record as an abbreviation.
Abbreviation:
\[
\begin{align*}
\emptyset & := \text{Top} \\
\bot & := \text{top}
\end{align*}
\]

Examples:
1. \(\vdash (\forall X \leq \text{Top}; X \# \text{Top}. \lambda x. X. x) : \forall X \leq \text{Top}; X \# \text{Top}. X \rightarrow X\)
   is the polymorphic identity
2. \(\vdash 3 \text{Int with Top top} : \text{Int} \cap \text{Top}\)
   this term is equal to 3

From now on we will add only compatibility rules. The complete system is given in appendix A.

2.7 Record updating

The polymorphic record update problem is the problem of writing a polymorphic function that has a record as argument and returns the same record with one field changed. The polymorphism means here that the function can handle different types of records, e.g. with a different number of fields. As we have seen in section 2.3, subtyping is not sufficient for such a function. This problem is a consequence of our framework being functional and not imperative.

The solution we present is to allow the concatenation operator to work as well on two records with common fields, and give then preference to the right argument. So it is possible to update a field \(f\) of a record with value \(e\), by concatenating it with the single-field record \(\{l=e\}\). Such a concatenation is only permitted when the common fields have the same type, so the intuition for changes.

Note:
- \(A \# B\) means: \(A\) and \(B\) agree on common fields.

To implement this extension, we have to add a compatibility rule:

Rules:
\[
\begin{align*}
\Gamma \vdash [\ell; T] : * & \quad \text{(}#\text{-Equal)} \\
\Gamma \vdash [\ell; T] \# [\ell; T] & \quad \text{(}#\text{-Equal)}
\end{align*}
\]

The equality rules of section 2.5 are chosen in such a way that the right-most occurrence of the selected field is chosen.

Note: The demands \(X \leq [\ell; A]\) and \(X \# [\ell; A]\) together mean that \(X\) has an \(l\)-field of type \(A\) (and not a subtype of \(A\)). In this way, the polymorphic update can be expressed, so this pair of demands will be used a lot. More generally, \(X \leq R\) and \(X \# R\) can be interpreted as:
- \(X\) has all the fields of \(R\), with the same type.

Abbreviation: We write
\[
\begin{align*}
\lambda X \leq T. e & := \lambda X \leq T; X \# T. e \\
\forall X \leq T. U & := \forall X \leq T; X \# T. U
\end{align*}
\]

Examples:
1. \(\vdash \{x=3\} \{x: \text{Int}\} \text{ with } \{x: \text{Int}\} \{x=6\} : \{x: \text{Int}\} \land \{x: \text{Int}\}\)
   The \(x\)-field of the \(\{x=3\}\) record is updated with 6.
2. \(\vdash \{x: \text{Int}\} \land \{x: \text{Int}\} \sim \{x: \text{Int}\}\)
   \(\vdash \{x=3\} \{x: \text{Int}\} \text{ with } \{x: \text{Int}\} \{x=6\} : \{x: \text{Int}\}\)
   In most examples we will simplify typing as much as possible.
3. \(\{x=3\} \{x: \text{Int}\} \text{ with } \{x: \text{Int}\} \{x=6\}.x = 6\)
   The right value has preference above the left value.
4. \(\{x=3\} \{x: \text{Int}\} \text{ with } \emptyset \{x=6\}.x = 3\)
   The right record is coerced to the empty record, so the \(x\)-field has to be the left one.
   Note that the value of the expression \(\{x=3\} \{x: \text{Int}\} \text{ with } T\{x=6\}\) depends on the type \(T\).
5. \( I \{ x=3, y='a' \} \{ x: \text{Int}, y: \text{Char} \} \) with \( I \{ x=5 \} \{ x: \text{Int}, y: \text{Char} \} \)
Here a two-field record is updated.

6. \( I \{ x=3 \} \{ x: \text{Int}, y: \text{Char} \} \) with \( I \{ x=5, z=3.4 \} \{ x: \text{Int}, y: \text{Char}, z: \text{Real} \} \)

7. \( I \{ x=3, y='a' \} \{ x: \text{Int}, y: \text{Char} \} \) with \( I \{ x=6, y='b' \} \{ x: \text{Int}, y: \text{Char} \} \)

8. \( I \{ x=3, y='a', z=8.2 \} \{ x: \text{Int}, y: \text{Char}, z: \text{Real} \} \) with \( I \{ x=5, z=3.4 \} \{ x: \text{Int}, y: \text{Char}, z: \text{Real} \} \)

9. \( (\lambda p: I \{ x: \text{Int} \} . p) \{ x: \text{Int} \} \) with \( (\lambda p: I \{ x: \text{Int} \} . p) \{ x=p.x+1 \} \{ x=4, y='a' \} \)

The function throws all other fields away, as it is not polymorphic.

10. \( I \{ x=4, y='a' \} \{ x=p.x+1 \} \{ x=5, y='a' \} \)

This function performs a polymorphic record update: it increases the \( x \)-field of an arbitrary record by one. We need the demand \( P \leq \| x: \text{Int} \| \) to select the \( x \)-field, and \( P \# \| x: \text{Int} \| \) to change it. All other fields are preserved.

Using the abbreviation above we write
\( (\forall P \leq \| x: \text{Int} \| ; P \# \| x: \text{Int} \| . \lambda p: I \{ x: \text{Int} \} . p \rightarrow P \wedge \| x: \text{Int} \| ) \)

This extension of \# implies that the functions \( \text{ex}8 \) and \( \text{ex}9 \) of section 2.5, that were intended to extend a record with a field, also can be applied to records already having this field (with the same type). In such a case, this field will be overwritten by the function.

Discussion:
There seem to be two approaches of making record update possible. One way is to have a record concatenation operator that works for disjoint records, and a restriction operator, that deletes fields from a record. These systems are powerful, but also complex, since there are two operators that can interact in various ways. This complexity reveals itself especially in the subtyping rules. Furthermore, we don’t need this power, because there is no need for record restriction in its own right.

Another approach, the one we have chosen here, is a record concatenation that gives precedence to the right argument for common fields, so record update is a special case of record concatenation. This approach comes in different flavours. We choose for a concatenation that is permitted only if the types of the common fields are the same, because this gives simple subtyping rules (e.g. S-Inter-LB). The standard flavour in implicitly typed systems is the concatenation that is always permitted [Wan 87]. In an explicitly typed system, this would lead to more complex rules.

2.8 Specialized record updating
For certain uses, it is desirable to update not only with the same type, but also with a subtype. E.g. a record with an \( x \)-field of type Real can be updated by an Int, because an integer can be converted to a real. This demands a generalisation of the (\#-Equal) rule, so we have \( R \# S \) if on common fields \( S \) is a subtype of \( R \).

Rules:
\[
\frac{\Gamma \vdash U \leq T}{\Gamma \vdash \| U \| \# \| U \|} \quad \text{(\#-Equal)}
\]
(this rule makes the old rule (#-Equal) redundant)

Examples:
1. $\{x=5.4, y='a'\} \{x:Real, y:Char\} \text{ with } \{x:Int, y:Char\} \\
2. $\{x=5.4, y='a'\} \{x:Real, y:Char\} \text{ with } \{x:Int\} \{x=3\}. x = 3$
3. Again functions $ex8$ and $ex9$ of section 2.5 can be applied to a wider class of records.

Discussion:
The types in the premise cannot be swapped, because this would violate the typing rules or the 'preference to right' meaning of with. This can be clarified by giving an example. Suppose we have the rule

\[ \Gamma \vdash T \leq U \]
\[ \Gamma \vdash \{y:T\} \not\in \{z:U\} \quad (\#-?) \]

Then the following term would be permitted:

\[ \vdash \{x=3\} \{x:Int\} \text{ with } \{x:Real\} \{x=5.4\} : \{x:Int\} \land \{x:Real\} \]

According to the 'preference to right' rule, the x-field of this term is equal to 5.4 (a real). However, this is in contradiction with rule (S-Inter-LB), that says the term should be convertible to $\{x:Int\}$, which is clearly not the case anymore. So we can't admit rule (#-?). For the same reason a possible rule for symmetry of # cannot be admitted anymore.

Note: The two demands $X \leq \{y:Int\}$ and $X \not\in \{y:Int\}$ can now be summarized by "$X$ has an Int-field with type equivalent to Int.

2.9 Transitivity
Suppose we have two functions updating a different field of a record:

\[
\begin{align*}
\text{changeX} &= \lambda r. r \text{ with } \{x:Int\} \{x=r.x + 1\} \\
\text{changeY} &= \lambda r. r \text{ with } \{y:Int\} \{y=r.y - 1\}
\end{align*}
\]

We might want to define the function that performs both updates on a record that has both an x and a y-field like this:

\[
\text{changeXY} = \lambda r. \text{changeX} \text{ with } \{x:y, y:Int\} \{x=r.x + 1, y=r.y - 1\}
\]

In OOP, functions would be frequently defined in such a way. Unfortunately, this function is not permitted (typable) with the current rules, because we can't derive:

\[ R \not\in \{x:Int, y:Int\} \quad \text{with} \quad R \not\in \{x:Int\} \]

We remedy this situation by adding the rule:

\[\begin{align*}
\Gamma \vdash R \leq S \quad \Gamma \vdash S \leq T & \quad \Rightarrow \quad \Gamma \vdash R \not\in T \quad (\#-Trans) \\
\Gamma \vdash R \not\in S \quad \Gamma \vdash S \not\in T & \quad \Rightarrow \quad \Gamma \vdash R \not\in T
\end{align*}\]

Discussion:
To understand this we have to remember what it means that types $R$ and $S$ are compatible and in the subtype relationship: recordtype $R$ has all of the fields of $S$ with the same type. If the same holds for $S$ and $T$, then we clearly have that $R$ has all of the fields of $T$ with the same type, so $R$ must be compatible with $T$.

With this rule it is permitted to define function $\text{changeXY}$ as we did above. The rule (#-Trans) is used twice for typing it: once to derive $R \not\in \{x:Int\}$, and once to derive $R \not\in \{y:Int\}$.

Examples:
1. $R \leq \text{Top}; R \not\in \{x:Int, y:Int\} \vdash R \not\in \{x:Int\}$
2.10 Recursive record update

A special form of the update problem is the so-called deep or recursive update. This is an update that takes place "deep" in a nested record, e.g. the update of the m-field of the l-field of a record of type $\llbracket l::\text{Int} \rrbracket$. We don't need this update in our model of OOP, so this section is not important for that goal. With the operations defined it is possible to perform such a deep update, but only in a very clumsy way:

$$\lambda S \leq \# \llbracket m::\text{Int} \rrbracket. \lambda R \leq \# \llbracket l::S \rrbracket. r:R.$$

$$r R \text{ with } \llbracket l::S \rrbracket \{l=r.l S \text{ with } \llbracket m::\text{Int} \rrbracket \{m=1 + r.l.m\}\}$$

We will generalize our compatibility rules and add a subtyping rule to make such an update easier:

Rules:

$$\frac{\Gamma \vdash U \leq T}{\Gamma \vdash T \# U} \quad (\#\text{-Sub})$$

$$\frac{\Gamma \vdash T \# U}{\Gamma \vdash \llbracket l::T \rrbracket \# \llbracket l::U \rrbracket} \quad (\#\text{-Equal})$$

$$\frac{\Gamma \vdash \llbracket l::(T \land U) \rrbracket}{\Gamma \vdash \llbracket l::(T \land U) \rrbracket} \quad (\#\text{-Label-Distr})$$

(Rule (\#-Sub) covers the base case, where two types are (almost) the same, and permits the overwriting of the left value by the right value.

Already derivable in our system is

$$\llbracket l::(R \land S) \rrbracket \leq \llbracket l::R \rrbracket \land \llbracket l::S \rrbracket$$

so together with (\#\text{-Label-Distr}) we have

$$\llbracket l::(R \land S) \rrbracket \sim \llbracket l::R \rrbracket \land \llbracket l::S \rrbracket$$

We need this equivalence for instance when typing the equation above or the selection of a field of a concatenated record (example 5 below).

Discussion:

The equality rules given in section 2.5 do not suffice any longer; the equality can depend on subtyping judgments. Only a full equational calculus can help here. All examples of the previous sections will remain valid, so the intuition gained there can be used here. The concatenation we have here is called recursive, because we have for compatible record types $R$ and $S$:

$$\{l=r\} \llbracket l::R \rrbracket \text{ with } \llbracket l::S \rrbracket \{l=s\} = \{l=(r R \text{ with } S s)\}$$

This is to say: Suppose we have two records $a$ and $b$ that have a field $l$ in common that contains in both cases another record, $r$ and $s$ respectively. Then the contents of the $l$ field of $a$ and $b$ concatenated will be the concatenation of $r$ and $s$. This corresponds to rule (\#-Equal).

Rule (\#-Sub) covers the base case, where two types are (almost) the same, and permits the overwriting of the left value by the right value.

Examples:

1. $\vdash \{l=(m=4, n='a')\} \llbracket l::\text{Int}, n::\text{Char} \rrbracket \text{ with } \llbracket l::\text{Int} \rrbracket \{l=(m=5)\}$

This is a deep update.

2. $\vdash \{l=(n='a')\} \llbracket l::\text{Char} \rrbracket \text{ with } \llbracket l::\text{Int}, p::\text{Real} \rrbracket \{l=(m=4, p=3.14)\}$

A recursive concatenation.

3. $\vdash \llbracket l::\text{Char} \rrbracket \land \llbracket l::\text{Int}, p::\text{Real} \rrbracket \sim \llbracket l::\text{Int}, n::\text{Char}, p::\text{Real} \rrbracket$

using (\#\text{-Label-Distr})

4. $\vdash \llbracket l::\text{Char} \rrbracket \land \llbracket l::\text{Int}, p::\text{Real} \rrbracket \sim \llbracket l::\text{Int}, n::\text{Char}, p::\text{Real} \rrbracket$

5. $\vdash \text{ex3.1} : \llbracket l::\text{Int}, n::\text{Char}, p::\text{Real} \rrbracket$

This uses the previous equivalence
6. $\lambda R \leq \# \{ [\mathit{m}: \mathit{Int} ] \}$. $\lambda R. \ r \ R$ with $\{ [\mathit{m}: \mathit{Int} ] \} \ {l=\{m=r\.1.m+1\}}
\text{VR} \leq \# \{ [\mathit{m}: \mathit{Int} ] \}$. $R \leftarrow R$
this function increases the $m$-field of the $l$-field and preserves all other fields. This
example is adopted from [CM 91].

