International Journal of Foundations of Computer Science

TYPE INFERENCE FOR FIRST-CLASS MESSAGES WITH FEATURE CONSTRAINTS

MARTIN MÜLLER*
Universität des Saarlandes, 66041 Saarbrücken, Germany
mmueller@ps.uni-sb.de

and

SUSUMU NISHIMURA†
RIMS, Kyoto University, Sakyo-ku, Kyoto 606-8502, Japan
nisimura@kurims.kyoto-u.ac.jp

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We present a constraint system, OF, of feature trees that is appropriate to specify and implement type inference for first-class messages. OF extends traditional systems of feature constraints by a selection constraint $x\langle y\rangle z$, "by first-class feature tree" y, which is in contrast to the standard selection constraint $x[f]y$, "by fixed feature" f. We investigate the satisfiability problem of OF and show that it can be solved in polynomial time, and even in quadratic time if the number of features is bounded. We compare OF with Treinen's system EF of feature constraints with first-class features, which has an NP-complete satisfiability problem. This comparison yields that the satisfiability problem for OF with negation is NP-hard. We even obtain NP-completeness, for a specific subclass of OF with negation that is useful for a related type inference problem. Based on OF we give a simple account of type inference for first-class messages in the spirit of Nishimura's recent proposal, and we show that it has polynomial time complexity: We also highlight an immediate extension of this type system that is desirable but makes type inference NP-complete.

Keywords: object-oriented programming; first-class messages; constraint-based type inference; complexity; feature constraints

1. Introduction

First-class messages add extra expressiveness to object-oriented programming. First-class messages are analogous to first-class functions in functional programming languages; a message triggers the call of an object's corresponding method, just as a functional argument represents the computation executed on application. For example, a **map** method can

*Partly supported by the Deutsche Forschungsgemeinschaft (DFG) through Sonderforschungsbereich 378 at the Universität des Saarlandes, Saarbrücken, 1996-98.

†Partly supported by the Japanese Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Encouragement of Young Scientists, 10780187, 1998.

be defined by means of first-class messages by

$$\mathsf{method}\ \mathbf{map}(\mathsf{o},\mathsf{l}) = \textit{for each message}\ \mathsf{m}\ \textit{in}\ \mathsf{l}\text{:}\quad \mathsf{o}\leftarrow\mathsf{m}$$

where o is an object, l is a list of first-class messages, and on execution of o←m, message m is sent to o.

First-class messages are more common in distributed object-oriented programming where they add crucial expressiveness. A typical use of first-class messages is the delegation of messages to other objects for execution. Such delegate objects are ubiquitous in distributed systems: for example, proxy servers enable access to external services (*e. g.*, ftp) beyond a firewall. A delegate object implementing a simple proxy server can be defined as follows.

$$\mathsf{let\ ProxyServer} = \{\ \mathbf{new}(\mathsf{o}) = \{\ \mathbf{send}(\mathsf{m}) = \mathsf{o}\leftarrow\mathsf{m}\}\ \};$$

This creates an object ProxyServer with a method **new** that receives an object o. The method returns another object that, on receipt of a message labeled **send** and carrying a message m, forwards m to o. To create a proxy to an FTP server, we can execute

$$\mathsf{let\ FtpProxy} = \mathsf{ProxyServer}\leftarrow\mathbf{new}(\mathsf{ftp});$$

where ftp refers to an FTP object. A typical use of this new proxy is the following one.

$$\mathsf{FtpProxy}\leftarrow\mathbf{send}(\mathbf{get}(\texttt{'paper.ps.gz'}))$$

Delegation cannot be easily expressed without first-class messages, since the relevant messages are not known statically and must be abstracted over by a variable m.

In a programming language with records, abstraction over messages corresponds to abstraction over field names. For example, one might want to use a function let f = fn x => y.x; to select any field x from record y. Static typing of first-class messages and of first-class record fields is difficult for an analogous reason: both message or record field identifiers may be bound to varying values depending on the execution. Neither first-class messages nor first-class record fields are not supported in statically typed languages such as Standard ML [20], Objective Caml [35], or Haskell [30]. There is a type system for extensible records with first-class record fields by Gaster [14], but it is restrictive in not allowing a single record field type to mention varying record fields.

Recently, the second author has proposed an extension to the ML type system that can deal with first-class messages [25]. In the spirit of Ohori's polymorphic record type system [28], he has formulated a type system for first-class messages as a kinded type system where, intuitively, kinds describe classes (or types) of types. The corresponding type inference procedure is given in terms of kinded unification. However, the presentation of both the type system and the type inference in [25] are formally involved and not easily understandable or suitable for further analysis.

In this paper, we give a constraint-based formulation of type inference for first-class messages in the spirit of [25] and analyze its complexity. To this end, we define a new constraint system over feature trees [3] that we call OF (*objects* and *features*). This constraint system extends known systems of feature constraints [6, 7, 38, 40] by a new, tailor-made constraint: this new constraint is motivated by the type inference of a message sending statement $\mathsf{o} \leftarrow \mathsf{m}$, and pinpoints the key design idea underlying Nishimura's system.

We investigate the (incremental) satisfiability problem for OF and show that it can be solved in polynomial time, namely in $O(n^4)$ in general and in time $O(n^2)$ for the important special case that the number of features is bounded. We also investigate the satisfiability problem for OF constraints with negation by comparing OF with Treinen's feature constraint system EF [40]. We show that checking satisfiability for positive and negative OF constraints is NP-hard in general, and NP-complete when negation is restricted to a certain class of formulas.

Based on OF, we define monomorphic type inference for first-class messages. Our formulation considerably simplifies the original one based on kinded unification. One advantage of our formulation is that dealing with constraints is more flexible than dealing with the large kinded types according to [25]. More important even is the fact that we strictly separate the types (semantics) from the type descriptions (syntax), whereas the original system confused syntax and semantics by allowing variables in the types themselves.

Our type system reformulates the monomorphic part of Nishimura's original type system as a constrained type system based on OF. This reformulation turns out to be insightful on its own (see Section 3). From our complexity analysis of OF we obtain that monomorphic type inference for first-class messages can be done in (incremental) polynomial time. Incrementality is important since it allows for modular (piece-wise) program analysis without loss of efficiency over global (monolithic) program analysis.

Our constraint-based setup of type inference allows us to explain ML-style polymorphic type inference [15, 19] as an instance HM(OF) of the HM(X) scheme [26]: Given a monomorphic type system based on a constraint system X, the authors give a generic construction of HM(X), *i. e.*, type inference for ML-style (*i. e.*, Hindley/Milner) polymorphic constrained types. Type inference for the polymorphic system remains DEXPTIME-complete, of course [16, 17].

In the remainder of the introduction we summarize the main idea of the type system for first-class messages and of the constraint system OF.

1.1. The Type System

The type system contains types for objects and messages and explains what type of messages can be sent to an object of a given type. An object type is a labeled collection of method types (a product of types) marked by obj. For example, the object o defined by

$$\mathsf{let}\ \mathsf{o} = \{\ \mathbf{pos}(\mathsf{x}) = \mathsf{x}{>}0,\ \mathbf{neg}(\mathsf{p}) = \neg\ \mathsf{p}\};$$

implements two methods **pos** and **neg** that behave like functions from integer and boolean, respectively, to Boolean. Hence, it has the following object type.footnote In contrast to what is common in the types community, the colons in the type $\mathsf{obj}(\mathbf{pos}{:}\mathsf{int} \to \mathsf{bool}, \mathbf{neg}{:}\mathsf{bool} \to \mathsf{bool})$ do not separate items from their type annotation, but rather the field names from the associated type components. This notation is inherited from the literature on feature trees and record typing.

$$\mathsf{obj}(\mathbf{pos}{:}\mathsf{int} \to \mathsf{bool}, \mathbf{neg}{:}\mathsf{bool} \to \mathsf{bool})\,.$$

When a message $f(M)$ is sent to an object, the method corresponding to the message label f is selected and then applied to the message argument M. Since a message identifier may refer to many specific messages at run-time, its type is a labeled collection of the corresponding argument types (a sum of types) marked by msg. For example, the expression

$$\mathsf{let}\ \mathsf{m} = \mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathbf{pos}(42)\ \mathsf{else}\ \mathbf{neg}(\mathsf{true});$$

defines that m be assigned one of the message **pos**(42) or **neg**(true) depending on the boolean b. Since this disjunction can, in general, not be resolved statically, m is given the disjunctive message type

$$\mathsf{msg}(\mathbf{pos}{:}\mathsf{int}, \mathbf{neg}{:}\mathsf{bool})\,.$$

In the context of the previous definitions, the expression $\mathsf{o} \leftarrow \mathsf{m}$ is well-typed since two conditions hold:

1. For both labels that are possible for m, **pos** and **neg**, the object o implements a method that accepts the corresponding message arguments of type int or bool.
2. Both methods **pos** and **neg** have the same return type, here bool. Thus the type of $\mathsf{o} \leftarrow \mathsf{m}$ is statically known even though the message is not.

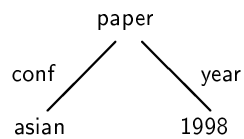
These are the invariants that Nishimura's type system [25] is constructed to guarantee.

In this paper, we devise a type system for first-class messages that is based on these invariants as well – very similar to that of [25]. In the course of the formal developments, it will become apparent that our type system is slightly weaker than Nishimura's original one in that it admits more programs: Some of them are well-typed only because certain methods are never executed. This weakness is, however, the price to pay in order to achieve polynomial time complexity of type inference. The obvious way of extending our type system in order to bridge this gap makes type inference NP-complete.

1.2. Constraint-based Type Inference

It is well-known that many type inference problems have a natural and simple formulation as the satisfiability problem of an appropriate constraint system (*e. g.* [29, 32, 42]). Constraints were also instrumental in generalizing the ML-type system towards record polymorphism [28, 34, 43], overloading [8, 27] and subtyping [1, 12, 32] (see also [26] for further references).

Along this line, we use *feature trees* [3] as the semantic domain of the constraint system underlying our type system. A feature tree is a possibly infinite tree with unordered marked edges (called *features*) and with marked nodes (called *labels*), where the features departing from the same node must be pairwise distinct. For example, the picture on the right shows a feature tree with two features conf and year that is labeled with paper at the root and asian and 1998, respectively, at the leaves.



Feature trees have been used as the interpretation domain for a class of constraint languages called *feature con-*

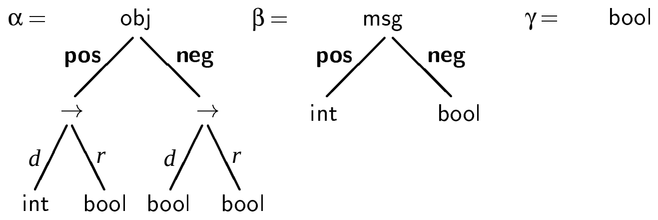


Figure 1: Interpretation of Types in Feature Trees

straints [5–7, 23, 38, 40]. These are a class of feature description logics, and, as such, have a long tradition in knowledge representation and in computational linguistics and *constraint-based grammars* [31, 36]. More recently, they have been used to model record structures in constraint programming languages [2, 33, 37, 38].

We use feature trees to represent types. Feature trees can naturally represent the types of all kinds of data structures with labeled components such as object, record, or message types. A base type like int is a feature tree with label int and no features. A message type $\mathsf{msg}(f_1{:}\tau_1,\ldots,f_n{:}\tau_n)$ is a feature tree with label msg, features $f_1,\ldots,f_n$, and corresponding subtrees $\tau_1,\ldots,\tau_n$, and an object type $\mathsf{obj}(f_1{:}\tau_1 \to \tau_1',\ldots,f_n{:}\tau_n \to \tau_n')$ is a feature tree with label obj, features $f_1,\ldots,f_n$, and corresponding subtrees $\tau_1 \to \tau_1'$ through $\tau_n \to \tau_n'$; the arrow notation $\tau \to \tau'$ in turn is a notational convention for a feature tree with label $\to$ and subtrees τ, τ' at fixed and distinct features d and r, the names of which should remind one of "domain" and "range".