7. $\vdash 6 \{ [\mathit{m}: \mathit{Int}, n: \mathit{Char} ], p: \mathit{Real} \} \ {l=\{m=4, n='a' \}, p=3.14}:
\{ [\mathit{m}: \mathit{Int}, n: \mathit{Char} ], p: \mathit{Real} \}$

8. $\vdash 7 \mathit{Int} \ with \ \mathit{Int} 5 : \mathit{Int} \land \mathit{Int}$

9. $\vdash 7 \mathit{Int} \ with \ \mathit{Int} 5 = 5$

10. $\vdash 7 \mathit{Int} \ with \ \mathit{Int} 5 = 5$

still preference to right.

2.11 Conclusion

We have presented the syntax and derivation rules of $F\#$, by starting with $F$, extending the
set of rules in each section. Since many rules have become redundant, the final rules are
put together in Appendix A. We have shown that the system elegantly solves the problem of
record update, with a quite general concatenation operator.

There are variations of $F\#$, where not all rules presented in sections 2.7-2.10 are used, e.g.
the system with only the rules of sections 2.7 and 2.9 added to the basic rules is strong
enough to model OOP as we do in section 3. There are also variations that include a rule for
symmetry of $\#$ (we then have to dispose specialised record updating, as discussed there).

3 Object Oriented Programming

In this chapter we will show how certain concepts of object oriented programming (OOP) can
be modelled in $F\#$. First we will give a quick introduction to some ideas of OOP, and give
an example program. Then we will show briefly how these ideas can be modelled, some in
$F\#$, and some in extensions of $F\#$. From section 3.1 onwards we will show how the example
program will be translated to increasingly faithful and complex models.

The most important concepts of object-oriented programming are object, class, method and
inheritance. Every object has a state and a collection of methods that are transformations on
the state. Objects are grouped into classes. All objects in one class have the same methods
and the same type of state. So a class describes the methods and the type of the state, and
an object is defined by its class and the state. The state consists of a number of instance
variables, that have some value. Instance variables of an object are comparable to fields of a
record in more traditional languages.

A class may be defined independently, but also as an extension of an existing class. The
new class is called a subclass of the old class, the superclass. A subclass may have more
instance variables or more methods, and typically has both. The subclass only has to specify
what it has more than the superclass, the rest of both state and methods is inherited. Methods
can be inherited because objects of a subclass have all instance variables of objects of the
superclass, so methods that work for objects of a superclass will also work for objects of the
subclass. Sometimes a subclass redefines existing methods. A subclass may also have more
superclasses, this idea is called multiple inheritance. In this way a whole class-hierarchy can
be defined.

Another important concept is encapsulation: The state of an object can be reached only
through its methods. OOP has this feature in common with the practice of abstract datatypes.
The last major concept is that of self-reference: A method applied to an object may invoke
other methods that are defined on this object. This encourages separation of tasks. There
are different possibilities to determine which method is actually invoked, as we will see later
on.

We have in figure 1 a small object oriented program in a functional OO language, derived
from [PT 93]. Normally object-oriented languages are imperative, but a functional variant
suits our purposes better. We will clarify the concepts introduced above with this example.
class Point is
    vars x:Int=0
with
    getX = state@x,
    setX = fun i:Int. state@x = i,
    bump = self.setX state (1 + (self.getX state))
end

class 2DPoint from Point is
    vars y:Int=0
with
    getY = state@y,
    setY = fun i:Int. state@y = i,
    bump = let state' = super.bump state in
        self.setY state' (1 + (self.getY state))
end

class Colour is
    vars c:String="red"
with
    getC = state@c,
    setC = fun col:String. state@c = col
end

class CPoint from Point, Colour is
with
    setX = fun i:Int. let state' = super1.setX state i in
        self.setC state' "blue"
end

Figure 1: An example of object oriented programming
The first lines define the class `Point`. An object in this class will represent a point on the line, that can be moved or bumped. The second line declares that there is just one instance variable, `x`, with initial value 0.

The rest of the class describes the methods of `Point`. The first method delivers the x-component of the state of a point, a method does not need to deliver an object. The abstraction over the point is implicit here, in contrast to pure functional languages. However, we are explicit in which state has to be examined, in contrary to imperative OO languages. This is a consequence of the absence of side-effects. The second method has the intention of moving the point to another place. Here is abstracted from the place, and a new point is returned. It is a copy of the old state, but with the x-component changed to i. Again, in an imperative language we would just change an instance variable. The third method bumps a point. For this class it means that the x-coordinate is increased by one. It does so by invoking the other methods defined on points. This self-reference is denoted by prefixing the method's name by "self.", and is applied to a state.

Now a simple example of the use of this class can be given. We will assume that the program text of figure 1 is already read in by the interpreter. The expressions given to the interpreter are preceded by #; the output is given right under each expression.

```
# `getX (bump point 'new) 1 : Int

The only remarkable thing is the function point 'new, which is implicitly defined when the class `Point` was declared, and delivers a new `Point` object with initial state as specified in the class.

The class `2DPoint` will consists of movable points in the plane. We define it as a subclass of `Point`, so it has an instance variable `x` in the state, and all the methods of `Point`. It adds an instance variable `y`, and two new methods. The `bump` method is redefined, but it still makes use of the old version by calling `super. bump`. In this definition we also see why we have to specify on which state methods must act.

The `bump` method applied to a point in the plane will change both coordinates:

```
# `getY (bump 2dpoint 'new) 1 : Int

Still the methods of `Point` can be applied, without disturbing any other information:

```
# `getY (setX (setY 2dpoint 'new 7) 5) 7 : Int

Next we will demonstrate multiple inheritance. First we define another base class, `Colour` (a base class is a class without superclasses). The class `CPoint` has all the instance variables and methods of both `Point` and `Colour`, and adds nothing. It only redefines the `setX` method so that the colour is changed to blue every time a point is moved.

Now a question arises around the self-reference in the definition of `bump`. When this method is applied to a `CPoint`, will `self.setX` refer to the `setX` of `class Point` (and not change the colour), or to the `setX` of `CPoint` (and change the colour to blue)? In every object-oriented language this question appears. Most languages (e.g. Smalltalk) have chosen the last option as this is the most flexible. Our more advanced models will also model this so-called `late-binding`, and give the result:

```
# `getC (bump cpoint 'new) "blue" : String

We give one by one the concepts of OOP that we will model in the following sections, and for each concept the extension of the model to incorporate the new concept is given.

1. Objects and methods (without inheritance)

   An object is modelled by a record, with a field for every instance variable. So we represent an object by just the state. A method is a global function that returns a new state.

2. The inheritance of instance variables. As a consequence, methods of a superclass K can read and modify an object of a subclass L.
The record type B of objects of a subclass L is an extension of the record type A of objects of the superclass I. Consequently, we have B⊆A and B#A, so functions that can read and modify A's (these functions represent methods of the superclass I), can read and modify B's (i.e. objects of L).

3. The inheritance of methods. When not explicitly redefined, a method is copied from its superclass. A method can refer to the methods of the superclass (super-reference).

The functions that represent the methods are packed in one record. The functions of the superclass are copied to the subclass by the \texttt{with} construction. We will need type operators as in \texttt{F#}.

4. Encapsulation. An object can be reached only through its methods.

The state and the methods are packed together in an object and the abstract datatype mechanism prevents the state from direct access. Existential types are needed.


The set of methods gets as extra parameter \texttt{self}. When an instance of the class is created, the record of functions representing the own methods is substituted for \texttt{self}. To implement this, we need a fixpoint combinator.

In the following sections we will further explain the modelling of the concepts above with the translation of the example OO program to a programming language based on \texttt{F#}. The language is extended with transparent definitions (global and local) and some standard types. Furthermore, the notation is a little different: \texttt{fun} and all are written instead of \lambda and \texttt{V}. The conventions about the program text are the same as before.

In the first section, we give an extremely poor translation of the program, as we only model concept 1. In each section we will model one more concept of the list above, so the model of OOP in section 3.5 is fairly complete.

3.1 Objects and methods

In this section we will give a very poor model of OOP: Objects and methods are modelled, but inheritance is not. We represent an object by a record with one field for each instance variable. So an object is solely represented by its state, not its methods. The state of a Point is a record with an x-field containing an integer:

```
# PointR = {1x:Intl}
PointR : *
```

We have as convention that a state type ends with \texttt{R} (for representation type). The methods are translated as global functions:

```
# getX = fun s:PointR. s.x
getX : PointR -> Int
# setX = fun s:PointR. fun i:Int. (x=i)
setX : PointR -> Int -> PointR
# bump = fun s:PointR. setX s (1+ getX s)
bump : PointR -> PointR
```

The functions get an extra parameter representing the state. This parameter is implicit in figure 1. Note that we resolved the self-references of the example program by giving the definitions in a specific order. In general, this doesn't work (cross-dependencies), and we need the fixpoint combinator to resolve the mutual recursive definitions. To complete the class, we give the start object, so we can use our functions\(^1\).

```
# point'nev = {x=0}
point'nev : PointR
# p1 = bump point'nev
p1 : PointR
# getX p1
1
```

\(^1\)The ' has no special meaning; it is considered to be a letter.
We cannot extend the definitions we have given before to make a class of two dimensional points. We have to start all over again, and give for each method again a definition.

\[
\begin{align*}
\text{2DPointR} &= \{x: \text{Int}, y: \text{Int}\} \\
\text{2DPointR} : &\ast \\
\text{2dGetX} &= \text{fun } s:\text{2DPointR}. s.x \\
\text{2dGetX} : &\text{2DPointR} \rightarrow \text{Int} \\
\text{2dGetY} &= \text{fun } s:\text{2DPointR}. s.y \\
\text{2dGetY} : &\text{2DPointR} \rightarrow \text{Int} \\
\text{2dGetX} &= \text{fun } i:\text{Int}. \{x=i, y=s.y\} \\
\text{2dGetX} : &\text{2DPointR} \rightarrow \text{Int} \\
\text{2dGetY} &= \text{fun } i:\text{Int}. \{x=s.x, y=i\} \\
\text{2dGetY} : &\text{2DPointR} \rightarrow \text{Int} \\
\text{2dbump} &= \text{fun } s:\text{2DPointR}. \text{let } s' = 2\text{dsetX } s (1+ 2\text{dgetX } s) \text{ in} \\
\text{2dbump} : &\text{2DPointR} \\
\text{2dpoint'new} &= \{x=0, y=0\} \\
\text{2dpoint'new} : &\text{2DPointR} \\
\text{Functions that work on points can be used for 2dpoints, but that results in loss of information.} \\
\text{p2} &= \text{getY} (\text{bump 2dpoint'new}) \\
\text{Type error} \\
\text{2dGetY} (\text{2dbump 2dpoint'new}) \\
\end{align*}
\]

Since inheritance is not yet implemented, it isn't particularly interesting to give the translation of the classes Colour and CPoint here, so we will not do so. For this model only the simply typed calculus extended with records is needed.

### 3.2 Inheritance of instance variables

In this section, we will give a model implementing the most crucial aspect of OOP: An object of a subclass may be used in any place where an object of the superclass is used.

This is possible because instance variables are inherited, i.e. a subclass always has all the instance variables the superclass has. A subclass doesn't have to name all instance variables it inherits, but name only the variables it adds. We can model this in F# by record concatenation.

\[
\begin{align*}
\text{PointR} &= \{x: \text{Int}\} \\
\text{PointR} : &\ast \\
\text{2DPointR} &= \text{PointR} \land \{y: \text{Int}\} \\
\text{2DPointR} : &\ast \\
\end{align*}
\]

Now we are justified in demanding that a function that works on Points, also works on 2DPoints, since “everything you do with a point can also be done with a 2dpoint”. The functions given in section 3.1 do not satisfy this demand. We have to make them polymorphic. We might try to make a polymorphic \text{getX} function by

\[
\begin{align*}
\text{try-getX} &= \text{fun } s:\text{FinalR}. s.x \\
\text{try-getX} : &\text{all } FinalR ::\ast. \text{FinalR} \rightarrow \text{Int} \\
\end{align*}
\]

Obviously, this doesn't work because it isn't ensured that \(s\) has an \(x\)-field. By allowing only types \(\text{FinalR}\) that are a subtype of \(\text{PointR}\) we can type the body of the function (in programs we write \(\le\)):

\[
\begin{align*}
\text{try-getX} &= \text{fun } \text{FinalR} :\text{PointR}. \text{fun } s:\text{FinalR}. s.x \\
\text{try-getX} : &\text{all } \text{FinalR} :\text{PointR}. \text{FinalR} :\text{Int} \\
\end{align*}
\]

We have \(\text{PointR} :\text{PointR}\), so \(\text{try-getX}\) accepts points, but also \(\text{2DPointR} :\text{PointR}\), so it accepts 2dpoints as well. Now we move to the setX method.

\[
\begin{align*}
\text{try-setX} &= \text{fun } \text{FinalR} :\text{PointR}. \text{fun } i:\text{Int}. \{x=i\} \\
\text{try-setX} : &\text{all } \text{FinalR} :\text{PointR}. \text{FinalR} :\text{Int} \\
\end{align*}
\]

21
This is not the desired typing, since it returns a PointR instead of a FinalR. We now have the update problem (see section 2.7). This is solved with the with-construction, yielding

```
# try2-setX = fun FinalR<#PointR. fun s:FinalR. fun i:Int.
  s FinalR with \{Ix:Int\} \{x=i\}
```

```
try2-setX : all FinalR<#PointR. FinalR->Int->FinalR
```

Here we demand that the state type of the object is compatible with PointR, so that it is permitted to update the object with a value of type PointR. We have PointR<#PointR, so try2-setX can be applied to points, and 2DPointR<#PointR, so the function can be applied to 2Dpoints as well.