A *feature constraint system* is given by a language of constraints that contains certain *primitive constraints* and is closed at least under conjunction, and their interpretation over feature trees. The most fundamental constraint languages proposed are those of FT [3] providing for primitive constraints for equality on feature trees, feature selection, and labeling, and of CFT [38] that extends FT by a constraint on the set of possible features (a so-called arity constraint).

Roughly, we obtain our constraint system OF from CFT by the addition of a primitive constraint whose semantics reflects the intuition underlying well-typed message passing in Nishimura's system. The constraint language of OF is this one:

$$\varphi \quad ::= \quad x = y \ \mid \ a(x) \ \mid \ x[f]y \ \mid \ F(x) \ \mid \ x\langle y\rangle z \ \mid \ \varphi \wedge \varphi'$$

The first three primitive constraints are well-known: The symbol $=$ denotes equality on feature trees, $a(x)$ holds if x denotes a feature tree that is labeled with a at the root, and $x[f]y$ holds if the subtree of (the denotation of) x at feature f is defined and equal to (the denotation of) y. For a finite set of features F, the constraint $F(x)$ holds if x has *at most* the features in F at the root; in contrast, the arity constraint of CFT forces x to have *exactly* the features in F. The constraint $x\langle y\rangle z$ is new. It holds for three feature trees τ_x, τ_y, and τ_z if (i) τ_x has at least the features at the root that τ_y has, and if (ii) for all root features f at τ_y, the subtree of τ_x at f equals $\tau_y.f \to \tau_z$ (where $\tau_y.f$ is the subtree of τ_y at f).

It is not difficult to see that $x\langle y\rangle z$ is tailored to type inference of message sending.[a] For

[a]The notation of the constraint $x\langle y\rangle z$ is chosen to indicate its close relationship to $x[f]y$. For its application to

example the ProxyServer above gets the following polymorphic constrained type:

$$\forall\alpha\beta\gamma\,.\mathsf{obj}(\alpha)\wedge\mathsf{msg}(\beta)\wedge\alpha\langle\beta\rangle\gamma\Rightarrow\ \mathsf{obj}(\mathbf{new}{:}\alpha\to\mathsf{obj}(\mathbf{send}{:}\beta\to\gamma))\,.$$

Using notation from [26], the matrix of this type has two parts, a term part right of $\Rightarrow$, and a constraint part left of $\Rightarrow$. The term part describes an object that accepts a message labeled **new** with argument type α, returning an object that accepts a message labeled **send** with argument type β and has corresponding return type γ. The constraint part in addition requires that α be an object type, β be a message type appropriate for α, and the corresponding method type in α have return type γ. A possible monomorphic instance of this type would bind said three variables was follows: $\alpha=\mathsf{obj}(\mathbf{pos}{:}\mathsf{int}{\to}\mathsf{bool},\mathbf{neg}{:}\mathsf{bool}{\to}\mathsf{bool})$, $\beta=\mathsf{msg}(\mathbf{pos}{:}\mathsf{int},\mathbf{neg}{:}\mathsf{bool})$, and $\gamma=\mathsf{bool}$. Figure 1 illustrates these bindings in terms of the corresponding feature trees.

Plan. Section 2 defines the constraint system OF, considers the complexity of its satisfiability problem, and proves that an extension of system OF with negation makes the satisfiability problem NP-complete. Section 3 applies OF to the type inference for first-class messages and shows that its complexity is polynomial. Section 4 discusses properties of the corresponding type system in relation to this complexity result. Section 5 concludes the paper.

2. The Constraint System OF

2.1. Syntax and Semantics

The constraint system OF is defined as a class of constraints along with their interpretations over feature trees. We assume three infinite sets $\mathcal{V}$, of *variables*, with typical members x,y, and z, $\mathcal{F}$, of *features*, with typical member f, where $\mathcal{F}$ contains at least d and r, and $\mathcal{L}$, of *labels*, with typical members a and b that contains at least $\to$. The meaning of constraints depends on this label. We write $\bar{x}$ for a sequence $x_1,\ldots,x_n$ of variables whose length n does not matter, and $\bar{x}{:}\bar{y}$ for a sequence of matching pairs $x_1{:}y_1,\ldots,x_n{:}y_n$. We use similar notation for other syntactic categories.

We also write $x\doteq y$ to denote that variables x and y are syntactically identical.

2.1.1. Feature Trees

A *path* π is a word over features. The *empty path* is denoted by ε and the free-monoid concatenation of paths π and π' as $\pi\pi'$; we have $\varepsilon\pi=\pi\varepsilon=\pi$. Given paths π and π', π' is called a *prefix of* π if $\pi=\pi'\pi''$ for some path π''. We write $|\pi|$ to denote the length of path π and also write $f\in\pi$ if there is an occurrence of feature f in π. A *tree domain* is a non-empty prefix closed set of paths. A *feature tree* τ is a pair (D,L) consisting of a tree domain D and a *labeling function* $L:D\to\mathcal{L}$. Given a feature tree τ, we write D_τ for its tree domain and L_τ for its labeling function. The *arity* $\mathsf{ar}(\tau)$ of a feature tree τ is defined by $\mathsf{ar}(\tau)=D_\tau\cap\mathcal{F}$. If $\pi\in D_\tau$, we write as $\tau.\pi$ the subtree of τ at path π: formally $D_{\tau.\pi}=\{\pi'\ \mid\ \pi\pi'\in D_\tau\}$

type inference, the following reading might be helpful: $x\langle y\rangle z$ has two parts, namely '$x\langle y$' and '$\rangle z$'. '$x\langle y$' represents the message y sent to object x (where $\langle$ is a stylized $\leftarrow$) and '$\rangle z$' represents the result z.

and $L_{\tau.\pi} = \{(\pi',a) \mid (\pi\pi',a) \in L_\tau\}$. A feature tree is *finite* if its tree domain is finite, and *infinite* otherwise. The *cardinality* of a set S is denoted by $\#S$. Given feature trees $\tau_1,\ldots,\tau_n$, distinct features $f_1,\ldots,f_n$, and a label a, we write as $a(f_1{:}\tau_1,\ldots,f_n{:}\tau_n)$ (simply a, when $n=0$) the feature tree whose domain is $\{\varepsilon\} \cup \bigcup_{i=1}^n \{f_i\pi \mid \pi \in D_{\tau_i}\}$ and whose labeling is $\{(\varepsilon,a)\} \cup \bigcup_{i=1}^n \{(f_i\pi,b) \mid (\pi,b) \in L_{\tau_i}\}$. We use $\tau_1 \to \tau_2$ to denote the feature tree τ with $L_\tau = (\varepsilon,\to)$, $\mathsf{ar}(\tau) = \{d,r\}$, $\tau.d = \tau_1$, and $\tau.r = \tau_2$.

2.1.2. Syntax

The class of *OF constraints* φ is defined by the following abstract syntax.

$$
\begin{array}{rcll}
\varphi & ::= & x = y & \text{(Equality)} \\
 & \mid & a(x) & \text{(Labeling)} \\
 & \mid & x[f]y & \text{(Selection)} \\
 & \mid & F(x) & \text{(Arity Bound)} \\
 & \mid & x\langle y\rangle z & \text{(Object Selection)} \\
 & \mid & \varphi \wedge \varphi' & \text{(Conjunction)}
\end{array}
$$

We call $x=y$, $a(x)$, $x[f]y$, $F(x)$, and $x\langle y\rangle z$ *primitive OF constraints*. A first-order formula built from OF constraints and existential quantification is called an *existential OF formula*.

We write $\varphi' \subseteq \varphi$ if all primitive constraints in φ' are also contained in φ, and we write $x=y \in \varphi$ [etc.] if $x=y$ is a primitive constraint in φ [etc.]. We denote with $F(\varphi)$, $L(\varphi)$, and $V(\varphi)$ the set of features, labels, and variables occurring in a constraint φ. The *size* $S(\varphi)$ of a constraint φ is defined as the number of variable, feature, and label occurrences in φ.

2.1.3. Semantics

We interpret OF constraints in the structure $\mathcal{FT}$ of feature trees. The signature of $\mathcal{FT}$ contains the symbol $=$, for every $a \in \mathcal{L}$ a unary relation symbol $a(\cdot)$, for every $f \in \mathcal{F}$ a binary relation symbol $\cdot[f]\cdot$, for every finite subset F of $\mathcal{F}$ a unary relation symbol $F(\cdot)$, and the ternary relation symbol $\cdot\langle\cdot\rangle\cdot$. We interpret $=$ as equality on feature trees and the other relation symbols as follows:

$$
\begin{array}{lll}
a(\tau) & \text{iff} & (\varepsilon,a) \in L_\tau \\
\tau[f]\tau' & \text{iff} & \tau.f = \tau' \\
F(\tau) & \text{iff} & \mathsf{ar}(\tau) \subseteq F \\
\tau\langle\tau'\rangle\tau'' & \text{iff} & \forall f \in \mathsf{ar}(\tau') : f \in \mathsf{ar}(\tau) \text{ and } \tau.f = \tau'.f \to \tau''
\end{array}
$$

Let Φ and Φ' be first-order formulas built from OF constraints with the usual first-order connectives $\vee$, $\wedge$, $\neg$, $\to$, *etc.*, and quantifiers. We call Φ *satisfiable* (valid) if Φ is satisfiable (valid) in $\mathcal{FT}$. We say that Φ *entails* Φ', written $\Phi \models_{\text{OF}} \Phi'$, if $\Phi \to \Phi'$ is valid, and that Φ is *equivalent* to Φ', written $\Phi \models\mid_{\text{OF}} \Phi'$, if $\Phi \leftrightarrow \Phi'$ is valid.

A key semantic difference between the selection constraints $x[f]y$ and $x\langle y\rangle z$ is that "selection by (fixed) feature" $x[f]y$ is functional, while object "selection by (first-class) feature tree" $x\langle y\rangle z$ is not, as expressed by the following statements.

$$\models_{\text{OF}} \quad x[f]y \wedge x[f]y' \quad \to \quad y = y' \tag{1}$$

$$\not\models_{\text{OF}} \quad x\langle y\rangle z \wedge x\langle y\rangle z' \quad \to \quad z = z' \tag{2}$$

The second implication is not valid since y may have no subtrees: In this case, the constraint $x\langle y\rangle z$ does not constrain z at all. That is, the following implication is valid.

$$\models_{\mathrm{OF}} \qquad \{\}(y) \quad \rightarrow \quad \forall z\, x\langle y\rangle z \tag{3}$$

If, however, y is known to have at least one feature at the root, then object selection becomes functional, too. For arbitrary f, the following holds:

$$\models_{\mathrm{OF}} \qquad y[f]y' \wedge x\langle y\rangle z \wedge x\langle y\rangle z' \quad \rightarrow \quad z = z' \tag{4}$$

The implications (3) and (4) are crucial for the polynomial complexity of OF satisfiability and they are also significant for type inference (see Section 3).

2.1.4. Feature Terms

For convenience, we will use *feature terms* [3] as a generalization of first-order terms: Feature terms t are built from variables by feature tree construction $a(f_1{:}t_1,\ldots,f_n{:}t_n)$ (denoting a when $n = 0$) where the features $f_1,\ldots f_n$ are required to be pairwise distinct.

Equations between feature terms can be straightforwardly expressed as a conjunction of OF constraints $x = y$, $a(x)$, $F(x)$, $x[f]y$, and existential quantification. For example, the equation $x = a(f{:}b)$ corresponds to the formula $\exists y\, (a(x) \wedge \{f\}(x) \wedge x[f]y \wedge b(y) \wedge \{\}(y))$. In analogy to the notation $\tau_1 \rightarrow \tau_2$, we use the additional abbreviation $x = y \rightarrow z$ for the equation $x = \rightarrow(d{:}y, r{:}z)$.

For the sake of conciseness in the following sections, we shall also extend the flat syntax of constraints to a "nested" one by allowing feature terms wherever only variables were allowed before:

$$\varphi \quad ::= \quad \cdots \quad | \quad t_1 = t_2 \quad | \quad a(t) \quad | \quad t_1[f]t_2 \quad | \quad F(t) \quad | \quad t_1\langle t_2\rangle t_3$$

As usual, the semantics of these constraints is understood as a homomorphic lifting of the flat ones from variables to feature terms. Notice, however, that the extended syntax is not part of the formal system of OF, but just a notational convention. Every nested OF constraint can be written as a flat OF constraint with existential quantification.