Thus we demand FinalR<#PointR to ensure we can read the object as if it were a point, and FinalR<#PointR to ensure we can update the object as if it were a point (update in the functional sense of course).

Now we give the uniform representation of all methods.

```
# getX = fun FinalR<#PointR. fun s:FinalR. s.x
```

```
getX : all FinalR<#PointR. FinalR->Int
```

```
# setX = fun FinalR<#PointR. fun s:FinalR. fun i:Int.
  s FinalR with \{Ix:Int\} \{x=i\}
```

```
setX : all FinalR<#PointR. FinalR->Int->FinalR
```

```
# bump = fun FinalR<#PointR. fun s:FinalR.
  setX FinalR s (1+(getX FinalR s))
```

```
bump : all FinalR<#PointR. FinalR->FinalR
```

To complete the class, we give start objects, so we can use our functions. Again, we use the record concatenation, this time to inherit the initial value of an object.

```
# point'new = \{x=0\}
```

```
point'new : PointR
```

```
# 2dpoint'new = point'new PointR with \{y:Int\} \{y=0\}
```

```
2dpoint'new : 2DPointR
```

```
# p1 = bump PointR point'new
```

```
p1 : PointR
```

```
# getX PointR p1
```

```
1
```

```
# p2 = bump 2DPointR 2dpoint'new
```

```
p2 : 2DPointR
```

Now we have successfully modelled inheritance of instance variables and the use of methods of a superclass for objects of a subclass. To complete the class 2DPoint, we give its methods.

```
# getY = fun FinalR<#2DPointR. fun s:FinalR. s.y
```

```
getY : all FinalR<#2DPointR. FinalR->Int
```

```
# setY = fun FinalR<#2DPointR. fun s:FinalR. fun i:Int.
  s FinalR with \{y:Int\} \{y=i\}
```

```
setY : all FinalR<#2DPointR. FinalR->Int->FinalR
```

```
# bump2 = fun FinalR<#2DPointR. fun s:FinalR.
  let s' = bump FinalR s in
  setY FinalR s' (1+(getY FinalR s))
```

```
bump2 : all FinalR<#2DPointR. FinalR->FinalR
```

To type the body of the function `getY`, `FinalR<#\{y:Int\}` has to be derived. This is done by using transitivity of subtyping and `2DPointR<#\{y:Int\}`. In the same way we use the transitivity law for `#`, to type the `setY` function. In typing `bump2`, both laws are used.

We have to give the `bump` method for two-dimensional points another name than the `bump` method for one-dimensional points, since the methods are global functions, and not

\*\*\*We have an "intelligent" interpreter, that always gives the shortest minimal type.\*\*\*
packed with the instance variables in the object. This is clearly a disadvantage of this simple model of objects and classes.

Some examples that illustrate the difference between the various bump functions:

```fsharp
# getY 2DPointR (bump 2DPointR 2dpoint'new)
0 : Int
# getY 2DPointR (bump 2DPointR 2dpoint'new)
1 : Int
```

Now we will show how multiple inheritance is modelled. First we define the other base class: Colour.

```fsharp
# ColourR = {lc:String!}
ColourR : *
# getC = fun FinalR<#ColourR. fun s:FinalR. s.c
getC : all FinalR<#ColourR. FinalR -> ColourR
#  s FinalR with {lc:String!} {c=col}
setC : all FinalR<#ColourR. FinalR -> ColourR -> FinalR
# colour'new = {c="red"}
colour'new : ColourR
```

Now we combine the class Point and Colour in one class of coloured points, to which all methods of both classes are applicable.

```fsharp
# CPointR = PointR \ ColourR
CPointR : *
# cpoint'new = point'new PointR with ColourR colour'new
```

Observe that we have that CPointR is a subtype of and compatible with both PointR and ColourR, so the elements of CPointR can be used everywhere where an element of PointR or ColourR can be used.

We define a bump method for coloured points, that bumps it as a point, and sets the colour to blue:

```fsharp
# setX2 = fun FinalR<#CPointR. fun s:FinalR. fun i:Int.
#  setC FinalR (setX FinalR s i) "blue")
setX2 : all FinalR<#CPointR. FinalR -> FinalR
```

```fsharp
# p1 = setX2 CPointR cpoint'new 3
p1 : CPointR
# get x CPointR p1
3
# getC CPointR p1
"blue"
```

But the bump function always uses setX, and not setX2 as is desirable for coloured points:

```fsharp
# getC CPointR (bump CPointR cpoint'new)
"red"
```

In this section we have shown that it is possible to give a simple model of classes and inheritance in F#.

This model has some important shortcomings. The most important one is that methods are not packed in the objects, but are separate functions. This has as a consequence that every time a class redefines a method a new name has to be chosen, and programs have to be explicit about which version of a method is used. This can result in undesired behaviour, as we see with the bump method of the class 2DPoint.

### 3.3 Inheritance of methods

Inheritance of methods is the mechanism that defines all methods of a subclass that are not redefined, to be equal to the corresponding methods of the superclass. We model this by first packing all methods together in a record. Now the combination of the methods of the
superclass and the methods of the subclass, is done with the \emph{with} construction. An added advantage of packing the methods together is that later on, the state and the methods can easily be put together.

**Extension of language**

To implement this aspect of inheritance, type operators are very convenient, if not necessary. Type operators are functions from types to types, and are admitted in higher order type systems [Gir 72]. These operators are used here to indicate which methods a class has, i.e. to give the \emph{interface} of a class. So we assume we have a higher order type system instead of second-order. The resulting system is very close to Compagnoni's $P_\lambda$[Com 95], to which compatibility is added.

The subtyping and compatibility rules are extended point-wise for type operators:

\[
\begin{align*}
\Gamma, X:* &\vdash S \leq T \\
\Gamma &\vdash \lambda X:* \cdot S \leq \lambda X:* \cdot T \quad (S-Oper) \\
\Gamma &\vdash S \leq T \quad \Gamma &\vdash S U:* \\
\Gamma &\vdash S U \leq T U \quad (S-App) \\
\Gamma, X:* &\vdash S \# T \\
\Gamma &\vdash \lambda X:* \cdot S \# \lambda X:* \cdot T \quad (#-Oper) \\
\Gamma &\vdash S \# T \quad \Gamma &\vdash S U:* \\
\Gamma &\vdash S U \# T U \quad (#-App)
\end{align*}
\]

That is, a type operator $S:* \to T$ is a subtype of type operator $T'$ iff for all types $U$ we have $S U \leq T U$, and likewise for compatibility of type operators. The intersection of type operators is also defined pointwise:

\[
(\lambda X:* \cdot S) \land (\lambda X:* \cdot T) := (\lambda X:* \cdot S \land T)
\]

**The model**

We put all methods in a record, where the field name corresponds to the name of the method. As all functions are polymorphic, we push the type abstraction outside the record.

# PointR = {lxlnt1} pointR : * # pointclass = 
# fun FinalR<#PointR. 
# {getX = fun s:FinalR. s.x, 
# setX = fun s:FinalR. fun i:Int. s FinalR with ({lxlnt1} (x=i)), 
# bump = fun s:FinalR. s FinalR with ({lxlnt1} (x=s.x+1))} pointclass : all FinalR<#PointR. {getX: FinalR->Int, 
setX: FinalR->Int->FinalR, 
bump: FinalR->FinalR}}

We have the naming convention that the collection of methods of class $C$ is called $C$class. Remarkable is the change in the body of \texttt{bump}: The previous definition used functions \texttt{getX} and \texttt{setX}, but since all functions are put in one record, we can't do this anymore. Therefore the definitions of these two methods have to be substituted, which is bad programming practice. This will be recovered in section 3.5 by introducing self-reference.

This set of methods is used in the following way:

# point'new = {x=0} point'new : PointR # p1 = (pointclass PointR).setX point'new 5 p1 : PointR # (pointclass PointR).getX p1 5 : Int

We can still get direct access to the object:
We can give pointclass a much shorter type if we define the interface of Points, that is, the set of methods abstracted from the representation.

```haskell
```

The typechecker responds with saying that PointM is a function from types to types. We use the name CM for the interface of class C. Now we have:

```haskell
# check pointclass : all FinalR<#PointR. PointM FinalR
```

This construction with an interface is useful below.

Abstracting from the interface and the representation, we get a general class constructor:

```haskell
# ClasB = fun K:*->*. fun R:*. all FinaIR<#R. K R
```

```haskell
# check pointclasB : Class PointM PointR
```

We start the treatment of 2DPoint by giving the interface for the subclass:

```haskell
# 2DPointM = PointM /
```

The type operator 2DPointM is the extension of PointM with another type operator. We have that 2DPointM is equivalent to:

```haskell
fun Rep:*.{lgetX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep,
```

Now the methods of class 2DPoint are defined in terms of the methods of its superclass: some methods use in their body the methods of the superclass, whilst methods that are not named are copied from pointclass.

```haskell
# 2DPointR = PointR /
#   {ly:Int!}
```

```haskell
# 2DPointR : *
```

```haskell
# 2dpointclass =
#   fun FinalR<#2DPointR.,
#     super = pointclass FinalR in
#     super (PointM FinalR) with
#     {lgetY: FinalR->Int, setY: FinalR->Int->FinalR, bump: FinalR->FinalR!}
#     {getY = fun s:FinalR. s.y,
#      setY = fun s:FinalR. fun i:Int. s FinalR with {ly:Int!} {x=i},
#      bump = fun s:FinalR. let s' = super.bump s in
#        s' FinalR with {ly:FinalR} {y=s'.y+i}}
```

2dpointclass : Class 2DPointM 2DPointR

The record concatenation operator with is used to extend the record with the fields getY and setY, and to overwrite the field bump. Here the full power of with is used, whereas in the bodies of methods it is only used to perform an update. Note that we need the type operator PointM to type super in this concatenation.

We now have only one name for the bump method, but we have to select the right class:

```haskell
# 2dpoint'new = point'new PointR with {ly:Int!} {y=0}
```

```haskell
2dpoint'new : 2DPointR
```

```haskell
# p1 = (2dpointclass 2DPointR).bump 2dpoint'new
p1 : 2DPointR
```

```haskell
# (2dpointclass 2DPointR).getY p1
```

```haskell
1
```

```haskell
# p2 = (pointclass 2DPointR).bump 2dpoint'new
```

```haskell
p2 : 2DPointR
```

```haskell
# (2dpointclass 2DPointR).getY p2
```

```haskell
0
```

---

3This is an example of a type operator that accepts a type operator as argument

---

25
For reasons of brevity, we will not give the code for multiple inheritance.

In this section we have shown our system is powerful enough to implement the concept inheritance of methods, if it is extended with type operators. We model this form of inheritance by interpreting the set of methods by a record and inheritance by record concatenation. So we use with here in two different situations: In methods to provide a polymorphic record update, and in classes, to provide inheritance of methods.

### 3.4 Encapsulation

The feature added to the model of the previous section is encapsulation. Encapsulation is the idea that an object can only be accessed through its methods. To implement this, we need to pack the state and methods together and apply a mechanism that hides the state.

In the literature two mechanisms for encapsulation have appeared: procedural abstraction (using recursive records, see e.g. [Car 88]) and type abstraction. Following [PT 93], we choose for type abstraction, which is formalized in the context of lambda-calculi by existential types [MP 88]. The rules for forming, introducing and eliminating an existential type are

\[
\frac{\Gamma, X : * \vdash U : *}{\Gamma \vdash \exists X : *. U : *} \quad \text{(K-Exists)}
\]

\[
\frac{\Gamma \vdash T : * \quad \Gamma \vdash e : U[X := T]}{\Gamma \vdash \text{pack } T \text{ and } e \text{ to } \exists X : *. U : \exists X : *. U} \quad \text{(T-Exists-Intro)}
\]

\[
\frac{\Gamma \vdash e_1 : \exists X : *. T \quad \Gamma, X : * \vdash x : T \quad \Gamma \vdash e_2 : U}{\Gamma \vdash \text{open } e_1 \text{ as } X \text{ and } x \text{ in } e_2 : U} \quad \text{(T-Exists-Elim)}
\]

where we have written $\exists$ for existential types. It is possible to define existential types in terms of universal (or polymorphic) types, so this is a very harmless extension of the system.

With the interface constructor, we can now pack state and methods together in a record, and encapsulate this to form the type of point objects (writing some for $\exists$):

```ocaml
PointM : *->*
```

```ocaml
# Point = some Rep : *. {lstate : Rep, methods : PointM Rep /
Point : *
```

Abstracting from the interface, we get the following higher order type operator:

```ocaml
Object : (*->*)->*
```

When supplied with some interface, Object yields the types of objects with that interface. Now the type Point can be expressed more concisely as:

```ocaml
# check Point = Object PointM
Yes
```

We are now in a position to define the class Point in this model. The interface and object-type are already given, so we will go to the methods. Because of the encapsulation, the implementation of an application of a method can be quite verbose. Therefore we introduce for every method a function that applies this method to an object. This function has to remove the encapsulation, select the right method, apply that to the state, and encapsulate the result with the unchanged methods. A naive implementation of the invocation of the bump method might be:

```ocaml
# try-bump = fun p : Point. open p as R and r in
   pack R and {state=r.methods.bump r.state, methods=r.methods}
to Point
try-bump : Point -> Point
```

This is perfectly all right for points, but doesn't take subtyping into account: try-bump applied to a value of a subtype of Point always returns a Point, effectively throwing away the rest of the state and the methods. The correct solution for this problem is again using bounded quantification (method m of class A is denoted by $A^\forall m$).
Point'bump = fun M<PointM. fun p:Object M. open p as R and r in pack R and {state=r.methods.bump r.state, methods=r.methods} to Object M

Point'bump : all M<PointM. Object M -> Object M

The higher order bounded quantification guarantees that the object has an interface at least as refined as PointM, so the bump method is present in an object with that interface. The code for the invocation of the setX method is similar, getX doesn't return an object and is even simpler.

point'setX = fun M<PointM. fun p:Object M. fun i:Int. open p as R and r in pack R and {state=r.methods.setX r.state i, methods=r.methods} to Object M

point'setX : all M<PointM. Object M -> Int -> Object M

point'getX = fun M<PointM. fun p:Object M. open p as R and r in r.methods.getX r.state

point'getX : all M<PointM. Object M -> Int

Noteworthy is that we don't have to demand M#PointM as we did for the state of an object, where we had FinalR#PointR; FinalR#PointR. This means we can actually have method specialisation, e.g. a subclass has a method that delivers an Int where the same method of the superclass delivers a Real. We cannot have "instance variable specialisation", i.e. a subclass has an instance variable with as type a proper subtype of the type of the same variable in the superclass, which would be unsound. Here we see a big difference between the state of an object and the methods of an object, in contrary to models of OOP with recursive records, where there is not such a big distinction between both concepts.