2.2. Constraint Solving

Theorem 1. *The satisfiability problem of OF constraints is decidable in incremental polynomial space and time.*

For the proof, we define constraint solving by a rewriting system on constraints and the failure flag *fail*. The rules in Figure 2 should be clear by themselves. Note that the treatment of object selection in two separate rules is not essential simplifies the subsequent analysis, as we believe. We call a constraint is *closed* if it is invariant under the rules.

Theorem 1 follows from Propositions 1 through 4 as stated herebelow.

Proposition 1 (Correctness). *The rules in Figures 2 define equivalence transformations on constraints.*

$$\frac{\varphi \wedge x = y}{\varphi[y/x] \wedge x = y} \quad \text{if } x \in \mathcal{V}(\varphi) \text{ and } x \neq y \qquad \text{(Substitution)}$$

$$\frac{\varphi \wedge x[f]y \wedge x[f]z}{\varphi \wedge x[f]z \wedge y = z} \qquad \text{(Selection)}$$

$$\frac{\varphi \wedge F(x) \wedge F'(x)}{\varphi \wedge F \cap F'(x)} \qquad \text{(Arity Intersection)}$$

$$\frac{\varphi}{\varphi \wedge x[f]x'} \quad \begin{array}{l}\text{if } x\langle y\rangle z \wedge y[f]y' \in \varphi \text{ and} \\ \quad \text{not exists } z : x[f]z \in \varphi, \quad x' \text{ fresh}\end{array} \qquad \text{(Object Selection I)}$$

$$\frac{\varphi}{\varphi \wedge x' = y' \to z} \quad \begin{array}{l}\text{if } x\langle y\rangle z \wedge y[f]y' \wedge x[f]x' \in \varphi \text{ and} \\ \quad x' = y' \to z \notin \varphi\end{array} \qquad \text{(Object Selection II)}$$

$$\frac{\varphi \wedge a(x) \wedge b(x)}{\text{fail}} \quad \text{if } a \neq b \qquad \text{(Label Clash)}$$

$$\frac{\varphi \wedge F(x) \wedge x[f]x'}{\text{fail}} \quad \text{if } f \notin F \qquad \text{(Arity Clash)}$$

Figure 2: Constraint Solving Rules

Proof. We check rule by rule. Rules (Substitution), (Selection), (Label Clash), and (Arity Clash) are standard rules for solving feature constraints. Rule (Arity Intersection) allows one to normalize a constraint to contain at most one arity bound per variable. (Object Selection I) reflects the fact that $x\langle y\rangle z$ implies all features necessary for y to be also necessary for x, and (Object Selection II) establishes the selection relation $x\langle y\rangle z$ at a feature f known for both x and y. □

Notice that the number of fresh variables introduced in rule (Object Selection I) is bounded: This rule adds at most one fresh variable per constraint $x\langle y\rangle z$ and feature f and the number of both is constant during constraint solving. For the subsequent analysis, it is convenient to think of the fresh variables as fixed *once and for all* for every constraint φ. Hence, we define the finite set :

$$V'(\varphi) \quad =_{def} \quad V(\varphi) \cup \{v_{x,f} \in \mathcal{V} \mid x \in V(\varphi), f \in F(\varphi), v_{x,f} \text{ fresh}\}$$

Proposition 2 (Termination). *The rewrite system in Figures 2 terminates on all OF constraints* φ*.*

Proof. Let φ be an arbitrary constraint. Obviously, $F(\varphi)$ is a finite set and the number of occurring features is fixed since no rule adds new feature symbols. Secondly, recall that the number of fresh variables introduced in rule (Object Selection I) is bounded. Call a variable x *eliminated* in a constraint $x = y \wedge \varphi$ such that $x \not\equiv y$ if $x \notin V(\varphi)$. We use the constraint measure $(O_1(\varphi), O_2(\varphi), NE(\varphi), S(\varphi))$ defined by

$O_1(\varphi)$**:** number of sextuples (x,y,z,x',y',f) of non-eliminated variables $x,y,z,x',y' \in V'(\varphi)$ and features $f \in F(\varphi)$ such that $x\langle y\rangle z \wedge x[f]x' \wedge y[f]y' \in \varphi$ but $x' = y' \rightarrow z \notin \varphi$.

$O_2(\varphi)$**:** number of pairs (x,f) of non-eliminated variables $x \in V'(\varphi)$ and features $f \in F(\varphi)$ such that there exists y, y' and z with $x\langle y\rangle z \wedge y[f]y' \in \varphi$ but $x[f]x' \notin \varphi$ for any x'.

$NE(\varphi)$**:** number of non-eliminated variables.

$S(\varphi)$**:** size of constraint as defined in Section 2.1.2.

The measure of φ is bounded from below and strictly decreased by every rule application as the following table shows. This proves our claim.

	O_1	O_2	NE	S
(Arity Intersection)	$=$	$=$	$=$	$<$
(Selection)	$=$	$=$	$=$	$<$
(Substitution)	$\leq$	$\leq$	$<$	
(Object Selection I)	$=$	$<$		
(Object Selection II)	$<$			

□

Proposition 3 (Polynomial Complexity). *We can implement the rewrite system in Figure 2 such that it uses at most space $O(n^3)$ and incremental time $O(n^4)$, and at most linear space and incremental time $O(n^2)$ if the number of features is bounded.*

Proof. See Section 2.3.1 for details. □

Proposition 4. *Every OF constraint φ which is closed under the rules in Figure 2 (and hence is different from fail) is satisfiable.*

Proof. See Section 2.3.2 for details. □

2.3. Proofs on Constraint Solving

2.3.1. Proposition 3: Constraint Solving has Polynomial Complexity

We implement the constraint solver as a rewriting on pairs (P,S) where S is the *store* that flags failure or represents a satisfiable constraint in a solved form, and where P is the *pool* (multiset) of primitive constraints that still must be added to S. To decide satisfiability of φ we start the rewriting on the pool of primitive constraints in φ and the empty store and check the failure flag on termination.

Define $n_i = \#V(\varphi)$, $n_f = \#F(\varphi)$, $n_v = n_i + n_i \cdot n_f = \#V'(\varphi)$, wherein the index φ is left implicit throughout the paper. The index i refers to the *i*nitially available variables in φ.

Data Structures. We use the usual union-find algorithm with path compression [18] for the representation of equivalence classes on equated variables. It uses a data structure of size $O(n_v)$ that allows the addition of a new equation in time $O(n \cdot \alpha(n_v))$ where $\alpha(n_v)$ is the inverse of the Ackermann function.

In addition, the store contains the following:

1. for every variable $x \in V'(\varphi) \backslash V(\varphi)$, a flag whether or not it has been introduced before: size $O(n_v)$
2. for every $x \in V(\varphi)$, at most one label a per variable $x \in V(\varphi)$ to represent constraints $a(x)$: size $O(n_i)$

   For the newly introduced variables in $V'(\varphi) \backslash V(\varphi)$ the label is always $\rightarrow$.
3. for every $x \in V'(\varphi)$ at most one variable entry y per feature f to represent constraints $x[f]y$: size $O(n_v \cdot n_f)$
4. for every $x \in V(\varphi)$, a boolean array of size n_f to represent $F(x)$: size $O(n_i \cdot n_f)$

   For all newly introduced variables in $V'(\varphi) \backslash V(\varphi)$ the arity bound is $\{d, r\}$ and, therefore, need not be represented explicitly.

   This representation allows one to decide in constant time whether or not $\neg\exists y\, x[f]y$ is implied by the store:
5. A list of object selection constraints $x\langle y\rangle z$: size $O(n)$.

   This size estimation exploits the fact that the constraint never introduces new selection constraints $x\langle y\rangle z$.
6. A directed graph G_φ whose nodes are the initial variables and whose edges are (x, y) such that there exists $y\langle x\rangle z$ for some z. This graph is represented by an incidence matrix mapping each node to an array of outgoing edges. This graph has overall $O(n)$ edges: size $O(n)$.

   The graph G_φ allows depth-first tree traversal in time $O(n)$.

This data structure has overall size

$$O(\overset{x=y}{n_v} \overset{+}{+} \overset{a(x)}{n_i} \overset{+}{+} \overset{x[f]y}{n_v \cdot n_f} \overset{+}{+} \overset{F(x)}{n_i \cdot n_f} \overset{+}{+} \overset{x\langle y\rangle z}{n}) = O(n_v \cdot n_f + n)$$

which is $O(n)$ if the number of features is bounded and $O(n_i \times n_f \times n_f + n) = O(n^3)$ otherwise. It allows to check in time $O(\alpha(n_v))$ whether it contains a given primitive constraint and to add the primitive constraint, if missing. This is clear in the non-incremental (off-line) case where n_v, n_i, and n_f are fixed. In the incremental (on-line) case, where n_v, n_i, and n_f may grow proportional to n in the worst case, we can use dynamically extensible hash tables [9] to retain (amortized) constant time check and update for primitive constraints.

One-step Satisfiability. Each step of the algorithm removes a primitive constraint from the pool P, adds it to the store S, and then derives all its immediate consequences under the constraint solving rules: Amongst them, equations $x = y$ and selections $x[f]y$ are put back into the pool, while selections $x\langle y\rangle z$ and arity bounds $F(x)$ are directly added to the store.

We show that every step can be implemented such that it costs linear time.

The subsequent discussion is understood modulo equality. This means that, every time a primitive constraint is picked up from the pool, the first operation is the replacement of each variable by its representative in the corresponding equivalence class. Since the union-find data structure allows one to lookup in constant time for a variable the representative of its equivalence class, this preprocessing does not change the complexity considerations.

We consider the primitive constraints one by one.

$F(x)$**:** We check rules (Arity Intersection) and (Arity Clash). If the store already contains an arity constraint $F'(x)$, we replace $F'(x)$ by $F \cap F'(x)$ which can be computed in time $O(n_f)$, otherwise we simply add $F(x)$ in time $O(1)$. Next, we check for all features f known for x, *i. e.*, in time $O(n_f)$, whether or not f is contained in the new arity. The overall cost is $O(n_f)$.

$a(x)$**:** It suffices to check applicability of rule (Label Clash) and add $a(x)$ to the store. This can obviously done in constant time $O(1)$.

$x[f]y$**:** We must consider rules (Selection), and (Object Selection I/II).

(Selection) We check whether the store contains $x[f]y'$ for some y'. If so, we add $y = y'$ to the pool and terminate (we need not consider the rules (Object Selection I/II) in this case); if not, we simply add $x[f]y$ and proceed. Furthermore, we check whether $F(x)$ exists with $f \notin F$. Both can be done in constant time $O(1)$.

(Object Selection I) We compute the set of all variables to which the existence of feature f propagates from x. This can be done by a depth-first search through the graph G_φ containing an edge (x,y) for every constraint $y\langle x\rangle z$.

For all these $O(n_i)$ variables we check whether the selection entry of z at f is filled. If not, we add $z[f]v_{z,f}$, *i. e.*, $O(n)$ selection constraints in the worst case.
The cost is $O(n)$.

(Object Selection II)

- For all $O(n)$ selection constraints of the form $z\langle x\rangle z'$ such that $z[f]z''$ exists for some z'', we assert $z'' = y \to z'$ as follows: We add $z''[d]y$ and $z''[r]z'$ to the pool, and $\{d,r\}(z'')$ to the store.
  In addition, we may add $O(n)$ new selection constraints.
  The cost is $O(n)$.
- For all $O(n)$ selection constraints of the form $x\langle z\rangle z'$ such that $z[f]z''$ exists for some z'', assert $y = z'' \to z'$. We do this dually to the previous case and with the same resources. This costs $O(n)$.

The overall cost is $O(n)$.

This step adds $O(n)$ new selection constraints.

$x\langle y\rangle z$**:** We first add $x\langle y\rangle z$ to the list of object selection constraints and set up the graph G_φ by simply adding an edge (x,y). This costs $O(1)$.

Then we consider (Object Selection I/II):

(Object Selection I) For all features f such that the store contains $y[f]y'$ for some y', we must assert that x has feature f too. This can be done in total time $O(n_f)$.

(Object Selection II) For all features f such that the store contains $y[f]y'$ and $x[f]x'$ for some x',y' the constraint $x' = y' \to z$ may have to be added as in the application of the same rule above. This costs $O(n_f)$.

In addition, this step might introduce $O(n_f)$ new selection constraints.

The overall cost of adding $x\langle y\rangle z$ is $O(n_f)$.

This step adds $O(n_f)$ new selection constraints.

$x = y$**:** If x and y are equal, nothing needs to be done. Otherwise, we must consider rule (Substitution) first, and then all other rules. At first, the equivalence classes of x and y are merged, which can be done in time $O(\alpha(n_v))$.

Secondly, all constraints on y must be transferred to x (or vice versa). This is done by an additional case distinction.

$a(y)$**:** Adding $a(x)$ costs constant time $O(1)$.

$F(y)$**:** Adding $F(y)$ costs time $O(n_f)$.

$y[f]z$**:** The $O(n_f)$ selection constraints $x[f]z$ will be added to the pool in time $O(n_f)$.

$y\langle y'\rangle y''$**:** All selections on y' by f have to be propagated to x by (Object Selection I/II). Notice that, however, for all features f of y' a selection constraint $y[f]z$ has been asserted when that object selection constraint was entered into the store. Hence propagation to y (equivalently propagation to x, after x and y are equated) is treated by the other step of satisfiability checking (*c.f.* the case $x[f]y$ (Object Selection I/II))

We need not touch the list of object selection constraints. Only what we have to do is to retain the consistency of the graph G_φ by merging the out-going edges of y to those of x. This costs $O(n)$.

$y'\langle y\rangle y''$**:** By a similar argument, the cost is shown to be $O(n)$.

In summary, one step of the algorithm costs $O(n)$, and every step may at most add a single equation and $O(n)$ selection constraints.

Putting it all together. It remains to estimate the number of steps:

- There are at least $O(n)$ steps needed for touching all primitive constraints in φ.
- Amongst the new equations, there are at most $O(n_v)$ relevant ones, in the sense that one can at most enforce n_v non-trivial equations before all variables are equated. That is, all but $O(n_v)$ equations cost constant time.

- Amongst the new selection constraints, there are at most $O(n_v \cdot n_f)$ relevant ones since adding a selection constraint $x[f]y$ induces immediate work only if x has no selection constraint on f yet. The others will generate a new equation and terminate then. Hence, all but $O(n_v \cdot n_f)$ selection constraints cost constant time.

In summary, there are

$$O(n_v + n_v \cdot n_f) = O(n_v \cdot n_f)$$

steps that cost $O(n)$ each. Each of these steps may add a single equation and $O(n)$ selections each of which may add a new equation itself. Hence we have

$$O(n_v \cdot n_f \cdot (1+n)) \quad = \quad O(n_v \cdot n_f \cdot n)$$

steps that cost $O(1)$ each. Overall, the algorithm has the complexity

$$O(n_v \cdot n_f \cdot n) \quad = \quad O(n_i \cdot {n_f}^2 \cdot n)$$

Since $O(n_f) = O(n_i) = O(n)$ in general, this bound is $O(n^4)$. If the number of features is bounded, *i. e.*, $O(n_f) = O(1)$, the bound is rather $O(n^2)$.

2.3.2. Proposition 4: Constraint Solving is Complete

In order to show that every constraint closed under the rules in Figure 2 is satisfiable, we need some additional machinery:

First, we define a notion of *path reachability* similar to the one used in earlier work on feature constraints, such as [10, 22, 23]. For all paths π and constraints φ, we define $\overset{\varphi}{\leadsto}_\pi$ as the smallest binary relation satisfying the following conditions. We read $x \overset{\varphi}{\leadsto}_\pi y$ as "y is reachable from x over path π in φ"

$$\begin{array}{lll} x \overset{\varphi}{\leadsto}_\varepsilon x & \text{if} & x \in V(\varphi) \\ x \overset{\varphi}{\leadsto}_\varepsilon y & \text{if} & x = y \in \varphi \\ x \overset{\varphi}{\leadsto}_f y & \text{if} & x[f]y \in \varphi \\ x \overset{\varphi}{\leadsto}_{\pi\pi'} y & \text{if} & x \overset{\varphi}{\leadsto}_\pi z \text{ and } z \overset{\varphi}{\leadsto}_{\pi'} y. \end{array}$$

Likewise, we define $x \overset{\varphi}{\leadsto}_\pi a$ reading as "label a can be reached from x over path π in φ":

$$x \overset{\varphi}{\leadsto}_\pi a \quad \text{if} \quad x \overset{\varphi}{\leadsto}_\pi y \text{ and } a(y) \in \varphi$$

Path reachability satisfies the following closure conditions.

Lemma 1.

1. *Whenever $x \overset{\varphi}{\leadsto}_{\pi f} y$ there exists z such that $x \overset{\varphi}{\leadsto}_\pi z$ and $z \overset{\varphi}{\leadsto}_f y$.*

2. *Whenever $x \overset{\varphi}{\leadsto}_{f\pi} y$ there exists z such that $x \overset{\varphi}{\leadsto}_f z$ and $z \overset{\varphi}{\leadsto}_\pi y$.*

Moreover, we observe the following simple facts.

Lemma 2.

1. *If φ is closed under rule (Substitition) and $x = y \in \varphi$ such that $x \not\doteq y$, then x does not occur in any other primitive constraint in φ apart from $x = y$.*

2. *If φ is closed under rule (Selection) and (Substitition) and $x \overset{\varphi}{\rightsquigarrow}_\pi y, x \overset{\varphi}{\rightsquigarrow}_\pi z$, then $y \doteq z$.*

Proof. Statement 1 is trivial. Statement 2 follows by induction over π using closure of φ under (Selection) and (Substitution). $\square$

We now proceed to prove Proposition 4.

Fix an arbitrary label unit. For every closed constraint φ we define the mapping α_φ from variables into feature trees defined as follows.

$$\begin{array}{lcl} D_{\alpha_\varphi(x)} & = & \{\pi \mid \text{exists } y: \ x \overset{\varphi}{\rightsquigarrow}_\pi y\} \\ L_{\alpha_\varphi(x)} & = & \{(\pi, a) \mid x \overset{\varphi}{\rightsquigarrow}_\pi a\} \cup \{(\pi, \mathsf{unit}) \mid \pi \in D_{\alpha_\varphi(x)} \text{ but } \nexists a : x \overset{\varphi}{\rightsquigarrow}_\pi a\} \end{array}$$

We have to show that this indeed defines a mapping into feature trees and that α_φ is a solution of φ.

1. α_φ defines a mapping into feature trees: Pick some variable $x \in \mathcal{V}(\varphi)$.

   $D_{\alpha_\varphi(x)}$ is *non-empty* since $\varepsilon \in D_{\alpha_\varphi(x)}$ due to $x \overset{\varphi}{\rightsquigarrow}_\varepsilon x$. $D_{\alpha_\varphi(x)}$ is *prefix-closed* due to Lemma 1.1. So, $D_{\alpha_\varphi(x)}$ is a tree domain.

   Let $(\pi, a), (\pi, b) \in L_{\alpha_\varphi(x)}$. If $a = \mathsf{unit}$, then $b = \mathsf{unit}$ by definition of $L_{\alpha_\varphi(x)}$. Otherwise, we prove by induction over π that $a = b$.

   $\pi = \varepsilon$: By definition of $L_{\alpha_\varphi(x)}$ we know that $x \overset{\varphi}{\rightsquigarrow}_\varepsilon a$ and $x \overset{\varphi}{\rightsquigarrow}_\varepsilon b$. Therefore, there exist variables $y, \ldots y_n$ and $z, \ldots, z_m$ such that

   $$\begin{array}{l} (x \doteq) y_1 = y_2, y_2 = y_3, \ldots, y_{n-1} = y_n, a(y_n) \in \varphi \\ (x \doteq) z_1 = z_2, z_2 = z_3, \ldots, z_{m-1} = z_m, b(z_m) \in \varphi \end{array}$$

   By Lemma 2.1 we know it must hold that $x \doteq y_0 \doteq \cdots \doteq y_{n-1} \doteq z_0 \doteq \cdots \doteq z_{m-1}$. We obtain $a = b$ from closure of φ under (Label Clash).

   $\pi = f\pi'$: By definition of $L_{\alpha_\varphi(x)}$ and Lemma 1.2 we know that there are variables x', x'' such that $x[f]x', x[f]x'' \in \varphi$, $x \overset{\varphi}{\rightsquigarrow}_f x', x' \overset{\varphi}{\rightsquigarrow}_{\pi'} a$ and $x \overset{\varphi}{\rightsquigarrow}_f x'', x'' \overset{\varphi}{\rightsquigarrow}_{\pi'} b$. From Lemma 2.2, we obtain that $x' \doteq x''$. Thus, $a = b$ follows directly from the induction assumption.

   Finally, $L_{\alpha_\varphi(x)}$ is total on $D_{\alpha_\varphi(x)}$ by definition.

2. α_φ is a solution of φ: We check every primitive constraint in φ.

   $x = y \in \varphi$: $D_{\alpha_\varphi(y)} \subseteq D_{\alpha_\varphi(x)}$ and $L_{\alpha_\varphi(y)} \subseteq L_{\alpha_\varphi(x)}$ follows directly from the definition of path reachability. The inverse inclusions follow from Lemma 2.1. Hence, $\alpha_\varphi(x) = \alpha_\varphi(y)$.

   $x[f]y \in \varphi$: $D_{\alpha_\varphi(y)} \subseteq D_{\alpha_\varphi(x).f}$ and $L_{\alpha_\varphi(y)} \subseteq L_{\alpha_\varphi(x).f}$ follows from definition of path reachability. The inverse inclusions follow from Lemma 2.2. Hence, $\alpha_\varphi(x).f = \alpha_\varphi(y)$.

$a(x) \in \varphi$: By definition of α_φ and path reachability, $(\varepsilon, a) \in L_{\alpha_\varphi(x)}$.

$F(x) \in \varphi$: If $f \in D_{\alpha_\varphi(x)}$, then there must exist variables $y_1, \ldots, y_n, z$ such that

$$(x \doteq) y_1, y_2, \ldots, y_{n-1} = y_n, y_n[f]z \in \varphi\,.$$

By Lemma 2.1 and $F(x) \in \varphi$, we know that $x \doteq y_1 \doteq \cdots \doteq y_n$ and $x[f]z \in \varphi$. Now, $\mathrm{ar}(x) \subseteq F$ follows from closure of φ under (Arity Clash).

$x\langle y\rangle z \in \varphi$: Let $f \in \mathrm{ar}(\alpha_\varphi(y))$. By an argument similar to the previous case using Lemma 2.1 we know that $y[f]y' \in \varphi$ for some y'. By closure of φ under (Object Selection I) this implies $x[f]x' \in \varphi$ for some x', and $\alpha_\varphi(x).f = \alpha_\varphi(x') = \alpha_\varphi(y).f \to \alpha_\varphi(z)$ follows from closure of φ under (Object Selection II).

2.4. Relation of OF to Known Feature Constraint Systems

Various feature constraint systems have been considered in the literature [3, 5, 23, 38, 40]. These comprise, amongst others, feature constraints from the following list.

$$\psi \quad ::= \quad x = y \mid a(x) \mid x[f]y \mid Fx \mid u = f \mid x[u]y \mid \psi \wedge \psi'\,.$$

The constraints $x = y$, $a(x)$, and $x[f]y$ are the ones of OF. The constraint $x[u]y$ is two-sorted: It contains variables x, y ranging over feature trees and a variable u ranging over features. In the arity constraint Fx, F is a finite set of features. It states that x has *exactly* the features in F at the root. That is, its semantics is given by

$$F\tau \quad \text{if} \quad \mathsf{ar}(\tau) = F$$

Apparently, both arity constraints are interreducible by means of (an exponential number of) disjunctions: $F(x) \leftrightarrow \bigvee_{F' \subseteq F} F'x$. The constraints of FT [3] contain $x = y$, $a(x)$, and $x[f]y$, CFT [38] extends FT by Fx, and EF [40] contains the constraints $x{=}y$, $a(x)$, $u = f$, Fx, and $x[u]y$.