Up to now, we haven't said anything about the actual representation of a point, but only which interface a point has. Let's turn towards one possible representation of points:

PointR = {lxlnt}
PointR :
pointrep'new : {x=0}

There is no change in the definitions of Class and pointclass, but the code for making a new object is new. The initial value and the methods are wrapped up to an existential type.

Class = "The same as in the previous section"
Class : (...->*)->*->*
pointclass = "The same as in the previous section"
pointclass : Class PointM PointR

# point'new = pack PointR and
# {state = pointrep'new,
# methods = pointclass PointR}
# to Point

point'new : Point

We can make an abstracted version of new:

# new = fun M:*->*. fun R:*. fun cl:Class M R. fun init:R.
# pack R and
# {state = init,
# methods = cl R}
# to Object M

new : all M:*->*. all R:*. Class M R -> R -> Object M

new applied to an interface, a representation type, the set of methods and an initial value delivers an object.

Now it is easy to use this machinery, and not possible to abuse objects:

Point'getX PointM (Point'bump PointM point'new)
1 : Int
point'new.x

Type error
The introduction of the subclass of two-dimensional points is straightforward:

```plaintext
# 2DPointM = PointM /\ fun Rep:*. {l getY: Rep->Int, setY: Rep->Int->Rep}
2DPointM : *->*
# 2DPoint = Object 2DPointM
2DPoint : *
# 2dpoint'getY = fun M<2DPointM. fun p:Object M. open p as R and r in
  r.methods.getY r.state
2dpoint'getY : all M<2DPointM. Object M -> Int
# 2dpoint'setY = fun M<2DPointM. fun i:Int. open p as R and r in
  pack R and {state=r.methods.setY r.state i, methods=r.methods}
  to Object M
2dpoint'setY : all M<2DPointM. Object M -> Int -> Object M
```

Observe that we have 2DPointM<PointM, so indeed we can use the method invocation mechanisms of Point, like point'setX. For this reason, we didn't define 2dpoint'setX, because it would be exactly the same as point'setX (apart from demanding an interface more refined than 2DPointM instead of PointM).

Again, the representation of a 2dpoint is the same as section 3.3.

```plaintext
# 2DPointR = PointR /\ {l y: Int}
2DPointR : *
# 2dpointrep'new = pointrep'new PointR with {l y: Int} {y=0}
2dpointrep'new : 2DPointR
# 2dpointclass = "The same as in the previous section"
2dpointclass: Class 2DPointM 2DPointR
# 2dpoint'new = new 2DPointM 2DPointR 2dpointclass 2dpointrep'new
2dpoint'new : 2DPoint
```

The bump method which is packed in a 2DPoint is used, not the one given in pointclass.

In some OO languages, there is an even stricter sense of encapsulation. Methods of subclasses may not touch the instance variables of superclasses. This can also be modelled in our calculus, but falls outside the scope of this article. See [PT 93].

### 3.5 Self-reference

In this section we will add self-reference to our model of OOP, so it will be fairly complete. This model is based on [PT 93, CP 93], but it is simpler because of the extended underlying calculus.

Self-reference is the idea that a method can use the own methods of an object. In particular, a method can use itself, i.e. be recursive. To model this, we certainly need the fixpoint combinator \( Y: \forall T: \ast. (T \rightarrow T) \rightarrow T \).

After some definitions are reintroduced, we present a simple way to resolve the self-references.

```plaintext
# Object, PointM, Point, point'getX, point'setX, point'bump,
# PointR, pointrep'new = "The same as in the previous section"
Object : (\ast\ast)\ast
PointM : \ast\ast
Point : *
point'getX : all M<PointM. Object M -> Int
point'setX : all M<PointM. Object M -> Int -> Object M
point'bump : all M<PointM. Object M -> Object M
PointR : *
pointrep'new : PointR
```

```plaintext
# try-pointclass =
# fun FinalR#PointR.
# Y (PointM FinalR)
```
(fun self: PointM FinalR.
  {getX = ...
  setX = ...
  \[bump\] = fun s:FinalR. self.setX (plus 1 (self.getX s)))

try-pointclass : all FinalR#PointR. PointM FinalR

Here self refers to the definitions given in the same record; self-references are immediately
short-circuited. This is exactly what happens in some OO languages. However, this is con­
sidered to be rather rigid. Maybe subclasses of Point have another, more appropriate way
of setting the $x$-coordinate, so we want self to refer to the final set of methods. This idea,
called late binding, is present in SmallTalk, for example. Late binding means we cannot fix
the references inside the definition of the class; the parameter self is left open.

# pointclass =
#  fun FinalR#PointR.
#   fun self: PointM FinalR.
#    {getX = fun s:FinalR. s.x,
#     setX = fun s:FinalR. fun i:Int. s FinalR with \{x:Int\} \{x=i\},
#     \[bump\] = fun s:FinalR. self.setX (plus 1 (self.getX s)))
pointclass : all FinalR#PointR. PointM FinalR -> PointM FinalR

Class = fun M:*->*. fun R:*. all FinalR#PointR R -> M R

# check pointclass : Class PointM PointR

Yes

When we make a new object, we may fill in the final set of methods for self, by "short­
circuiting" the class by $Y$.

# new = fun M:*->*. fun R:.*
#  fun cl:Class M R. fun init:R.
#   pack R and
#    \{state = init,
#    methods = Y (M R) (cl R)\}
#  to Object M

new : all M:*->*. all R:*. Class M R -> R -> Object M

For objects of base class pointclass there is no difference in behaviour between late
binding and not. So we introduce the subclass 2dpointclass. Most definitions stay the same:

# 2DPointM, 2DPoint, 2dpoint'getY, 2dpoint'setY, 2DPointR, 2dpoint'repnew
# = "The same as in the previous section"
2DPointM : *->* 2DPoint : *
2dpoint'getY : all M<2DPointM. Object M -> Int
2dpoint'setY : all M<2DPointM. Object M -> Int -> Object M
2DPointR : *
2dpoint'repnew : 2DPointR

Only the class definition changes:

# 2DPointR = PointR \{y:Int\}
# 2dpointclass =
#   fun FinalR#2DPointR.
#   fun self: 2DPointM FinalR.
#   let super = pointclass FinalR self in
#    super (PointM FinalR) with
#     \{\[getX\]: FinalR->Int, \[setX\]: FinalR->Int->FinalR, \[bump\]: FinalR->FinalR\}
#     \[getY\] = fun s:FinalR. s.y,
#     \[setY\] = fun s:FinalR. fun i:Int. s FinalR with \{y:Int\} \{y=i\},
#     \[bump\] = fun s:FinalR. fun i:Int. let s' = super.bump s in
We define super as pointclass applied to the final set of methods, self. This is allowed, because 2DPointM FinalR < PointM FinalR. It means that self in the superclass refers to the same methods as self in this subclass.

The circle of references is closed in the definition of a new 2DPoint:

```haskell
# 2dpoint'new = new 2DPointM 2DPointR 2dpointclass 2dpointrep'new
2dpoint'new : 2DPoint

# point'getY 2DPointM (point'bump 2DPointM 2dpoint'new)

1
```

Unfortunately our example class 2DPoint cannot illustrate late binding. However the class CPoint can, so we will first show how multiple inheritance can be modelled. The base class Colour will be defined without any comments:

```haskell
# ColourM = fun Rep:* {getC : Rep->String, setC : Rep->String->Rep} ColourM : *=>*
# Colour = Object ColourM
Colour : *
# colour'getC = fun M<ColourM. fun p:Object M. open p as R and r in r.methods.getC r.state
colour'getC : all M<ColourM. Object M -> String
# colour'setC = fun M<ColourM. fun p:Object M. fun c:String. open p as R and r in pack R and (state=r.methods.setC r.state c, methods=r.methods) to Object M
colour'setC : all M<ColourM. Object M -> String -> Object M

# ColourR = {lc:String}
ColourR : *
# colourrep'new = "red"
colourrep'new : ColourR
# colourclass =
#   fun FinalR<#ColourR.
#   fun self: ColourM FinalR.
#   {getC = fun s:FinalR. s.c,
#    setC = fun s:FinalR. fun col:String. s FinalR with {lc:col}}
colourclass: Class ColourM ColourR
# colour'new = new ColourM ColourR colourclass colourrep'new
colour'new : Colour
```

Now we will combine the interfaces of both superclasses and the new (empty) part of the interface. Here the power of a full concatenation operation is needed, the operation of adding one field is not enough here.

```haskell
# CPointM = PointM \ ColourM
CPointM : *=>*
# CPoint = Object CPointM
CPoint : *
```

Observe that we have both CPointM<PointM and CPointM<ColourM, so the method invocation mechanisms of Point and Colour can be used.

The definition of cpointclass in terms of pointclass and colourclass also relies heavily on a full concatenation operator, on the level of values, and the level of types. Since we need to concatenate three record values a couple of times, we introduce a special operation.

```haskell
# with3 = fun C:*. fun B&C. fun A#(B\C).
    fun a:A. fun b:B. fun c:C. a A with (B\C) (b B with C c)
with3 : all C:*. all B&C. all A#(B\C). A -> B -> C -> A/(B\C)
# CPointR = PointR \ ColourR
CPointR : *
# cpointrep'new = pointrep'new PointR with ColourR colourrep'new
```
cpointrep'new : CPointR
# cpointclass =
#  fun FinalR#CPointR.
#  fun self: CPointM FinalR.
#  let super1 = pointClass FinalR self in
#  let super2 = colourClass FinalR self in
#  with3 {lsetX: FinalR->Int->FinalR} (ColourM FinalR) (PointM FinalR)
#    super1
#    super2
#    {setX = fun s:FinalR. fun i:Int. let s' = super1.setX s i in
#      self.setC s' "blue")
cpointclass: Class CPointM CPointR
Note that in the case where super1 and super2 have some fields in common, the methods of
the latter would prevail.

The code for making a new cpoint and applying some methods to it is standard:
# CPoint'new = new CPointM CPointR cpointclass cpointrep'new
CPoint'new : CPoint

# colour'getC CPointM (point 'bump CPointM cpoint'new)
"Blue"

At last, we see the effects of late binding. bump makes use of self.setX, and setX is redefined
in the cpointclass to change the colour to blue. The late binding mechanism means the new
setX is used, so a bump on a coloured pointed changes the colour to blue.

3.6 Conclusion

The final model is based on Pierce & Turner [PT 93]. A class is just an object with enough of
the packaging left off, and enough decisions about representation and recursive self-reference
postponed, that it can still be extended. When the class is instantiated to form an object, the
representation and references to self are fixed and the methods all become concrete functions.

However, the encoding in \( F\# \) is much simpler and more natural, than in \( F_\leq \), since it
allows polymorphic record updates and concatenation of records of methods, both using the
\( \textit{with} \) -construction. Still it is desirable to design high-level syntax for OO programs, since
the translated code can be rather verbose. The translation from high-level syntax to \( F\# \) is
relatively straightforward, because of the powerful new primitives.

Let us give a short inventarisation what we used of our calculus in the model of 3.5. The
concatenation \( \textit{with} \) is used in three places:
1. For the polymorphic record update in the body of methods; the concatenation is used to
   overwrite some fields.
2. For the inheritance of methods. Here the full power of \( \textit{with} \) is used to overwrite old
   methods and add some new ones.
3. For the inheritance of the initial state. Again the full power of \( \textit{with} \) is used. (Overwriting
   occurs in the case of multiple inheritance, where the superclasses have a common
   instance variable.)

We used bounded quantification in two places:
1. For the polymorphic record update, where we demand that the final representation type
   is \( \leq \) and \# to the current representation, so subtyping on fields of the state is not
   possible.
2. For making polymorphic method calls, to ensure an object has the appropriate functions
defined. Here we need only the \( \leq \) demand.

Of the compatibility laws we used overwriting (section 2.7) and transitivity (2.9), but not
specialized record updating (2.8) or recursive concatenation (2.10). These are needed for
method specialisation (page 27) and for instance variables that are extendible records them-


deselves respectively. There are concepts of OOP that we didn't model, such as private instance
variables, binary methods and template classes. Because of the simplicity of our model we
expect that these concepts can also be implemented without major changes in the system.

31
4 Translation semantics

We adopt the so-called Penn interpretation of F#, which views subtyping as implicit coercion. This leads to a translation of F#, where the coercions are implicit, to Fx, with explicit coercions. Fx is the second-order lambda calculus with cartesian products and labelled types. This translation was given first for FG by Breazu-Tannen, Coquand, Gunter, Scedrov [BCGS 91], and for F, by Pierce [Pie 91h]. Our translation is an extension of the latter.

Such a translation is sometimes called a semantics because any semantics for the target system can be composed with this translation to yield a translation for the source system.

The idea is that well-typed terms are translated by induction on the typing derivation. So for each typing rule, we have to state how terms typed with that rule should be translated, given the translation of the terms occurring in the premiss of the typing rule. Effectively, we have to give a translation for each typing rule.

Since there are typically several typing derivations possible for one term, we will also get several translations. We have to show that all derivations result in essentially the same translation. This demand is called coherence and will be formulated more formally in section 4.4. This will place a very important restraint on the translation, i.e. we cannot take just anything for a coercion.