Recall that OF cannot express the fact that a feature tree has a feature at the root.[b] In contrast, EF can by means of existential quantification over the feature selector:

$$\exists u\, \exists y\, (x[u]y)$$

The satisfiability problems for FT and CFT are quasi-linear [38]. In contrast, the satisfiability problem for EF is NP-complete [40]. Treinen shows NP-hardness of satisfiability for EF by reduction of the minimal cover problem (see [13, 40] and compare Section 4.2). In his NP-hardness proof, the following fact is crucial.

$$\models_{\mathrm{EF}} \qquad \{f_1, \ldots, f_n\}x \wedge x[u]y \quad \to \quad \bigvee_{i=1}^{n} u = f_i$$

In order to express a corresponding disjunction in OF, we need existential quantification and, in particular, constraints of the form $\neg\{\}(y)$:

$$\models_{\mathrm{OF}} \qquad \{f_1, \ldots, f_n\}(x) \wedge x\langle y\rangle z \wedge \neg\{\}(y) \quad \to \quad \bigvee_{i=1}^{n} \exists z_i\, y[f_i]z_i$$

[b] In the sense that there is no OF constraint φ such that all solutions of φ for a fixed variable y is the set of feature trees with at least one feature (*c. f.*, [5]).

Call OF^{ne} the constraint system that is obtained from OF by addition of constraints of the form $\neg\{\}(x)$. Now we show that we can reduce the satisfiability check for EF to the one for OF^{ne}.

Proposition 5. *There is an embedding* $[\![\cdot]\!]$ *of EF into* OF^{ne} *such that every EF constraint* ψ *is satisfiable if and only if* $[\![\psi]\!]$ *is.*

Proof. We assume a special label unit which we use to represent labels f in EF by feature trees $\mathsf{unit}(f{:}\mathsf{unit})$.[c]

$$
\begin{array}{lcl}
[\![a(x)]\!] & = & a(x) \\
[\![x[f]y]\!] & = & \exists z\,\exists w\,(x[f]z \wedge z = w{\to}y \wedge \mathsf{unit}(w) \wedge \{\}(w)) \\
[\![x = y]\!] & = & x = y \\
[\![u = f]\!] & = & \exists x\,(\mathsf{unit}(u) \wedge \{f\}(u) \wedge u[f]x \wedge \mathsf{unit}(x) \wedge \{\}(x)) \\
[\![x[u]y]\!] & = & x\langle u\rangle y \wedge \neg\{\}(u) \\
[\![\{f_1,\ldots,f_n\}x]\!] & = & \{f_1,\ldots,f_n\}(x) \wedge \bigwedge_{i=1}^{n} \exists y\, x[f_i]y
\end{array}
$$

$\Rightarrow$**:** Assume a satisfiable EF constraint ψ and let α be an EF solution of ψ.

Without loss of generality, we can assume that no feature tree in the image of α contains features d and r and labels $\to$ and unit (if α does not satisfy this condition we can always rename the features and labels in the image of α to another EF solution which does, because we have assumed infinitely many features and labels; see [21] for a detailed argument to this end).

Given a feature tree τ, we define $\tau^{\uparrow}$ as the feature tree where

- The tree domain $D_{\tau^{\uparrow}}$ is the smallest prefix-closed set of path containing $\{f_1 r \cdots f_{n-1} r f_n r,\ f_1 r \cdots f_{n-1} r f_n d \mid f_1 \cdots f_n \in D_\tau\}$.
- The labeling function $L_{\tau^{\uparrow}}$ is defined by

$$
L_{\tau^{\uparrow}}(\pi) = \begin{cases} L_\tau(\varepsilon) & \text{if } \pi = \varepsilon \\ L_\tau(f_1 \cdots f_n) & \text{if } \pi = f_1 r \cdots f_{n-1} r f_n r\ (n \geq 1) \\ \mathsf{unit} & \text{if } \pi = f_1 r \cdots f_{n-1} r f_n d\ (n \geq 1) \\ \to & \text{if } \pi = f_1 r \cdots f_{n-1} r f_n\ (n \geq 1) \end{cases}
$$

It is easy to see $\tau^{\uparrow}$ is well-defined. Intuitively, $\tau^{\uparrow}$ is obtained from τ by recursively replacing all subtrees of τ of the form $a(f_1 : \tau_1, \ldots, f_n : \tau_n)$ by $a(f_1 : \mathsf{unit} \to \tau_1, \ldots, f_n : \mathsf{unit} \to \tau_n)$.

Now, we define a mapping α' from variables x and u to feature trees based on the EF solution α so that α' is an OF solution of $[\![\psi]\!]$.

$$
\begin{array}{lcll}
\alpha'(u) & = & \mathsf{unit}(f{:}\mathsf{unit}) & \text{if } \alpha(u) = f \\
\alpha'(x) & = & \alpha(x)^{\uparrow} & \text{otherwise}
\end{array}
$$

We check that α' is indeed a solution of $[\![\psi]\!]$ by case analysis.

[c] Notice that, for conciseness, we use feature variables u just like ordinary (feature tree) variables on the right hand side of the equations. Notice also, that the existential quantifiers are a matter of convenience only: Their addition does not affect the complexity of the satisfiability problem for OF.

$a(x) \in \psi$**:** It holds that $(\varepsilon, a) \in L_{\alpha(x)}$. Hence, $(\varepsilon, a) \in L_{\alpha'(x)}$ by definition of $L_{\tau^\uparrow}$.

$x[f]y \in \psi$**:** Since α is a solution for $x[f]y$, we have $\alpha(x).f = \alpha(y)$. By definition, $\alpha'(x).f = \mathsf{unit}{\to}\alpha'(y)$. Hence, α' solves $[\![x[f]y]\!]$.

$x = y \in \psi$**:** $\alpha(x) = \alpha(y)$ implies $\alpha(x)^\uparrow = \alpha(y)^\uparrow$.

$u = f \in \psi$**:** By definition, $\alpha'(u) = \mathsf{unit}(f{:}\mathsf{unit})$. Hence, α' solves $[\![u = f]\!]$.

$x[u]y \in \psi$**:** Let $f = \alpha(u)$. Since α is a solution for $x[u]y$, we have $\alpha(x).f = \alpha(y)$. By definition, $\alpha'(x).f = \mathsf{unit}{\to}\alpha'(y)$ and $\alpha'(u) = \mathsf{unit}(f{:}\mathsf{unit})$. Hence $\mathsf{ar}(\alpha'(u)) = \{f\} \neq \emptyset$, $f \in \mathsf{ar}(\alpha'(x))$ and $\alpha'(x).f = \alpha'(u).f{\to}\alpha'(y)$.

$\{f_1, \ldots, f_n\}x \in \psi$**:** $\mathsf{ar}(\tau) = \mathsf{ar}(\tau^\uparrow)$ holds for any feature tree τ by definition. Hence α' is a solution for $[\![\{f_1, \ldots, f_n\}x]\!]$.

$\Leftarrow$**:** Let ψ be an EF constraint such that $[\![\psi]\!]$ is satisfiable. We can assume that ψ does not contain any occurrence of features d and r and labels $\to$ and unit without loss of generality.

Since $[\![\psi]\!]$ is satisfiable, there is a choice, for all feature variables of a feature f_u and a fresh variable x_u such that

$$\varphi \quad =_{def} \quad [\![\psi]\!] \wedge \bigwedge_{u \in V(\psi)} u[f_u]x_u$$

is satisfiable. Notice that the f_u are usually distinct for distinct feature variables u. For instance, if ψ is $\{f_1, f_2\}x \wedge x[u]y \wedge \{g_1, g_2\}x \wedge x'[u']y'$ then $f_u \in \{f_1, f_2\}$ and $f_{u'} \in \{g_1, g_2\}$.

Let φ' be the largest (positive) OF constraint contained in φ. Apparently, all constraints of the form $\neg\{\}(u)$ in $[\![\psi]\!]$ are trivially satisfied by any solution of φ'. Hence, every solution of φ' is also a solution of φ and, thus, of $[\![\psi]\!]$.

Let $\alpha_{\varphi'}$ be solution of φ' as defined in the proof of Proposition 4. Since ψ does not contain $d, r, \to$ and unit, without loss of generality also $\alpha_{\varphi'}$ does not.

Next, we show, for all feature variables $u \in V(\psi)$, that all feature selection constraints on u in the closure of φ' mention the same feature f_u. There are two cases:

$u = f \in \psi$**:** In this case, the claim follows from satisfiability of $[\![\psi]\!]$ where, of course, $f_u = f$.

$u = f \notin \psi$**:** In this case, $x[u]y \in \psi$ holds for some x, y, since $u \in V(\psi)$. Moreover, by definition of $[\![\psi]\!]$, u occurs only in the corresponding object selection constraints in $[\![\psi]\!]$ (apart from the negated ones) and, additionally, in the feature selection $u[f_u]x_u \in \varphi'$. By inspection of the rules of Figure 2 (in particular, rule (Object Selection I)) one obtains that no selection constraints on u are added during constraint solving.

As a consequence, we conclude that $\alpha_{\varphi'}$ maps all feature variables u to a tree with the singleton arity $\{f_u\}$.

From the OF solution $\alpha_{\varphi'}$ we will now construct an EF solution α of ψ. Intuitively, for all variables x, $\alpha(x)$ will be the feature tree obtained by recursively replacing all subtrees of $\alpha_{\varphi'}(x)$ of the form $\tau' \to \tau$ by τ. Moreover, all feature variables u are mapped to the unique feature in the arity of $\alpha_{\varphi'}(u)$. Formally, for all $u, x \in V(\varphi')$:

$$\begin{array}{rcll} \alpha(u) & = & f & \text{if } \mathsf{ar}(\alpha_{\varphi'}(u)) = \{f\} \\ \alpha(x) & = & \alpha_{\varphi'}(x)^{\downarrow} & \end{array}$$

wherein $\alpha_{\varphi'}(x)^{\downarrow}$ is defined now. First, define h as the function h on paths without feature d that purges all occurrences of feature r.

$$\begin{array}{rcl} h(\varepsilon) & = & \varepsilon \\ h(\pi f) & = & \begin{cases} h(\pi) & \text{if } f = r \\ h(\pi) f & \text{if } f \neq d, r \end{cases} \end{array}$$

Given h, for all τ, define the tree domain

$$D^{\downarrow}(\tau) \; = \; \{h(\pi) \mid \pi \in D_\tau \text{ and } d \notin \pi\}$$

and the labeling

$$L^{\downarrow}(\tau) \quad = \quad \{(\pi, L_\tau(\pi')) \mid d \notin \pi',\ h(\pi') = \pi \text{ and } 2|\pi| = |\pi'|\}$$

Given these, we define

$$\alpha_{\varphi'}(x)^{\downarrow} \; = \; (D^{\downarrow}(\alpha_{\varphi'}(x)), L^{\downarrow}(\alpha_{\varphi'}(x)))$$

In general, $(D^{\downarrow}(\tau), L^{\downarrow}(\tau))$ does not define a feature tree for arbitrary τ since $h(\pi')$ in the definition of $L^{\downarrow}(\tau)$ may not work injective. However, this is the case for all paths in the image of $\alpha_{\varphi'}$ for the occurring variables.

We show, for all $x \in V(\varphi')$ and for all paths π, π' in $D_{\alpha_{\varphi'}(x)}$ with $d \notin \pi, \pi'$, that $\pi = \pi'$ holds whenever $h(\pi) = h(\pi')$ and both π and π' have even lengths. Proof is by induction on the length of $h(\pi)$.

$h(\pi) = h(\pi') = \varepsilon$: π and π' must be ε or a non-empty sequence of r. However, the latter case does not occur: ϕ' mentions d and r either by $[\![x[f]y]\!]$ or by $[\![x[u]y]\!]$ through the rule (Object Selection II). In either case, any occurrence of feature r in a path $\pi \in D_{\alpha_{\varphi'}(x)}$ is always preceded by some feature $f \neq d, r$ so that $h(\pi) = \varepsilon$ if and only if $\pi = \varepsilon$. Thus $\pi = \pi' = \varepsilon$.

$h(\pi) = h(\pi') \neq \varepsilon$: In this case there exist a path π'' and a feature $f \neq d, r$ such that $h(\pi) = f\pi''$. By a similar discussion as above, it holds that $\pi r \in D_{\alpha_{\varphi'}(x)}$ whenever $\pi \in D_{\alpha_{\varphi'}(x)}$ and the last feature in π is different from d or r. Hence, by the defintion of h, there exist paths π_0 and π'_0 such that $\pi = fr\pi_0$, $\pi' = fr\pi'_0$ and $h(\pi_0) = h(\pi'_0) = \pi''$. Since π_0 and π'_0 have even lengths, $\pi_0 = \pi'_0$ by induction hypothesis, and thus $\pi = \pi'$.

It remains to check that α is indeed a solution of $[\![\psi]\!]$ by case analysis:

$a(x) \in \psi$: It holds that $(\varepsilon, a) \in L_{\alpha_{\varphi'}(x)}$. Hence, $(\varepsilon, a) \in L_{\alpha(x)}$ by definition.

$x[f]y \in \psi$**:** By the definition of $\alpha_{\varphi'}$, we have $\alpha_{\varphi'}(x).f = \mathsf{unit}{\to}\alpha_{\varphi'}(y)$. Hence by the definition of α, $\alpha(x).f = \alpha(y)$.

$x = y \in \psi$**:** $\alpha_{\varphi'}(x) = \alpha_{\varphi'}(y)$ implies $\alpha(x)_{\varphi'}{}^{\downarrow} = \alpha(y)_{\varphi'}{}^{\downarrow}$.

$u = f \in \psi$**:** It holds that $\alpha_{\varphi'}(u) = \mathsf{unit}(f{:}\mathsf{unit})$. Hence, $\alpha(u) = f$ by definition.

$x[u]y \in \psi$**:** By the definition of $\alpha_{\varphi'}$, we can assume $\alpha_{\varphi'}(u) = \mathsf{unit}(f_u : \mathsf{unit})$. Since $\alpha_{\varphi'}$ validates $x\langle u\rangle y$, we have $f_u \in \mathsf{ar}(\alpha_{\varphi'}(x))$ and $\alpha_{\varphi'}(x).f_u = \mathsf{unit}{\to}\alpha_{\varphi'}(y)$. By the definition of α, it holds that $\alpha(x).f_u = \alpha(y)$ and $\alpha(u) = f_u$. Hence, α is a solution for $x[u]y$.

$\{f_1, \ldots, f_n\}x \in \psi$**:** By $\{f_1, \ldots, f_n\}(x) \in [\![\psi]\!]$, we have $\mathsf{ar}(\alpha_{\varphi'}(x)) \subseteq \{f_1, \ldots, f_n\}$. The inverse inclusion is by $\bigwedge_{i=1}^{n} \exists y\, x[f_i]y \in [\![\psi]\!]$. Hence, $\mathsf{ar}(\alpha_{\varphi'}(x)) = \{f_1, \ldots, f_n\}$ and also $\mathsf{ar}(\alpha(x)) = \{f_1, \ldots, f_n\}$ by definition of α.

□

Corollary 1. *The satisfiability problem for OF^{ne} is NP-complete.*

Proof. NP-hardness follows from Proposition 5 in combination with the facts that satisfiability for EF is NP-complete [40] and that $[\![\cdot]\!]$ is a polynomial-size embedding.

An NP algorithm to decide satisfiability of an OF^{ne} constraint is straightforward: Given an OF^{ne} constraint φ, make a non-deterministic choice of a feature f_u and a fresh variable v_u for every u such that $\neg\{\}(u) \in \varphi$ and check satisfiability of

$$\varphi'' \quad =_{def} \quad \varphi' \wedge \bigwedge_{\neg\{\}(u)\in\varphi} u[f_u]v_u\,.$$

This non-deterministic algorithm is correct because whenever φ is satisfiable there must be a choice that φ'' is satisfiable as well. By Theorem 1, the test for satisfiability of φ'' is polynomial in the size of φ''. Furthermore, since there are no negated selection constraints, it suffices to chose the features f_u from the finite set $F(\varphi)$. Hence, the choice is finite and the size of φ and φ'' are asymptotically the same. Hence, the algorithm takes non-deterministic polynomial time. □

Corollary 2. *The satisfiability problem of every extension of OF that can express $\neg\{\}(x)$ is NP-hard.*

For example, the satisfiability problem of positive and negative OF constraints is NP-hard. The precise complexity of OF constraints with negation is left open.

2.5. Additional Simplification Rules

This section shortly discusses some constraint simplification rules that are not necessary for the satisfiability check but are worth considering for other reasons.

The following two additional rules are justified by implications (3) and (4):

$$\frac{\varphi \wedge x\langle y\rangle z \wedge x\langle y\rangle z'}{\varphi \wedge x\langle y\rangle z \wedge z = z'} \qquad \text{if } y[f]y' \in \varphi \qquad \text{(Double Object Selection)}$$

$$\frac{\varphi \wedge x\langle y\rangle z}{\varphi} \qquad \text{if } \{\}(y) \in \varphi \qquad \text{(Feature-less Selector)}$$

The rule (Double Object Selection) is derived from (Object Selection I/II) and can be used to speed up the satisfiability test when given priority over rule (Object Selection II). In contrast, the rule (Feature-less Selector) is not a derived one; it can be used to reduce the size of a constraint and therefore may save space and time.

The following rule allows the arity bound to be propagated through object selection.

$$\frac{\varphi \wedge x\langle y\rangle z}{\varphi \wedge x\langle y\rangle z \wedge F(x)} \qquad \text{if } F(x) \in \varphi \qquad \text{(Arity Propagation)}$$

This rule is justified by the following implication valid in OF:

$$\models_{\text{OF}} \qquad x\langle y\rangle z \wedge F(x) \quad \rightarrow \quad F(y)$$

This rule makes explicit arity constraints that are mediated through selection constraints $x\langle y\rangle z$. By so propagating arity constraints and by normalizing them with (Arity Intersection) one obtains a normal form that allows one to read off the smallest implied arity bound per variable. This rule appears useful to make type information easily accessible: The set of possible message names for every bound message identifier is directly represented by the arity bound on its type. In Section 4.3, we adopt this rule to determine identifiers with empty message type. Note that arity propagation can be incorporated into the satisfiability check without affecting polynomial complexity [24].

3. Type Inference

In this section, we reformulate the type inference of Nishimura [25] in terms of OF constraints.

Let us consider a tiny object-oriented programming language whose abstract syntax is defined as follows.

$$\begin{array}{lcll}
M & ::= & b & \text{(Constant)} \\
& \mid & \mathsf{x} & \text{(Variable)} \\
& \mid & f(M) & \text{(Message)} \\
& \mid & \{f_1(\mathsf{x}_1) = M_1, \ldots, f_n(\mathsf{x}_n) = M_n\} & \text{(Object)} \\
& \mid & M \leftarrow N & \text{(Message Passing)} \\
& \mid & \mathsf{let\ y} = M \ \mathsf{in}\ N & \text{(Let Binding)}
\end{array}$$

$$\frac{\mathsf{x} : t \in \Gamma}{\varphi, \Gamma \vdash \mathsf{x} : t} \textsc{ Var} \qquad \frac{}{\varphi, \Gamma \vdash b : \mathsf{typeof}(b)} \textsc{ Const}$$

$$\frac{\varphi, \Gamma \vdash M : t' \quad \varphi \models_{\text{OF}} t :: \mathsf{msg}(f{:}t')}{\varphi, \Gamma \vdash f(M) : t} \textsc{ Msg}$$

$$\frac{\varphi, \Gamma; \mathsf{x}_i : t_i \vdash M_i : t_i' \quad \text{for every } i = 1, \ldots, n}{\varphi, \Gamma \vdash \{f_1(\mathsf{x}_1) = M_1, \ldots, f_n(\mathsf{x}_n) = M_n\} : \mathsf{obj}(f_1{:}t_1 {\to} t_1', \ldots, f_n{:}t_n {\to} t_n')} \textsc{ Obj}$$

$$\frac{\varphi, \Gamma \vdash M : t_1 \quad \varphi, \Gamma \vdash N : t_2 \quad \varphi \models_{\text{OF}} \mathsf{obj}(t_1) \wedge \mathsf{msg}(t_2) \wedge t_1 \langle t_2 \rangle t_3}{\varphi, \Gamma \vdash M \leftarrow N : t_3} \textsc{ MsgPass}$$

$$\frac{\varphi, \Gamma; \mathsf{y} : t_1 \vdash M : t_1 \quad \varphi, \Gamma; \mathsf{y} : t_1 \vdash N : t_2}{\varphi, \Gamma \vdash \mathsf{let}\ \mathsf{y} = M\ \mathsf{in}\ N : t_2} \textsc{ Let} \text{ (monomorphic)}$$

Figure 3: The monomorphic type system for first-class messages

The language syntax is simplified over [25] by dropping letobj altogether; it should be understood that the let expression allows recursive definition for a certain relevant class of expressions, as letobj in [25] allows recursive definition only for objects.

The operational semantics is defined along the line of [25]. We do not repeat it here since it is just the intuitive call-by-value semantics adapted to an object-oriented language.

For the types, we assume additional distinct labels msg and obj to mark message and object types, and a set of distinct labels such as int, bool, *etc.*, to mark base types. Monomorphic *types* are certain feature trees over this signature, and monomorphic *type terms* are the corresponding feature terms. Type terms obey the following abstract syntax:

t	::=	α	(Type variable)
	$\mid$	$\mathsf{int} \quad \mid \quad \mathsf{bool} \quad \mid \quad \ldots$	(Base type)
	$\mid$	$\mathsf{msg}(f_1{:}t_1, \ldots, f_n{:}t_n)$	(Message type)
	$\mid$	$\mathsf{obj}(f_1{:}t_1 {\to} t_1', \ldots, f_n{:}t_n {\to} t_n')$	(Object type)

3.1. Monomorphic Type System and Type Inference

We assume a mapping typeof from constants of base type to their corresponding types, for instance $\mathsf{typeof}(1) = \mathsf{typeof}(2) = \ldots = \mathsf{int}$ and $\mathsf{typeof}(\mathsf{true}) = \mathsf{typeof}(\mathsf{false}) = \mathsf{bool}$. We also use the *kinding* notation $t :: a(f_1{:}t_1, \ldots, f_n{:}t_n)$ to state that t denotes a feature tree with underspecified arity containing the features $\{f_1, \ldots, f_n\}$ and corresponding subtrees; for example, $t :: a(f_1{:}t_1, \ldots, f_n{:}t_n)$ is equivalent to $a(t) \wedge \bigwedge_{i=1}^{n} t[f_i]t_i$.