4.1 Introduction

In a system with subtyping each occurrence of the subsumption rule, say

\[ \Gamma \vdash e : S \quad \Gamma \vdash S \subseteq T \quad \text{(T-Subsumption)} \]

introduces a coercion from S to T that is applied to e, so each subtyping judgment \( S \subseteq T \) delivers a coercion function \( c \) of type \( S \rightarrow T \), so the rule above will be translated as

\[ \Gamma \vdash e : S \quad \Gamma \vdash c : S \rightarrow T \quad \text{(Translation)} \]

Just as terms are translated by induction on the typing derivation, coercions are defined by induction on the subtyping derivation. Thus each subtyping rule determines a coercion, typically based on coercions in the premises of the rule.

To what type should an intersection \( S \cap T \) be translated, in order to have coercions to \( S \) and \( T \)? There has to be a simple function from the translation of \( S \cap T \) to S, and one to T. Clearly, the cartesian product \( S \times T \) satisfies this requirement: The coercions are just the projections (here we follow [Pie 91h]). So the type \( \{[l:\text{Int}, m:\text{Char}]\} \), which is an abbreviation for \( \{[l:\text{Int}]\} \wedge \{[m:\text{Char}]\} \), will be translated as \( \{[l:\text{Int}]\} \times \{[m:\text{Char}]\} \). Suppose e is an \( F_\# \)-term of this type. To type e.l, e has to be coerced to \( \{[l:\text{Int}]\} \) (remember the (T-Sel) rule). So the translation of e.l will be \( (\pi_1 e) \).

To what type should an intersection \( S \cap T \) be translated, in order to have coercions to \( S \) and \( T \)? There has to be a simple function from the translation of \( S \cap T \) to S, and one to T. Clearly, the cartesian product \( S \times T \) satisfies this requirement: The coercions are just the projections (here we follow [Pie 91h]). So the type \( \{[l:\text{Int}, m:\text{Char}]\} \), which is an abbreviation for \( \{[l:\text{Int}]\} \wedge \{[m:\text{Char}]\} \), will be translated as \( \{[l:\text{Int}]\} \times \{[m:\text{Char}]\} \). Suppose e is an \( F_\# \)-term of this type. To type e.l, e has to be coerced to \( \{[l:\text{Int}]\} \) (remember the (T-Sel) rule). So the translation of e.l will be \( (\pi_1 e) \).

Now consider the term \( \{l=7, m='a'\} \{[l:\text{Int}, m:\text{Char}]\} \) with \( \{[l:\text{Int}]\} \) \( l=5 \). The type of this term is \( \{[l:\text{Int}]\} \wedge \{[m:\text{Char}]\} \). Again, when the l-field of this term is selected, the term has to be coerced to \( \{[l:\text{Int}]\} \). But this time it can be done in two different ways: via \( \{[l:\text{Int}]\} \wedge \{[m:\text{Char}]\} \) or directly. Those two ways give rise to different coercions:

\[ \lambda x : \{[l:\text{Int}]\} \times \{[m:\text{Char}]\} \times \{[l:\text{Int}]\} \quad \pi_1 \quad \pi_2 \]

Of course, we want both coercions to give the same result when applied to the term. This is the coherence property.

In order to translate the value \( \{l=7, m='a'\} \) \{[l:\text{Int}, m:\text{Char}]\} with \( \{[l:\text{Int}]\} \) \( l=5 \), we cannot just pair both records and be done with it, but we have to make all common fields equal, to ensure coherence. We chose the right record to have preference above the left record, so all common fields have to be overwritten in the left record. We will call the function that makes the common fields equal the overwriter. Just as the coercions are generated by subtyping derivations, overwriters will be generated by compatibility derivations. A function that overwrites parts of a value of type \( A \) by a value of type \( B \), has type \( A \rightarrow B \rightarrow A \). Each occurrence of the and-introduction rule,

\[ \Gamma \vdash e : T \quad \Gamma \vdash f : U \quad \Gamma \vdash T \neq U \quad \text{(T-And)} \]
is translated with

$$\Gamma \vdash e : T \quad \Gamma \vdash f : U \quad \Gamma \vdash p : T \to U \to T \quad (\text{Translation})$$

where \( p \) is the overwriter, that will be defined by induction on the compatibility judgment.

First we will give a formal definition of the target calculus, then we will show how terms are translated.

### 4.2 The target calculus

Our target calculus is \( F \) extended with cartesian products, a one-element type and labelled types. Formally, the syntax and the derivation rules are those of section 2.1 extended by:

#### Rules:

**Types:**

\[ T ::= \quad T \times T \quad \text{cartesian product} \]

\[ One \quad \text{one-element type} \]

\[ \{ [], T \} \quad \text{labelled type} \]

**Terms:**

\[ e ::= \quad (e, e) \quad \text{pairing} \]

\[ \pi_1 e \quad \text{first projection} \]

\[ \pi_2 e \quad \text{second projection} \]

\[ \text{one} \quad \text{unique element} \]

\[ \{ l = e \} \quad \text{labelled value} \]

\[ e.l \quad \text{selection} \]

**Reduction rules:** \( \{ l = e \}, l \triangleright_\beta e \)

**Derivation rules:**

\[ \frac{\Gamma \vdash T : * \quad \Gamma \vdash U : *}{\Gamma \vdash T \times U : *} \quad \text{(K-Cart)} \]

\[ \frac{\Gamma \vdash T : *}{\Gamma \vdash \text{One} : *} \quad \text{(K-One)} \]

\[ \frac{\Gamma \vdash (e, f) : T \times U}{\Gamma \vdash \pi_1 e : T} \quad \text{(T-Cart)} \]

\[ \frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \pi_2 e : T_2} \quad \text{(T-Proj)} \]

\[ \frac{\Gamma \vdash \text{one} : \text{One}}{\Gamma \vdash \text{One} : *} \quad \text{(T-One)} \]

\[ \frac{\Gamma \vdash \{ [], T \} : *}{\Gamma \vdash \{ [], T \} : *} \quad \text{(K-Label)} \]

\[ \frac{\Gamma \vdash \{ l = e \} : \{ [], T \}}{\Gamma \vdash e.l : T} \quad \text{(T-Sel)} \]

#### Examples:

1. \( \vdash \{ [], \text{Int} \} \times \{ m : \text{Char} \} : * \)
   
   This will be the translation of the \( F_\# \) type \( \{ k : \text{Int}, m : \text{Char} \} \).

2. \( \vdash \{ \{ l = 7 \}, \{ m = 'a' \} \} : \{ [], \text{Int} \} \times \{ m : \text{Char} \} \)
   
   This term will be the translation of the \( F_\# \) term \( \{ l = 7, m = 'a' \} \).

3. \( \vdash (\pi_1 \{ \{ l = 7 \}, \{ m = 'a' \} \}).l : \text{Int} \)
   
   This term will be the translation of \( \{ l = 7, m = 'a' \}.l \).
4. \( \vdash \pi_1 (\tau', a') : \text{Int} \)

The labels in this calculus are silly: they are not necessary. Terms and types of this calculus can be converted directly to a calculus without labels, by deleting all label constructions (example 4 is the conversion of example 3). Then records with labels are translated to cartesian products, and field selection to projection, which is all very natural. Another nice effect is that this translation automatically delivers coercions (here: permutations) to convert between records that differ only in the order of their fields.

However, terms from \( F_\# \) translated to \( F_x \) retain their structure better, so it is easier to see what happens in the translation.

4.3 The translation

For each \( F_\# \) item \( i \), we will denote its \( F_x \) translation by \( i^0 \).

**Definition 4.1** The translation of types:

\[
\begin{align*}
T_0^0 &= X \\
(S \wedge T)^0 &= S_0^0 \times T_0^0 \\
[\{T\}]^0 &= [\{T^0\}] \\
(\forall X \leq T; X \# U; S)^0 &= \forall X : * \cdot (X - T^0) \rightarrow (X - U^0 - X) \rightarrow S^0
\end{align*}
\]

**Definition 4.2** The translation of contexts:

\[
\begin{align*}
\emptyset^0 &= \emptyset \\
(\Gamma, x : T)^0 &= \Gamma_0^0, x : T_0^0 \\
(\Gamma, X \leq T; X \# U)^0 &= \Gamma_0^0, X : *, \pi_X : X_0 \rightarrow T_0^0, \pi_X : X_0 \rightarrow U_0 \rightarrow X
\end{align*}
\]

**Lemma 4.3** If \( \Gamma \vdash \text{ok} \) then \( \Gamma_0^0 \vdash \text{ok}^0 \).

**Lemma 4.4** If \( \Gamma \vdash T : * \) then \( \Gamma_0^0 \vdash \text{\#_\#}^0 : * \).

The translation of the other sorts of judgments depends on the derivation of the judgment: for each derivation rule a translation is given, i.e. a (derived) rule in \( F_x \). The translation of a derivation of a judgment is defined inductively on the structure of the derivation, by replacing occurrences of the derivation rules by their translations.

**Definition 4.5** The translation of derivation \( \Delta \) is of the following form:

If \( \Delta \) derives \( \Gamma \vdash e : T \), then \( \Delta^0 \) derives \( \Gamma_0^0 \vdash e' : T_0^0 \) for some \( F_x \) term \( e' \)

If \( \Delta \) derives \( \Gamma \vdash T \leq U \), then \( \Delta^0 \) derives \( \Gamma_0^0 \vdash c : T_0^0 - U_0^0 \) for some term \( c \)

If \( \Delta \) derives \( \Gamma \vdash T \# U \), then \( \Delta^0 \) derives \( \Gamma_0^0 \vdash p : T_0^0 - U_0^0 - T_0^0 \) for some term \( p \)

where \( c \) stands for a coercion, and \( p \) for an overwriter.

First we give the translation of some rules already present in \( F_x \), then we will give some typical \( F_\# \) rules. We will use the meta-variables \( c \) and \( d \) for coercions, and \( p \) and \( q \) for overwriters. We assume that the new variables, introduced by the translation, are always fresh. For the rest of the rules we refer to appendix B.

A coercion is made explicit in the translation of the subsumption rule:

\[
\frac{\Gamma_0^0 \vdash x : S_0^0}{\Gamma_0^0 \vdash x : T_0^0} \quad (\text{T - Subsumption})^0
\]

Of course the coercion from a type to itself is the identity:

\[
\frac{\Gamma_0 \vdash x : T_0^0}{\Gamma_0 \vdash x : T_0^0} \quad (\text{S - Refl})^0
\]

And we already gave the coercions for intersection types:

\[
\frac{\Gamma_0 \vdash S_0^0 \times S_0^2 : *}{\Gamma_0 \vdash \lambda x : S_0^0 \times S_0^2 \cdot x : S_0^1 \times S_0^2 \rightarrow S_0^0} \quad (\text{S - Inter - LB})^0
\]

Now the rules that are specific for \( F_\# \). We discussed the (T-\text{With}) rule:

\[
\begin{align*}
\frac{\Gamma_0 \vdash x : R_0^0 \vdash x : S_0^0 \vdash p : R_0^0 - S_0^0 - R_0^0}{\Gamma_0 \vdash (p \cdot r, s, a) : R_0^0 \times S_0^0} \quad (\text{T - With})^0
\end{align*}
\]
The premise of the (#-Sub) rule says that $T$ has more fields than $S$, so all fields of $S$ have to be overwritten. The translation accordingly:
\[
\Gamma \vdash \alpha : T^0 \rightarrow S^0 \\
\Gamma \vdash \lambda x : \llbracket |T^0| \rrbracket \cdot \lambda y : \llbracket |U^0| \rrbracket \cdot \tau : \llbracket |T^0| \rrbracket \rightarrow \llbracket |U^0| \rrbracket \rightarrow \llbracket |T^0| \rrbracket (\# - Sub)^0
\]
The records in the conclusion of the (#-Diff) rule have no common fields, so nothing has to change in the left record:
\[
\llbracket |T^0| \rrbracket : * \\
\llbracket |U^0| \rrbracket : * 
\]
\[
\Gamma \vdash \lambda x : \llbracket |T^0| \rrbracket \cdot \lambda y : \llbracket |U^0| \rrbracket , x : \llbracket |T^0| \rrbracket \rightarrow \llbracket |U^0| \rrbracket \rightarrow \llbracket |T^0| \rrbracket (\# - Diff)^0
\]
The (#-Inter-L) is fairly straightforward, both parts of the left type have to be updated:
\[
\Gamma \vdash p : R^0 \rightarrow T^0 \rightarrow R^0 \\
\Gamma \vdash q : S^0 \rightarrow T^0 \rightarrow S^0 \\
\Gamma \vdash R^0 \times S^0 : * 
\]
\[
\Gamma \vdash \lambda r : \llbracket |T^0| \rrbracket . \lambda s : \llbracket |U^0| \rrbracket \cdot (p \cdot (\pi_1 r) \cdot (\pi_2 r) \cdot t) : \llbracket |R^0 \times S^0| \rrbracket \rightarrow \llbracket |T^0| \rrbracket \rightarrow \llbracket |U^0| \rrbracket (\# - Inter - L)^0
\]
but the (#-Inter-R) rule is more tricky. The left value has to be updated by two values. This can happen in two orders, that won’t make any difference if the translation is coherent:
\[
\Gamma \vdash p : R^0 \rightarrow S^0 \rightarrow R^0 \\
\Gamma \vdash q : R^0 \rightarrow T^0 \rightarrow R^0 \\
\Gamma \vdash S^0 \times T^0 : * 
\]
\[
\Gamma \vdash \lambda r : \llbracket |S^0 \times T^0| \rrbracket . \lambda s : \llbracket |R^0 \times S^0| \rrbracket \rightarrow \llbracket |R^0| \rrbracket (\# - Inter - R)^0
\]
This concludes the explanation of the translation of #-rules. It is interesting to see now how the (T-TApp) rule is translated. In a system with subtyping, each type application gets a coercion in the translation, in this system with compatibility, there is an additional function, the overwriter.
\[
\Gamma \vdash f : \lambda X : S^0 \rightarrow (X \rightarrow T^0 \rightarrow R^0) \\
\Gamma \vdash c : V^0 \rightarrow S^0 \\
\Gamma \vdash p : V^0 \rightarrow T^0 \rightarrow V^0 
\]
\[
\Gamma \vdash f \cdot c \cdot p : U^0[X := V^0] (T - TApp)^0
\]
Lemma 4.6 All translated rules are derivable in $F_x$.

Examples:

1. A translation of
\[
\vdash \llbracket |Int, m:Char| \rrbracket \leq \llbracket |Int| \rrbracket \\
\vdash \lambda x : \llbracket |Int| \rrbracket \times \llbracket |m:Char| \rrbracket \cdot \pi_1 \cdot \pi_2 : \llbracket |Int| \rrbracket \times \llbracket |m:Char| \rrbracket \rightarrow \llbracket |Int| \rrbracket
\]
2. A translation of
\[
\vdash \llbracket |Int| \rrbracket \neq \llbracket |m:Char| \rrbracket \\
\vdash \lambda x : \llbracket |Int| \rrbracket \cdot \lambda s : \llbracket |m:Char| \rrbracket \cdot \tau : \llbracket |Int| \rrbracket \rightarrow \llbracket |m:Char| \rrbracket \rightarrow \llbracket |Int| \rrbracket
\]
3. A translation of
\[
\vdash \llbracket |Int| \rrbracket \neq \llbracket |m:Char| \rrbracket \\
\vdash \lambda x : \llbracket |Int| \rrbracket \cdot \lambda s : \llbracket |Int| \rrbracket \cdot \tau : \llbracket |Int| \rrbracket \rightarrow \llbracket |m:Char| \rrbracket \\
The identity function is generated by the subtyping judgment $\llbracket |Int| \rrbracket \leq \llbracket |Int| \rrbracket$ in the premise of the (#-Sub) rule. This makes the overwrite function less readable. Therefore we will sometimes apply some reduction steps in the terms.
4. A translation of
\[
\vdash \llbracket |Int| \rrbracket \cdot \llbracket |m:Char| \rrbracket \cdot \llbracket |Int| \rrbracket \cdot \llbracket |m:Char| \rrbracket \\
\vdash \lambda x : \llbracket |Int| \rrbracket \cdot \lambda s : \llbracket |Int| \rrbracket \cdot \tau : \llbracket |Int| \rrbracket \times \llbracket |m:Char| \rrbracket
\]
With the convention of example 3 we say the translation is:
\[
\vdash \lambda x : \llbracket |Int| \rrbracket \cdot \llbracket |m:Char| \rrbracket
\]
5. A translation of
\[
\vdash \llbracket |Int| \rrbracket \cdot \llbracket |m:Char| \rrbracket \neq \llbracket |Int| \rrbracket \\
\vdash \lambda x : \llbracket |Int| \rrbracket \times \llbracket |m:Char| \rrbracket \cdot \lambda s : \llbracket |m:Char| \rrbracket \cdot \pi_1 \cdot \pi_2 : \llbracket |Int| \rrbracket \times \llbracket |m:Char| \rrbracket
\]
6. In our explanation we will assume $S$ and $T$ are record types.
7. In a context where the type of an argument is clear, we will sometimes replace the type by _.
6. A translation of
\[ \Gamma \vdash \{\ell=7, m=a\} \{l:\text{Int}, m:\text{Char}\} \land \{l=5\} : \{l:\text{Int}, m:\text{Char}\} \]
is
\[ \{\ell:\text{Int} \times \{l:m=\text{Char}\}, \{l=5\}, \{l=5\}\} \]
(using example 5). This term reduces to \(\{\{l=5\}, \{m=\text{a}'\}\}, \{l=5\}\) so whatever coercion is applied to this term, the \(\ell\)-field will always be 5.

7. A translation of
\[ \Gamma \vdash \{\ell=7, m=a\} \{l:\text{Int}, m:\text{Char}\} \land \{l=5\} : \{l:\text{Int}, m:\text{Char}\} \]
is
\[ \{\ell:\text{Int} \times \{l:m=\text{Char}\}, \{l=5\}, \{l=5\}\} \]
The polymorphic update is translated to this simple function.

Discussion: We give here the reasons for not having a symmetry rule for \# . First of all, such a rule is inconsistent with specialized updating (section 2.8), so we lose some generality of the system. Suppose we don’t allow specialized updating and have a symmetry rule for \# . This symmetry has a complicating effect on the semantics, because the semantics (as described above) is not symmetric: If \(S \# T\), we have an overwriter that overwrites an \(S\) (of type \(S \to T \to S\)), but not the dual one that overwrites a \(T\). For a symmetric \# both are needed, so the translation of \(S \# T\) will be a pair of overwriters, with type \((S \to T \to S) \times (T \to S \to T)\), which is symmetric.

**Definition 4.7** Let \(e\) and \(e'\) be \(F\#\)-terms, with type \(T\) and translations \(f, f'\) respectively. Then \(e = f \equiv e'\) if \(f = f'\).

Here a formal meaning to equality in \(F\#\) is given via the translation.

### 4.4 Coherence

Coherence is the property that all translations of some \(F\#\) judgment are equal with respect to the equational theory of \(F\) (i.e. \(\beta\eta\)-convertible). In specific:

**Conjecture 4.8** Coherence of subtyping: Suppose \(\Gamma \vdash S \# T\) has derivations \(\Delta_1\) and \(\Delta_2\), giving rise to translations \(I^0 \vdash_{x} c_1 : S^0 \to T^0\) and \(I^0 \vdash_{x} c_2 : S^0 \to T^0\) respectively. Then \(c_1\) and \(c_2\) are equal in \(F\).

**Conjecture 4.9** Coherence of compatibility: Suppose \(\Gamma \vdash S \# T\) has derivations \(\Delta_1\) and \(\Delta_2\), giving rise to translations \(I^0 \vdash_{x} p_1 : S^0 \to T^0 \to S^0\) and \(I^0 \vdash_{x} p_2 : S^0 \to T^0 \to S^0\) respectively. Then \(p_1\) and \(p_2\) are equal in \(F\).

**Conjecture 4.10** Coherence of typing: Suppose \(\Gamma \vdash e : T\) has derivations \(\Delta_1\) and \(\Delta_2\), giving rise to translations \(I^0 \vdash_{x} e_1 : T^0\) and \(I^0 \vdash_{x} e_2 : T^0\) respectively. Then \(e_1\) and \(e_2\) are equal in \(F\).

As even for \(F\) coherence is not yet proved, the proofs of these conjectures are outside the scope of this article.

### 5 Conclusions

#### 5.1 Related work

**Record concatenation**

First we will compare \(F\#\) with other explicitly typed second-order record calculi, viz. Cardelli [Car 92], Cardelli and Mitchell [CM 91] and Harper and Pierce [HP 90]. All of these calculi have a restriction operator, denoted by \(\setminus\), that removes a field from a record, and a way of extending a record with one new field. A polymorphic record update is performed by removing the field, and extending the record with the one-field record containing the new value.

In principle the combination of extension and restriction is in some respects more powerful than our calculus without a way of removing a field. However, such power is not needed for...
a fairly extensive model of OOP, as section 3 suggests. Only for renaming of methods — not a very important feature of some OO languages — restrictions are necessary.

In the presence of a restriction operator, the combination of subtyping and a full concatenation (extend with a possibly unknown number of fields) is problematic; none of the systems has both. Subtyping is needed for a natural encoding of OO programs, concatenation for multiple inheritance. We dropped record restriction in favour of both other properties. The possibility of a polymorphic record update was retained by permitting concatenation of record with some common fields, as long as they are of the same type.

Now we will compare these calculi to $F\#$, in some more detail. For each calculus we will show how a polymorphic record update is implemented. In $F\#$ this function is:

$$\lambda R \subseteq \{ x : \text{Int} \}, \ r \in R \mapsto \{ x : \text{Int} \} \{ x = r \cdot x + 1 \}$$

$$\forall R \subseteq \{ x : \text{Int} \}, \ r \in R \mapsto \{ x : \text{Int} \}$$

The examples that we will give are largely adapted to our syntax.

• In [Car 92] Cardelli gives a record calculus based on row-variables. Just as in our system, Cardelli provides a translation of the system with records to a system with only cartesian products. There are two differences. First, his target system includes subtyping whereas ours does not. More importantly, there is some order fixed on the fields, say $1^0, 1^1, 1^2$ etc. The translation of a record type depends on this order. E.g. the type $\{ x : \text{Int}, y : \text{Real} \}$ will be translated to $\text{Int} \times (\text{Top} \times (\text{Real} \times \text{Top}))$. This interpretation of records makes it very difficult to make a concatenation.

Each extensible record has a row variable representing the unknown fields of the record. In this example of a polymorphic update:

$$\lambda X \subseteq \{ x : \text{Int}, X \}, \ z = r \cdot x + 1, \ y \in x$$

$$\forall X \subseteq \{ x : \text{Int}, X \}$$

$X$ is the row-variable representing all fields other than $x$.

• In [CM 91] Cardelli and Mitchell give a record calculus where both positive (a record has a field) and negative information (a record doesn't have a field) is determined by the record type. All record values that comply with this information belong to this record type. E.g. $\{ x : \text{Int} \}$ is the type of all records that have an $x$-field of type integer, such as $\{ x = 3 \}$ and $\{ x = 3, y = 'a' \}$. This means that the identity $(\lambda r : \{ x : \text{Int} \}, \ x) \in$ applied to the term reduces to $\{ x = 3, y = 'a' \}$. This is quite in contrast to $F\#$ where the emphasis is on coercions and applying the identity above can have an effect, in this case it would remove the y-field. An example of a type with negative information is $\{ y : \text{Int} \}$, which is the type of all records with an $x$-field, but without a $y$-field. The example of the polymorphic update can be written simply as

$$\lambda R \subseteq \{ x : \text{Int} \}, \ r \cdot x \mid x = r \cdot x + 1$$

$$\forall R \subseteq \{ x : \text{Int} \}, \ R \rightarrow R \mid x : \text{Int}$$

where $\mid$ is the extension with a field.

The very different philosophy makes it impossible to define a disjoint record concatenation, because two records, of whatever type, can always have a field in common. Furthermore this view complicates the semantics of the system.

• Harper and Pierce give in [HP 90] a calculus with only negative information and with a full concatenation operator on disjoint records. For negative information the same symbol as in $F\#$ for compatibility is used, viz. $\|$ . This relation expresses that two records are disjoint, so may be concatenated with $\|$

The polymorphic update is implemented by

$$\lambda R \| \{ x : \text{Int} \}, \ r \cdot x \| \{ x : \text{Int} \} \{ x = r \cdot x + 1 \}$$

$$\forall R \| \{ x : \text{Int} \}, \ R \| \{ x : \text{Int} \} \rightarrow R \| \{ x : \text{Int} \}$$

---

*Actually our concatenation is a little more liberal, but that is not relevant now.*
In fact the presence of fields is forced by the typing in value-abstractions, whereas the absence of fields is forced by type abstractions.

The most important drawback of this calculus is that it does not have subtyping, which seems to be necessary to make a natural encoding of object-oriented features.

Our update is easier to use than the others because the type that has to be supplied to the polymorphic update is just the type of the record, not the type of a part of the record.

The idea of using intersection types for record concatenation already appeared in Reynolds' imperative language Forsythe [Rey 88]. In Forsythe there is also a construction to make a value of an intersection type, called merge. Because Reynolds permits only merging with one field, there is no need for a compatibility relation. The restrictions for merge can be expressed directly by the derivation rules. Remarkable is that some totally different values may be combined. Translated to our framework, there are rules like

\[ \frac{\Gamma \vdash A:B \quad \Gamma \vdash [\langle \Gamma; C \rangle] : *}{\Gamma \vdash A:B \# [\langle \Gamma; C \rangle]} \quad (\text{-Fun-Rec}) \]

This means a function and a record may be combined. This is justified by the fact that in any context only one of the two can be used, so there is no danger of losing coherence.

A difference with our system is that it is possible to replace a field in a record with a value of some different type.

Modelling OOP concepts in a type system

There are different styles of OOP programming languages. First there are class based and delegation based languages. In class based languages, the implementation of an object is specified by its class, and a class can be extended to form a new class. In delegation based languages an object can be extended to form a new object. We will only study class based languages in this paper.

The class based languages have two categories: single-dispatch and multiple dispatch. In multiple-dispatch languages, such as Common Lisp [Ste 84], the method is determined by any number of arguments instead of just one for single-dispatch languages. For a model of this more symmetrical notion of method invocation, run-time type information is needed. We concentrate on the single-dispatch variants, such as SmallTalk [GR 83] and C++.

There appeared two ways to model the concept of encapsulation of single-dispatch OOP languages in a type-theoretic setting. The first is procedural abstraction, objects are encoded as elements of recursive record types. The second is type abstraction, which hides the type of the state, but permits the invocation of methods that can work on the state. A comparison of both methods can be found in [FM 94], a more theoretic treatment of the similarities and differences in [HP 95].

Our model is the same as the model of Pierce and Turner [PT 93] and Compagnoni and Pierce [CP 93], who chose for the second way, since the combination of subtyping with recursive types is still problematic for modelling inheritance. Because our type system is more powerful, the encoding is simpler. The differences are:

1. In both models, the state of a subclass (designated by B) is a longer record than the state of the corresponding superclass (designated by A). The methods of the superclass should not only work for A, but also for B. In their system, this is solved by introducing an extractor get to convert a B to an A, and an overwriter put that overwrites the A-part in a B. get is used whenever a method reads the state, and put whenever a method changes the state. The put is always used in conjunction with a non-polymorphic update, which can be implemented more elegantly using record concatenation.

In our system, the problem is solved by demanding that B is a subtype of and compatible with A. The first thing means that the state of the subclass can always be converted to a state of the superclass. This makes the extractor get redundant. The compatibility means that the state of the subclass may be concatenated with (a part of) the state of the superclass, so a field of the state of a subclass can be overwritten. In this way put becomes redundant.

In fact the translation we gave in section 4 translates the subtyping and compatibility demands in coercions (= extractors) and overwriters, so one could consider their model...
to be a (partial) translation of our model. The extractors and overwriters are explicit in their system and implicit in ours.