The monomorphic type system is given in Figure 3. As usual, a *type environment* Γ is a finite mapping from variables x to type terms t, and $\Gamma; \mathsf{x} : t$ extends Γ so that it maps variable x to t. The type system defines judgments such as $\varphi, \Gamma \vdash M : t$ which reads as

$$
\begin{array}{lcll}
I(x,\Gamma,b) & = & a(x) \wedge \{\}(x) & \text{if } a = \mathsf{typeof}(b) \\
I(x,\Gamma,\mathsf{y}) & = & x = \Gamma(\mathsf{y}) & \\
I(x,\Gamma,f(M)) & = & \exists y\,(\mathsf{msg}(x) \wedge x[f]y \wedge I(y,\Gamma,M)) & \\
\multicolumn{4}{l}{I(x,\Gamma,\{f_1(\mathsf{x}_1) = M_1,\ldots,f_n(\mathsf{x}_n) = M_n\})} \\
\multicolumn{4}{l}{\quad = \quad \mathsf{obj}(x) \wedge \{f_1,\ldots,f_n\}(x) \wedge \bigwedge_{i=1}^{n} \exists x_i\, \exists x'\, \exists z\, (x[f_i]x' \wedge x' = x_i \to z \wedge I(z,\Gamma;\mathsf{x}_i : x_i, M_i))} \\
I(x,\Gamma,M \leftarrow N) & = & \exists y\, \exists z\, (y\langle z\rangle x \wedge \mathsf{obj}(y) \wedge I(y,\Gamma,M) \wedge \mathsf{msg}(z) \wedge I(z,\Gamma,N)) & \\
I(x,\mathsf{let}\ \mathsf{y} = M\ \mathsf{in}\ N) & = & \exists y\,(I(y,\Gamma;\mathsf{y}:y,M) \wedge I(x,\Gamma;\mathsf{y}:y,N)) &
\end{array}
$$

Figure 4: Monomorphic type inference for first-class messages with OF constraints

"under the type assumptions in Γ subject to the constraint φ, the expression M has type t";[d] the constraint φ in well-formed judgments is required to be satisfiable. We do not comment further on the type system here but refer to [25] for intuitions and to [26, 39] for notation.

Notice that terms are, as usual, finite entities that do, however, denote infinite feature trees. That means the type system of Figure 3 can deal with recursive types without the need for an explicit μ notation as commonly used (*e.g.*, see [4]). Recursive types are necessary for the analysis of recursive objects.

The corresponding type inference is given in Figure 4 as a mapping I from a variable x, a type environment Γ, and a program expressions M to an OF constraint such that every solution of x in $I(x,\Gamma,M)$ is a type of M under the type assumptions in Γ. For ease of reading, we reuse the bound variables in program expressions as their associated type variables.

Correctness of the type inference with respect to the type system is obvious. Soundness of the type system (with respect to the assumed operational semantics) can be shown along the line given in [25].

Theorem 2. *Type inference for first-class messages is polynomial in time and space.*

Proof. The type inference generates an existential formula Φ over OF constraints whose size is proportional to the size of the given program expression. From Proposition 3 we know that satisfiability of Φ can be decided in polynomial time and space. Finally, it is easy to show that every OF-formula $I(x,\Gamma,M)$ that is satisfiable over arbitrary feature trees is already satisfiable over the smaller domain of types. □

3.2. Polymorphic Type Inference

We can obtain the polymorphic type inference by applying the scheme HM(X) [26]. The constraint system OF is a viable parameter for HM(X) since it satisfies the two required properties, called coherence and soundness. Both assume a notion of monomorphic types

[d]This terminology is slightly sloppy but common: Since t may contain type variables it is rather a type *term* than a type and it would be accurate to say that M has "some type matching t".

and a (subtyping) order on them. In our case, these are given by feature trees and equality on them; it does no harm that our monomorphic types may be infinite. The *coherence* property requires that the considered order on types is semantically well-behaved and holds; for equality, this condition becomes trivial. The *soundness* property that a solved constraint indeed has a solution follows from Proposition 4.

3.3. Examples

Let us consider some typing examples in the monomorphic type system. In the subsequent discussions, we will freely use the compact feature term notation of OF constraints.

Remark. In general, type inference requires that constraints representing a type must be compactly presented in order to make them easily digestible by programmers. The use of a term notation is crucial here, even though it is not during type inference. But, as the OCaml [35] experience shows, terms do not suffice. In OCaml an additional abbreviation mechanism for object types is provided which usually grow rather large. Corresponding mechanisms seem to be in place when putting our system into practice.

As a first example, the statement

$$\mathsf{let\ o1} = \{\textbf{succ}(\mathsf{x}){=}\mathsf{x}{+}1,\ \textbf{pos}(\mathsf{x}){=}\mathsf{x}{>}0\};$$

defines an object with two methods **succ** : $int{\to}\mathsf{int}$ and **pos** : $int{\to}\mathsf{bool}$. Type inference gives the type of this object as an OF constraint on the type variable o_1 equivalent to

$$\varphi_1 \quad =_{def} \quad o_1 = \mathsf{obj}(\textbf{succ} : int{\to}int, \textbf{pos} : int{\to}\mathsf{bool}).$$

A delegate object for the object o1 is defined as follows:

$$\mathsf{let\ o2} = \{\textbf{redirect}(\mathsf{m}){=}\ \mathsf{o1} \leftarrow \mathsf{m}\};$$

where m is a parameter that binds messages to be redirected to o1. Assuming the variable o_1 to be constrained by φ_1, the constraint φ_2 restricts o_2 to the type of o2:

$$\varphi_2 \quad =_{def} \quad \exists m\, \exists z\, (o_2 = \mathsf{obj}(\textbf{redirect} : m{\to}z) \wedge o_1\langle m\rangle z \wedge \mathsf{msg}(m)).$$

The return type of a message passing to this object, for instance as in

$$\mathsf{let\ w} = \mathsf{o2}{\leftarrow}\ \textbf{redirect}(\textbf{succ}(1));$$

is described as the solution of $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ for the type variable w, where

$$\varphi_3 \quad =_{def} \quad \exists z'\, (o_2\langle z'\rangle w \wedge z' :: \mathsf{msg}(\textbf{redirect} : \mathsf{msg}(\textbf{succ} : \mathsf{int}))),$$

The solved form of $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ contains the constraint $\mathsf{int}(w) \wedge \{\}(w)$, which represents the intended result type int.

If o1 does not respond to the message argument of **redirect**, for instance as in

$$\mathsf{let\ v} = \mathsf{o2}{\leftarrow}\ \textbf{redirect}(\textbf{pred}(1));$$

a type error is detected as inconsistency in the derived constraint. Here, the constraint

$$\varphi_4 \quad =_{def} \quad \exists z' \ (o_2\langle z'\rangle w' \wedge z' :: \mathsf{msg}(\textbf{redirect} : \mathsf{msg}(\textbf{pred} : \mathsf{int})))$$

implies $\exists z' \ (o_1\langle z'\rangle w' \wedge z' :: \mathsf{msg}(\textbf{pred} : \mathsf{int}))$, and hence that o_1 has a feature **pred** which contradicts φ_1 by (Arity Clash).

4. Empty Message Types

In Section 2.1.3, we have seen that the OF first-order selection constraint $x\langle y\rangle z$ is not functional, *i. e.*, the implication $x\langle y\rangle z \wedge x\langle y\rangle z' \rightarrow z = z'$ does *not* hold because y may denote a tree without any features. In terms of typing, this means that $M \leftarrow N$ may be well-typed even if N has the *empty message type* msg, *i. e.*, the message type represented by a feature tree without any feature. The empty message type does not mention any message names or argument types as possible types for the expression N. Hence the empty message type is given to an expression that is syntactically used as a message but will not to any message at run-time.

We consider this phenomenon more closely which may be called an undesirable property of our type system. However, we also show that a straightforward fix of this problem makes type inference NP-complete. This illustrates our conviction that empty message types are the price to pay for a polynomial type inference of typing first-class messages.

4.1. Empty Message Types are Weird

Consider the following well-typed program.

```
let o1 = {a(x)=x+1, b(x)=x>0} in
let o2 = {b(x)=x=0,c(x)=x*2} in
let o3 = {foo(m)= begin o1← m; o2←m end};
```

It is easy to see that every successful execution of the body of the method **foo** must return bool: The argument message m of **foo** must be accepted by both the objects o1 and o2, which share only the method **b** of type int → bool.

However, the body of method **foo** is not necessarily executed at all in which case the return type is irrelevant. Type inference reflects this effect by deriving from this program (essentially) the following constraint:

$$\begin{array}{l} o_1 = \mathsf{obj}(\textbf{a} : \mathsf{int}{\rightarrow}\mathsf{int}, \textbf{b} : \mathsf{int}{\rightarrow}\mathsf{bool}) \quad \wedge \\ o_2 = \mathsf{obj}(\textbf{b} : \mathsf{int}{\rightarrow}\mathsf{bool}, \textbf{c} : \mathsf{int}{\rightarrow}\mathsf{int}) \quad \wedge \\ o_3 = \mathsf{obj}(\textbf{foo} : m{\rightarrow}z) \wedge o_1\langle m\rangle z_1 \wedge o_2\langle m\rangle z_2 \end{array}$$

Notice that $z = \mathsf{bool}$ is not entailed! Also, the type of the message passings o1←m and o2←m need not coincide with the return type of **foo**: Neither $z = z_1$ nor $z = z_2$ is entailed.

By a similar argument, the following program can be considered acceptable even though the method **foo** cannot be executed at all without failure:

```
let o1 = {a(x)=x+1} in
let o2 = {c(x)=x*2} in
let o3 = {foo(m)= begin o1←m; o2←m end}
```

$$\frac{\varphi,\Gamma\vdash M:t_1 \quad \varphi,\Gamma\vdash N:t_2 \quad \varphi\models_{\mathrm{OF}} \mathsf{obj}(t_1)\wedge\mathsf{msg}(t_2)\wedge\neg\{\}(t_2)\wedge t_1\langle t_2\rangle t_3}{\varphi,\Gamma\vdash M\leftarrow N:t_3}\ \textsc{MsgPass}'$$

Figure 5: The typing rule for message passing when empty messages are excluded

These examples may be surprising, since a program is well-typed even though it may contain statements for which there is no effective execution. In this respect, our type system is weaker than that of [25]. The weakness is apparently due to the admission of empty message types where type inference stops and leaves possibilities of further analysis unexploited.

Nonetheless, there is a strong rationale that we say our type system is a relevant one: It is still *sound* in the sense that execution of a well-typed program is type safe. Type safety is guaranteed since, when an identifier has the empty message type, it is never bound to a message at run-time.

4.2. Type Inference is NP-complete if Empty Messages are Excluded

One may insist that method invocation by empty messages should be detected as a type error. In this case, it is easy to manipulate the type system and the type inference to ensure this: One just needs to disallow the empty message types using negation.

The only typing rule affected by this restriction is MsgPass which changes to MsgPass' as shown in Figure 5. The corresponding clause for type inference is this one:

$$I(x, M\leftarrow N) \quad = \quad \exists y\,\exists z\,(y\langle z\rangle x\wedge\mathsf{obj}(y)\wedge I(y,M)\wedge\mathsf{msg}(z)\wedge\neg\{\}(z)\wedge I(z,N)) \tag{5}$$

However, recall that the polynomial time complexity of the analysis depends on the above-mentioned weakness. Type inference for the type system with the rule MsgPass' instead of MsgPass would be NP-complete, since the general satisfiability problem OF with negative constraints $\neg\{\}(x)$ is NP-complete (Corollary 1 of Proposition 5).

To prove NP-completeness, the close correspondence between OF with negation and EF helps us again: Treinen reduced Minimal Cover Problem [13] to the satisfiability of EF [40]. Following Treinen, we give an encoding of the Minimal Cover Problem to the type inference problem for first-class messages where the empty message type is disallowed (*i. e.*, we consider the type system given by Figure 3 and the rule MsgPass replaced by MsgPass').

The Minimum Cover Problem (MCP) is defined as follows:

> Given a collection $S_1,\ldots,S_n$ of finite sets and a natural number $k\leq n$, is there a subset $I\subseteq\{1,\ldots,n\}$ whose cardinality is at most k such that $\bigcup_{j\in I} S_j = \bigcup_{i=1}^{n} S_i$?

Since the MCP is known to be NP-complete and the reduction is polynomial, this proves that type inference problem to be NP-hard.

The adaptation of Treinen's reduction is an immediate one and given here for completeness' sake. For proofs, we refer the reader to Treinen's exposition [40].

4.2.1. The Encoding

We assume that an instance $S_1,\ldots,S_n,k$ of the MCP is given. We define the set U to be covered, $U =_{def} \bigcup_{i=1}^{n} S_i$, and, for every $u \in U$, the set δ_u of indexes j of those sets in which u occurs: $\delta_u = \{j \mid u \in S_j\}$. Without loss of generality, we assume that $\mathbf{1},\ldots,\mathbf{n} \in \mathcal{F}$.