2. The encoding of inheritance of methods is also facilitated by the concatenation: the record of old methods is concatenated with the record of new methods. A similar construction is used for inheritance of the initial state. Concretely, the code for inheritance does not depend on definitions of superclasses etc. This suggests that other concepts of OOP, such as template classes, probably can be modelled more easily.

So $F_\#$ gives rise to a more direct and natural model of OOP than $F_\leq$.

**Haskell type classes**

A subtyping judgment $A \leq B$ is translated to a function of type $A \rightarrow B$. A compatibility judgment $A \# B$ is translated to a function of type $A \rightarrow B \rightarrow A$. This asks for a generalisation. Subtyping and compatibility are predicates on two types. Their translation depends on the two types. In general, there will not always be two types involved, in other useful cases it will be just one. We can have the unary predicate "has equality defined on it", let's denote that with $Eq_A$, with as translation $A \rightarrow A \rightarrow \text{Bool}$. We could have a derivation rule that says that $\text{Int}$ has equality defined on it, and a derivation rule that says that if $Eq_A$, then also $Eq(\text{List} A)$, where $\text{List} A$ is the type of lists of $A$. The equality on type $A$, can be invoked by $\text{is A}$, provided that $Eq_A$ is valid. Polymorphic functions can demand that a type variable has an equality, this is comparable to bounded quantification.

$Eq$ is an example of a so-called type class, as described by Wadler and Blott [WB 89] (the word type class has nothing to do with an object-oriented class). The derivation rules for type classes may be extended, but a few restrictions apply, that prevent the loss of coherence and make type checking for the type classes feasible. Even new type classes can be introduced. This type-class prevents the programmer having to make a boring definition of equality for every type that he uses. So the definition of equality is implicit, just like the definitions of coercions.

One could see both subtyping and compatibility as binary type classes, because they are translated to normal functions, just as ordinary type classes. But in other respects there are some differences. The most important one is that for $\leq$ the place where to use a coercion is implicit, in contrary to $\#$ or $Eq$ where the place to use the corresponding function is always explicit. Furthermore, type-checking the subtype or compatibility predicate is more complicated than unary predicates, because the former are binary and have some sort of transitivity law. The unification of type classes on one hand, and subtyping and compatibility on the other hand is an interesting point for future work.

### 5.2 Future work

**Proving coherence** As we pointed out in section 4.4, of $F_\lambda$ is not yet proved. A major problem is caused by rule (T-Inter') on page 11, maybe the variant of the rules that we propose is easier proved coherent.

**Typing algorithm** For implementation, a typing algorithm is very important. Such a typing algorithm relies on a subtyping and a compatibility algorithm. For the subtyping and typing of similar systems algorithms have been proposed [Pie 91b, Com 95], and adaption seems straightforward, but for compatibility a new algorithm has to be constructed.

Just as with subtyping, the rule for transitivity poses the most problems. However the subtyping premises present in the ($\#$-Trans) rule can help with determining the unknown intermediate type, and compatibility seems decidable.

**Equational semantics for $F_\#$** An equational semantics is a derivation system, in which judgments state that two terms are equal as seen from some type. In systems with subtyping, this is a very important clause, since terms that are not equal on some type, may be equal on other types (e.g. all records are equal when seen from the empty record type).

Equational semantics gives a direct way of reasoning about equality of terms if there are no reduction rules (as in $F_\#$). It can also be used to guide the design of axioms for
proof rules of the system. The design of the equational semantics itself can be guided by the translation to $F_x$.

**Higher order types** As we have seen in section 3, we need to extend $F_\#$ to use it for a satisfying model of OOP. The higher-order aspect is the most involving: It complicates all points above.

**More demands for type variables** The sort of demands that one can place on a type variable in a system is a balance between pragmatics and feasibility (e.g. decidable typechecking). We have shown that we can come a long way towards OOP, by having type variables compatible to one type. We can also define a system where it is possible to demand from a variable that a certain type is compatible with the variable (the other way around). It turns out that with (the higher order version of) this system, we can define template classes relatively simple.

**Variant types** A record is seen as an intersection of labeled fields. The dual notion of intersection types are union types [Pie 91a]. So it is tempting to see a variant type as the union of labeled fields. We will see however, that this cannot be done in a coherent way.

Consider the judgment $\Gamma \vdash [\{l: Int\}] \land [\{m: Char\}] \leq [\{l: Int\}] \lor [\{m: Char\}]$. There are two simple derivations of it, one via type $[\{l: Int\}]$, and one via type $[\{m: Char\}]$. The coercions differ accordingly, the former always delivers the $l$-choice and the latter delivers the $m$-choice, so coherence is lost unfortunately.

A solution for this is to have two kinds of labeled types. However, for one thing there doesn't seem to be such a big need for flexible variants, and for another there is still less known about union types than about intersection types. The combination of $F_\#$ with normal (not extendable) labeled variants seems unproblematic, however.

**Alternative for $\leq$** The coupled use of $\leq$ and $\#$ in polymorphic update functions suggests a system where only the coupled use of subtyping and compatibility is permitted. This new relation, denoted by $\preceq$, is a variant of subtyping that allows updating. So the translation of $S \subseteq T$ is $S - T \times S - T - S$, both a coercion and an overwriter. There are two uses for $\preceq$, the subsumption rule and an update rule:

\[
\Gamma \vdash c : S \quad \Gamma \vdash S \subseteq T \\
\Gamma \vdash c : T
\]

(T-Subsum)

\[
\Gamma \vdash c : S \quad \Gamma \vdash f : T \quad \Gamma \vdash S \subseteq T \\
\Gamma \vdash \text{update } S \quad T \quad e \quad f : S
\]

(T-Update)

Clearly, not all subtyping rules admissible for $\leq$ are admissible for $\preceq$, e.g. $\text{Int} \leq \text{Real}$ or contravariant arrow subtyping, but reflexivity, transitivity and record subtyping do transfer.

Such a system can be very interesting if only polymorphic updates are needed (but no record concatenation), since the syntax is almost as simple as the standard subtyping syntax. This approach has been taken in [HP 94].

**Acknowledgments**

Thanks to Kees Hemerik, Benjamin Pierce and Erik Poll for reading drafts of this article.

**References**


[Car 88] Luca Cardelli. A Semantics of Multiple Inheritance. Information and Computation 76.


[Pie 94] Benjamin C. Pierce. *Bounded Quantification is Undecidable*. Information & Computation 112. Also in [Pie 91b] and [GM 94].


A Rules of F#

Labels:
\[ \mathcal{L} \ ::= \text{the set of labels} \]

Types:
\[ T ::= X \quad \text{type variable} \]
\[ T \rightarrow T \quad \text{function type} \]
\[ \forall X \leq T; X \# T. \ T \quad \text{restricted quantification} \]
\[ \text{Top} \quad \text{top-type} \]
\[ \{ \mathcal{L} : T \} \quad \text{one-field record type} \]
\[ T \land T \quad \text{record concatenation} \]

Terms:
\[ e ::= x \quad \text{variable} \]
\[ \lambda x : T. \ e \quad \text{abstraction} \]
\[ e \ e \quad \text{application} \]
\[ \forall X \leq T; X \# T. \ e \quad \text{type abstraction} \]
\[ e \ T \quad \text{type application} \]
\[ \text{top} \quad \text{top-value} \]
\[ \{ \mathcal{L} = e \} \quad \text{one-field record} \]
\[ e \ T \text{ with } T \ e \quad \text{concatenation} \]
\[ e . \mathcal{L} \quad \text{field selection} \]

Contexts:
\[ \Gamma ::= \emptyset \quad \text{empty context} \]
\[ \Gamma, x : T \quad \text{term variable declaration} \]
\[ \Gamma, X \leq T; X \# T \quad \text{type variable declaration} \]

Judgments:
\[ \Gamma \vdash \text{ok} \quad \text{well-formed context} \]
\[ \Gamma \vdash T : * \quad \text{well-formed type} \]
\[ \Gamma \vdash S \leq T \quad \text{subtype} \]
\[ \Gamma \vdash S \# T \quad \text{compatible} \]
\[ \Gamma \vdash e : T \quad \text{well-typed term} \]

Meta-variables:
\[ \Gamma \quad \text{for contexts} \]
\[ X \quad \text{for type variables} \]
\[ R, S, T, U, V \quad \text{for types} \]
\[ x \quad \text{for term variables} \]
\[ e, f \quad \text{for terms} \]
\[ l, m \quad \text{for labels} \]

Derivation rules:

Context formation:

\[ \emptyset \vdash \text{ok} \quad \text{(C-Empty)} \]
\[ \Gamma \vdash T : * \quad \text{x} \notin \text{dom}(\Gamma) \]
\[ \Gamma \vdash T : * 
\]
\[ \Gamma, x : T \vdash \text{ok} \quad \text{(C-Var)} \]
\[ \Gamma \vdash T : * \quad \Gamma \vdash U : * \quad \text{x} \notin \text{dom}(\Gamma) \]
\[ \Gamma, X \leq T; X \# U \vdash \text{ok} \quad \text{(C-TVar)} \]

Type formation:

\[ \Gamma, X \leq T; X \# U, \Gamma_2 \vdash \text{ok} \]
\[ \Gamma, X \leq T; X \# U, \Gamma_2 \vdash X : * \quad \text{(K-TVar)} \]
\[ \Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : * 
\]
\[ \Gamma \vdash T_1 \rightarrow T_2 : * \quad \text{(K-Arrow)} \]
Subtyping:

\[ \Gamma, X : T, X \# U \vdash S : * \quad (K-All) \]

\[ \Gamma \vdash \forall X : T, X \# U. S : * \]  

\[ \Gamma \vdash T : * \quad (K-Single) \]

\[ \Gamma \vdash R : * \quad \Gamma \vdash S : * \]

\[ \Gamma \vdash R \& S : * \quad (K-With) \]

\[ \Gamma \vdash \text{ok} \]

\[ \Gamma \vdash \text{Top} : * \quad (K-Top) \]

\[ \Gamma \vdash T : * \]

\[ \Gamma \vdash T \leq T \quad (S-Refl) \]

\[ \Gamma \vdash S \leq T \]

\[ \Gamma \vdash T \leq U \quad (S-Trans) \]

\[ \Gamma, X : T, X \# U, \Gamma \vdash \text{ok} \]

\[ \Gamma, X : T, X \# U, \Gamma \vdash X \leq T \quad (S-Var) \]

\[ \Gamma \vdash T_1 \leq S_1 \quad \Gamma \vdash S_2 \leq T_2 \]

\[ \Gamma \vdash S_1 \& S_2 \leq T_1 \& T_2 \quad (S-Inter-G) \]

\[ \Gamma, X : U, X \# V \vdash S \leq T \]

\[ \Gamma \vdash \forall X : U, X \# V. S \leq \forall X : U, X \# V. T \quad (S-All) \]

\[ \Gamma \vdash R \leq S \]

\[ \Gamma \vdash \llbracket R \rrbracket \leq \llbracket S \rrbracket \quad (S-Deep) \]

\[ \Gamma \vdash S_i \& S_2 : \star \quad i = 1, 2 \]

\[ \Gamma \vdash R \leq S \quad \Gamma \vdash R \leq T \]

\[ \Gamma \vdash R \leq S \& T \quad (S-Inter-LB) \]

\[ \Gamma \vdash \llbracket \llbracket T \rrbracket \rrbracket \& \llbracket \llbracket U \rrbracket \rrbracket \leq \llbracket \llbracket T \& U \rrbracket \rrbracket \quad (S-Label-Distr) \]

\[ \Gamma \vdash \llbracket \llbracket T \rrbracket \rrbracket \& \llbracket \llbracket U \rrbracket \rrbracket \leq \llbracket \llbracket T \& U \rrbracket \rrbracket \]

\[ \Gamma \vdash S : \star \]

\[ \Gamma \vdash S \leq \text{Top} \quad (S-Top) \]

Compatibility rules:

\[ \Gamma, X : T, X \# U, \Gamma \vdash \text{ok} \quad (\#-TVAR) \]

\[ \Gamma, X : T, X \# U, \Gamma \vdash X \# U \quad (\#-TVar) \]

\[ \Gamma \vdash R : * \]

\[ \Gamma \vdash \text{Top} \# R \quad (\#-Top-L) \]

\[ \Gamma \vdash R : * \]

\[ \Gamma \vdash R \# \text{Top} \quad (\#-Top-R) \]

\[ l \neq m \quad \Gamma \vdash T : * \quad \Gamma \vdash U : * \quad (\#-Diff) \]

\[ \Gamma \vdash \llbracket l[T] \rrbracket \# \llbracket l[U] \rrbracket \]

\[ \Gamma \vdash R \# T \quad \Gamma \vdash S \# T \quad (\#-Inter-L) \]

\[ \Gamma \vdash R \& S \quad \Gamma \vdash R \# T \quad (\#-Inter-R) \]

\[ \Gamma \vdash R \# \llbracket T \rrbracket \]

\[ \Gamma \vdash \llbracket T \rrbracket \# \llbracket U \rrbracket \quad (\#-Sub) \]

\[ \Gamma \vdash T \# \llbracket T \rrbracket \]

\[ \Gamma \vdash \llbracket T \rrbracket \# \llbracket U \rrbracket \quad (\#-Equal) \]

\[ \Gamma \vdash S \leq T \quad \Gamma \vdash S \leq T \quad (\#-Trans) \]

\[ \Gamma \vdash T \# T \]

43
Term formation:

\[
\frac{\Gamma_1, x : T, \Gamma_2 \vdash o}{\Gamma_1, x : T, \Gamma_2 \vdash x : T} \quad \text{(T-Var)}
\]

\[
\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \lambda x : S. e : S \to T} \quad \text{(T-Abs)}
\]

\[
\frac{\Gamma \vdash f : S \to T, \Gamma \vdash e : S}{\Gamma \vdash f \ e : T} \quad \text{(T-App)}
\]

\[
\frac{\Gamma, X \leq S; X \# T \vdash e : U}{\Gamma \vdash \lambda X \leq S; X \# T. e : \forall X \leq S; X \# T. U} \quad \text{(T-TAbs)}
\]

\[
\frac{\Gamma \vdash f : \forall X \leq S; X \# T. U, \Gamma \vdash V \leq S, \Gamma \vdash V \# T}{\Gamma \vdash f \ V : U[X := V]} \quad \text{(T-TApp)}
\]

\[
\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \text{top}; \text{Top}} \quad \text{(T-Top)}
\]

\[
\frac{\Gamma \vdash e : T}{\Gamma \vdash \{l = e\} : \{l : T\}} \quad \text{(T-Single)}
\]

\[
\frac{\Gamma \vdash r : R, \Gamma \vdash s : S, \Gamma \vdash R \# S}{\Gamma \vdash r \ R \ \text{with} \ s : R \wedge S} \quad \text{(T-With)}
\]

\[
\frac{\Gamma \vdash e : \{l : T\}}{\Gamma \vdash e.l : T} \quad \text{(T-Sel)}
\]

\[
\frac{\Gamma \vdash e : S, \Gamma \vdash S \leq T}{\Gamma \vdash e : T} \quad \text{(T-Subsumption)}
\]
B  Translation of $F_\#$ to $F_\times$

Notes:
- Translations of rules not presented here are straightforward.
- We will use the meta-variables $c$ and $d$ for coercions, and $p$ and $q$ for overwriters. We assume that the new variables, introduced by the translation, are always fresh.

Subtyping:

$$\Gamma \vdash x : T^0 : *$$
$$\Gamma \vdash x : \lambda x : T^0. : x : T^0 \rightarrow T^0$$  \quad (S - Refl)^0

$$\Gamma \vdash_{c x} : S^0 \rightarrow T^0 \quad \Gamma \vdash_{d x} : T^0 \rightarrow U^0$$  \quad (S - Trans)^0

$$\Gamma, x : c x : X \rightarrow T^0, \ldots \vdash \Gamma, x : d x : X \rightarrow U^0 \quad \Gamma, x : \ldots \vdash \Gamma, x : ok$$  \quad (S - TVar)^0

$$\Gamma, x : c x : X \rightarrow T^0, \ldots \vdash \Gamma, x : \ldots \vdash \Gamma, x : ok$$  \quad (S - TVar)^0

$$\Gamma, x : c x : X \rightarrow T^0, \ldots \vdash \Gamma, x : \ldots$$

Compatibility rules:

$$\Gamma, x : c x : X \rightarrow T^0, \ldots \vdash \Gamma, x : \ldots \vdash \Gamma, x : ok$$  \quad (# - TVar)^0

$$\Gamma, x : c x : X \rightarrow T^0, \ldots \vdash \Gamma, x : \ldots$$

$$\Gamma, x : \ldots \vdash \Gamma, x : ok$$  \quad (# - TVar)^0
Term formation (selection):

\[
\begin{align*}
\Gamma^0 \vdash f : \forall X : (X \to S^0) \to (X \to T^0 \to X) \to U^0 \\
\Gamma^0 \vdash c : V^0 \to S^0 \\
\Gamma^0 \vdash p : V^0 \to T^0 \to V^0 \\
\Gamma^0 \vdash f : V^0 \to U^0 \text{[}X := V^0\text{]} \\
\Gamma^0 \vdash r : R^0 \\
\Gamma^0 \vdash s : S^0 \\
\Gamma^0 \vdash p : R^0 \to S^0 \to R^0 \\
\Gamma^0 \vdash (p \circ s, s) : R^0 \times S^0 \\
\Gamma^0 \vdash e : S^0 \\
\Gamma^0 \vdash c : S^0 \to T^0 \\
\Gamma^0 \vdash c : e : T^0 \\
(T - 
\text{TApp})^0 \\
(T - \text{With})^9 \\
(T - \text{Subsumption})^9
\end{align*}
\]
<table>
<thead>
<tr>
<th>Computing Science Reports</th>
<th>Department of Mathematics and Computing Science</th>
</tr>
</thead>
<tbody>
<tr>
<td>In this series appeared:</td>
<td>Eindhoven University of Technology</td>
</tr>
</tbody>
</table>

| 93/01 | R. van Geldrop | Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36. |
| 93/02 | T. Verheoфф | A continuous version of the Prisoner’s Dilemma, p. 17 |
| 93/03 | T. Verheoфф | Quicksort for linked lists, p. 8. |
| 93/04 | E.H.L. Aarts, J.H.M. Korst, P.J. Zwistering | Deterministic and randomized local search, p. 78. |
| 93/05 | J.C.M. Baeten, C. Verhoeфф | A congruence theorem for structured operational semantics with predicates, p. 18. |
| 93/06 | J.P. Velkamp | On the unavoidability of metastable behaviour, p. 29 |
| 93/07 | P.D. Moerland | Exercises in Multiprogramming, p. 97 |
| 93/08 | J. Verhoeveг | A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32. |
| 93/10 | K.M. van Hee | Systems Engineering: a Formal Approach Part II: Frameworks, p. 44. |
| 93/16 | H. Schepers, J. Hoorman | A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27 |
| 93/17 | D. Alstein, P. van der Stok | Hard Real-Time Reliable Multicast in the DEDOS system, p. 19. |
| 93/18 | C. Verhoeфф | A congruence theorem for structured operational semantics with predicates and negative premises, p. 22. |
| 93/19 | G.J. Houben | The Design of an Online Help Facility for ExSpect, p.21. |
| 93/21 | M. Codish, D. Dams, G. Fié, M. Bruynooghe | Freeness Analysis for Logic Programs - And Correctness, p. 24 |
| 93/22 | E. Poll | A Typewriter for Bijective Pure Type Systems, p. 28. |
| 93/23 | E. de Kogel | Relational Algebra and Equational Proofs, p. 23. |
| 93/24 | E. Poll and Paula Seveги | Pure Type Systems with Definitions, p. 38. |
| 93/26 | W.M.P. van der Aalst | Multi-dimensional Petri nets, p. 25. |
| 93/27 | T. Kloks and D. Kratsch | Finding all minimal separators of a graph, p. 11. |
| 93/28 | F. Kamareddine and R. Nederpelt | A Semantics for a fine λ-calculus with de Bruijs indices, p. 49. |
| 93/29 | R. Post and P. De Bra | GOLD, a Graph Oriented Language for Databases, p. 42. |
| 93/30 | J. Deogun, T. Kloks, D. Kratsch, H. Müller | On Vertex Ranking for Permutation and Other Graphs, p. 11. |
93/31 W. Korver

Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.

93/32 H. ten Eikelder and H. van Geldrop


93/33 L. Loyoens and J. Moonen

ILIAS, a sequential language for parallel matrix computations, p. 20.

93/34 I.C.M. Baeten and J.A. Bergstra

Real Time Process Algebra with Infinitesimals, p. 39.

93/35 W. Ferrer and P. Severi

Abstract Reduction and Topology, p. 28.

93/36 J.C.M. Baeten and J.A. Bergstra

Non Interleaving Process Algebra, p. 17.

93/37 J. Brouckrefi

Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.

93/38 C. Verhoeff

A general conservative extension theorem in process algebra, p. 17.

93/39 W.P.M. Nuijten

Job Shop Scheduling by Constraint Satisfaction, p. 22.

E.H.L. Aarts


D.A.A. van Erp Taalman Kip

93/40 K.M. van Hee

94/01 A. Blijdema

Temporal operators viewed as predicate transformers, p. 11.

94/02 P.M.P. Rambags

Automatic Verification of Regular Protocols in P/T Nets, p. 23.

94/03 B.W. Watson

A taxonomy of finite automata construction algorithms, p. 87.

94/04 B.W. Watson

A taxonomy of finite automata minimization algorithms, p. 23.

94/05 E.J. Luit

J.M.M. Marin

94/06 T. Kloke

J. Krasch

J. Spinrad


94/07 W. v.d. Aalst

P. De Bra

G.J. Houben

Y. Komatsky

94/08 R. Gerth

Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

94/09 P. America

M. van der Kammen

R.P. Nederpelt

O.S. van Roosmalen

H.C.M. de Swart

94/10 F. Kamareddine

R.P. Nederpelt

The object-oriented paradigm, p. 28.

94/11 L.B. Hartman

K.M. van Hee

Canonical typing and IT-conversion, p. 51.

94/12 J.C.M. Baeten

J.A. Bergstra

94/13 T. Zhou

J. Hooman

94/14 T. Basten

T. Kunz

J. Black

M. Coffin

D. Taylor

94/15 K.R. Apt

R. Bol


94/16 O.S. van Roosmalen

A Hierarchical Diagrammatic Representation of Class Structure, p. 22.

94/17 J.C.M. Baeten

J.A. Bergstra

Process Algebra with Partial Choice, p. 16.
94/10 T. Verheijen
94/11 J. Peleska, C. Huizing, C. Petersohn
94/12 T. Kloks, D. Kratsch, H. Müller
94/13 R. Seljé
94/14 W. Peremans
94/15 R.J.M. Vaessens, E.I.H. Aarts, J.K. Lenstra
94/16 R.C. Backhouse, H. Doornbos
94/17 S. Mauw, M.A. Reniers
94/18 F. Kamareddine, R. Nederpelt
94/19 B.W. Watson
94/20 R. Bloo, F. Kamareddine, R. Nederpelt
94/21 B.W. Watson
94/22 B.W. Watson
94/23 S. Mauw and M.A. Reniers
94/24 D. Dams, O. Grumberg, R. Gerth
94/25 T. Kloks
94/26 R.R. Hoogerwoord
94/27 S. Mauw and H. Mulder
94/28 C.W.A.M. van Overevel, M. Verhoeven
94/29 J. Hooman
94/30 J.C.M. Baeten, J.A. Bergstra, G.H. Stefanescu
94/31 B.W. Watson, R.E. Watson
94/32 J.J. Vereijken
94/33 T. Laan
94/34 R. Bloo, F. Kamareddine, R. Nederpelt
94/35 J.C.M. Baeten, S. Mauw
94/36 F. Kamareddine, R. Nederpelt
94/37 T. Basten, R. BöI, M. Voorhoeve
94/38 A. Bijlsma, C.S. Scholten

The testing Paradigm Applied to Network Structure. p. 31.
A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
Ups and Downs of Type Theory, p. 9.
Job Shop Scheduling by Local Search, p. 21.
Mathematical Induction Made Calculational, p. 36.
An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
Refining Reduction in the Lambda Calculus, p. 15.
The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
Beyond β-Reduction in Church's λ, p. 22.
The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
An algebraic semantics of Message Sequence Charts, p. 43.
Abstract Interpretation of Reactive Systems: Abstractions Preserving ∀CTL*, 3CTL* and CTL*, p. 28.
K_{1p}-free and W_{1p}-free graphs, p. 10.
On the foundations of functional programming: a programmer's point of view, p. 54.
Correctness of Real Time Systems by Construction, p. 22.
Process Algebra with Feedback, p. 22.
A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
A formalization of the Ramified Type Theory, p.40.
The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
Delayed choice: an operator for joining Message Sequence Charts, p. 15.
Canonical typing and II-conversion in the Barendregt Cube, p. 19.
Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
Point-free substitution, p. 10.
On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160.

Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.

The λ-cube with classes of terms modulo conversion, p. 16.

On Π-conversion in Type Theory, p. 12.

Fixed-Point Calculus, p. 11.

The Construction of a small Communication Library, p. 16.

Formalizing Process Algebraic Verifications in the Calculus of Constructions, p. 49.

Concrete process algebra, p. 134.

An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.

A Type Inference Algorithm for Pure Type Systems, p. 20.

A Quantitative Analysis of Iterated Local Search, p. 23.

Drawing Execution Graphs by Parsing, p. 10.

Preservation of Strong Normalisation for Explicit Substitution, p. 12.

Discrete Time Process Algebra, p. 20

Mathpad: A System for On-Line Preparation of Mathematical Documents, p. 15
104x755

95/11 R. Seljée
Deductive Database Systems and integrity constraint checking, p. 36.

95/12 S. Mauw and M. Reniers
Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.

95/13 B.W. Watson and G. Zwaan

95/14 A. Ponec, C. Verhoef,
S.F.M. Vlijmen (eds.)
De proceedings: ACP95, p.

95/15 P. Niebert and W. Penczek
On the Connection of Partial Order Logics and Partial Order Reduction Methods,
p. 12.

95/16 D. Dam, O. Grumberg, R. Gerth
Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.

95/17 S. Mauw and E.A. van der Meulen
Specification of tools for Message Sequence Charts, p. 36.

95/18 F. Kamareddine and T. Laan
A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths,

95/19 J.C.M. Baeten and J.A. Bergstra
Discrete Time Process Algebra with Abstraction, p. 15.

95/20 F. van Raamsdonk and P. Severi
On Normalisation, p. 33.

95/21 A. van Deursen
Axiomatizing Early and Late Input by Variable Elimination, p. 44.

95/22 B. Arnold, A. v. Deursen, M. Res
An Algebraic Specification of a Language for Describing Financial Products,
p. 11.

95/23 W.M.P. van der Aalst
Petri net based scheduling, p. 20.

95/24 F.P.M. Dignum, W.P.M. Nuijten,
L.M.A. Janssen

95/25 L. Feijs
Synchronous Sequence Charts In Action, p. 36.

95/26 W.M.P. van der Aalst
A Class of Petri nets for modeling and analyzing business processes, p. 24.

95/27 P.D.V. van der Stok, J. van der Wal

95/28 W. Fokkink, C. Verhoef
A Conservative Look at term Deduction Systems with Variable Binding, p. 29.

95/29 H. Jurjus
On Nesting of a Nonmonotonic Conditional, p. 14

95/30 I. Hidders, C. Hoskins, J. Paredaens

95/31 P. Kolb, D. Dam and R. Gerth
Practical Symbolic Model Checking of the full \( \mu \)-calculus using Compositional
Abstractions, p. 17.

95/32 W.M.P. van der Aalst
Handboek simulatie, p. 51.

95/33 J. Engelfriet and J.J. Vereijken
Context-Free Graph Grammars and Concatenation of Graphs, p. 35.