Following Treinen, we construct a program that is well-typed if and only if the given instance of the MCP has a solution. We use variables x_u to represent the elements $u \in U$ and variables $z_1,\ldots,z_n$ to represent the sets $S_1,\ldots,S_n$.

In order to stay close to Treinen's encoding in syntax, five schematic statements are used.

- The first statement introduces an identifier of some type and, thus, simulates an existential quantifier.

$$\exists \mathsf{x}\, M \qquad =_{def} \qquad \{\ \mathbf{foo}(\mathsf{x}) = M\ \}$$

- The second statement forces x and y to have the same type:

$$\mathsf{x} \sim \mathsf{y} \qquad =_{def} \qquad \exists \mathsf{f}\ (\ \mathsf{f} \leftarrow \mathbf{bar}(\mathsf{x});\ \mathsf{f} \leftarrow \mathbf{bar}(\mathsf{y})\ )$$

- The third statement says that an object x has *exactly* methods labeled $\{f_1,...,f_n\}$:

$$\{f_1,...,f_n\}\mathsf{x} \qquad =_{def} \qquad \exists \mathsf{y}_1,\ldots,\exists \mathsf{y}_n\ (\ \mathsf{x} \sim \{f_1(\mathsf{z}_1) = \mathsf{y}_1,...,f_n(\mathsf{z}_n) = \mathsf{y}_n\})$$

- The last two statements say that x is labeled by obj (*resp.*, msg).

$$\mathsf{IN}\ \mathsf{x} \qquad =_{def} \qquad \exists \mathsf{y}\ (\mathsf{x} \leftarrow \mathsf{y})$$

$$\mathsf{OUT}\ \mathsf{x} \qquad =_{def} \qquad \exists \mathsf{y}\ (\mathsf{y} \leftarrow \mathsf{x})$$

  The syntax of these statements is motivated by Treinen's encoding, whose intention will become clear below.

Furthermore, conjunctive notation $\bigwedge_{i=1}^{n} M_i$ means the corresponding sequence of statements $M_1;\ldots;M_n$.

The program M that we construct is a sequence of three program expressions:

$$M = M_1; M_2; M_3$$

Well-typedness of the first program M_1 requires that x_u has an object type whose set of method labels coincides with δ_u, and that the type of z_j is the return type of the method j of x_u if and only if $u \in S_j$.

$$M_1 \quad =_{def} \quad \bigwedge_{u \in U} \delta_u \mathsf{x}_u; \quad \bigwedge_{u \in U} \bigwedge_{j \in \delta_u} \exists \mathsf{z}\ ((\mathsf{x}_u \leftarrow j(\mathsf{z})) \sim \mathsf{z}_j)$$

The choice of an appropriate set I is now expressed by labeling on the variables z_i. The idea, as in Treinen's reduction, is to enforce one of two constraints on every z_i: $\mathsf{IN}\ \mathsf{z}_i$ (that expresses z_i is a member of, *i. e.*, *in* the minimum cover) if $i \in I$ and $\mathsf{OUT}\ \mathsf{z}_i$ (z_i is not a member of,*i. e.*, *out* of the minimum cover) otherwise. Intuitively, this encoding works because at most a single label is allowed on the same node.

Well-typedness of the second program M_2 implies the fact that for at least $n-k$ of the z_i it holds that $\mathsf{OUT}\ \mathsf{z}_i$.

$$M_2 \quad =_{def} \quad \exists \mathsf{x}\ (\{\mathbf{1},\ldots,\mathbf{n}\}\mathsf{x};\quad \bigwedge_{i\in\{\mathbf{1},\ldots,\mathbf{n}\}} \exists \mathsf{z}\ (\mathsf{x} \leftarrow i(\mathsf{z})) \sim \mathsf{z}_i; \\ \bigwedge_{i\in\{\mathbf{1},\ldots,\mathbf{n-k}\}} \exists \mathsf{v}\ \exists \mathsf{y}\ ((\mathsf{x} \leftarrow \mathsf{v}) \sim \mathsf{y};\quad \mathsf{OUT}\ \mathsf{y};\quad \{i\}\mathsf{y}))$$

The statement $\{i\}\mathsf{y}$ forces for each i a different type of y.

Well-typedness of the statement M_3 requires that each x_i has a method whose return type, according to the definition of M_1, must be one of the z_i and also it holds that $\mathsf{IN}\ \mathsf{z}_i$.

$$M_3 \quad =_{def} \quad \bigwedge_{i\in U} \exists \mathsf{v}\ \exists \mathsf{z}\ ((\mathsf{x}_i \leftarrow \mathsf{v}) \sim \mathsf{z};\quad \mathsf{IN}\ \mathsf{z})$$

We notice that our encoding implements the EF labeling constraints $\mathsf{IN}\ \mathsf{x}$ and $\mathsf{OUT}\ \mathsf{x}$ in Treinen's original encoding by OF labels obj and msg, respectively. We need this translation, since the type inference cannot enforce arbitrary labeling constraint. Our encoding, however, preserves the intended function of Treinen's encoding that separates the given sets into two disjoint classes.

The length of the statement M is in fact linear in the size of the representation of the MCP. Hence, we obtain

Theorem 3. *Type checking and type inference for first-class messages is NP-hard when the empty message type is disallowed.*

Proof. See the proof of Theorem 4 in [40]. □

Combining this theorem and Corollary 1 of Proposition 5, we conclude that

Corollary 3. *Type checking and type inference for first-class messages is NP-complete when the empty message type is disallowed.*

4.3. Discussion

The immediate question arising Theorem **??** is this: *Is there any polynomial time type inference algorithm for first-class messages that prohibits empty message types?*

According to the discussion so far, there is no such algorithm – we believe that the problem is inherently NP-complete. Of course, there might be an entirely different approach to typing first-class messages that would give rise to such an algorithm. We must leave the problem open. Instead, we suggest two pragmatic ways of having your cake (no empty messages) and eating it, too (reasonably efficient type inference).

We could just ignore NP-completeness and use negation to disallow empty message types as sketched. If the number of message labels in a program is significantly smaller than the size of the program, then the enumeration of labels might be tolerable. Exponential behaviour might simply not show up.

One could also require the programmer to provide at least one witness label for every message identifier in the program. This indirectly avoids empty message types without the need for negation in type inference. In practice, the compiler would complain about every message identifier for which no witness label were explicit in the program. To overcome this complaint, a type annotation would be needed that could, admittedly, restrict polymorphism. Polynomial type inference would be achieved by passing the obligation of providing witness features from the compiler (search) to the programmer.

4.4. Comparison with Nishimura's System

In Nishimura's original type system [25], referred to as $\mathcal{D}$ in the following, constraints are modeled as kinded type variables. The kindings have a straightforward syntactic correspondence with OF constraints: the message kinding $x :: \langle\langle f_1{:}t_1, \ldots, f_n{:}t_n\rangle\rangle_F$ corresponds to $x :: \mathsf{msg}(f_1{:}t_1, \ldots, f_n{:}t_n) \wedge F(x)$ and the object kinding $x :: \{\!|y_1{\to}t_1, \ldots, y_n{\to}t_n|\!\}_F$ corresponds to $\mathsf{obj}(x) \wedge \bigwedge_{i=1}^{n} x\langle y_i\rangle t_i \wedge F(x)$.

Our reformulation HM(OF) of $\mathcal{D}$ is in the same spirit as the reformulation HM(REC) [26] of Ohori's type system for the polymorphic record calculus. One might thus expect the relation of $\mathcal{D}$ and HM(OF) to be as close as that between Ohori's system and HM(REC) which type exactly the same programs ("full and faithful"); this is, however, not the case.

There is a significant difference between the kind system in $\mathcal{D}$ and OF. In $\mathcal{D}$, (kinded) types may contain variables: For instance, an object returning integers as a response to messages of type y receives the type kinded by $\{\!|y{\to}\mathsf{int}|\!\}_F$. On unifying two kinds $\{\!|y{\to}\mathsf{int}|\!\}_F$ and $\{\!|y{\to}z|\!\}_F$, the type inference for $\mathcal{D}$ derives equality of z and int since it is *syntactically* known that both z and int denote the type of the response of the same object to the same message. Thus in $\mathcal{D}$, the name of type variables is crucial. In this paper, variables only occur as part of type descriptions (*i. e.*, syntax) while the (semantic) domain of types does not contain variables. That is, we understand $\{\!|y{\to}\mathsf{int}|\!\}$ not as a *type* but as part of a *type description* which can be expressed by a constraint like $\mathsf{obj}(x) \wedge x\langle y\rangle\mathsf{int}$.

As a consequence, well-typedness in our system does not depend on the choice of variable names but only on the type of variables. This is usual for ML-style type systems but does not hold for $\mathcal{D}$. Consider the following example:

$\{\mathbf{foo}(\mathsf{m}) = (\mathsf{o}{\leftarrow}\mathsf{m}) + 1;\ (\mathsf{o}{\leftarrow}\mathsf{m})\ \&\ \mathsf{true}\}$

This program is accepted by the OF-based type system, since the constraint $o\langle m\rangle\mathsf{int} \wedge o\langle m\rangle\mathsf{bool}$ is satisfiable with m as the empty message. The type system $\mathcal{D}$, however, rejects that program after trying to unify int and bool during type inference.

The following example shows why this syntactic argument may be confusing. System $\mathcal{D}$ accepts the program

$\{\mathbf{bar}(\mathsf{m}) = (\{\}{\leftarrow}\mathsf{m}) + 1;\ (\{\}{\leftarrow}\mathsf{m})\ \&\ \mathsf{true}\}$

but rejects the equivalent one

```
let o={} in {foo(m) = (o←m) + 1; (o←m) & true};
```

As a final difference between $\mathcal{D}$ and our modified type system notice that $\mathcal{D}$ accepts sending messages to an empty object such as

```
{bar(m)={}← m}
```

whereas our system does not accept this program.

5. Conclusion

We have presented a new constraint system OF over feature trees and investigated the complexity of its satisfiability problem. OF is designed for specification and implementation of type inference for first-class messages in the spirit of Nishimura's system [25]. We have given a type system for which monomorphic type inference with OF constraints can be done in polynomial time; this system is weaker than the original one, but, as we have shown, the additional expressiveness would have rendered monomorphic type inference NP-complete. Given OF, we can add ML-style polymorphism by instantiating the recent HM(X) scheme to the constraint system OF.

OF developed from the practical problem of understanding better a given type system and its type inference problem. Although it turned out very fruitful to define OF as a member of the family of feature constraint systems, we do not consider OF to be a very natural such member from a predicate logical point of view: The semantics of $x\langle y\rangle z$ is application-specific, fairly complex, and signature-dependent.

More fundamental, seems to be another relative of OF: Assume, in addition to the feature tree variables x, y, z a class of variables ranging over sets of features, with typical members u, v and define the class of constraints

$$\begin{array}{lllllllllll} \varphi & ::= & x = y & \mid & x[f]y & \mid & F(x) & \mid & a(x) & \mid & \\ & & u = v & \mid & f \in u & \mid & x[u]y & \mid & F(u) & \mid & \varphi \wedge \varphi' \end{array}$$

with the now obvious semantics. This system is an extension of EF as well, and it is not signature-dependent as OF is. It can easily be embedded into OF by representing sets of features $\{f_1, \ldots, f_n\}$ by feature trees with the corresponding arity, say $\mathsf{set}(f_1{:}\mathsf{unit}, \ldots, f_n{:}\mathsf{unit})$, and all complexity results carry over. It appears as if these constraints could be useful in type inference for a system of record types with first-class labels as alluded to in the introduction. This is left to further investigation, however.

In another line of research, it could be interesting to make precise the relationship between kind based analysis of types and solving feature based constraints. In particular, Ohori's polymorphic record type [28] seems to be closely related to CFT [38].

From the application point of view, constraints are a useful guide for providing type information in a succinct presentation. In recent studies [11, 32], constraints are a central tool of simplifying verbose type information and to assist the programmer to detect the source of type errors. As touched upon in Section 3.3, OF constraints alone are not sufficient for this purpose. This issue is beyond the subject of the present paper, but the experience of OCaml [35] is likely to be relevant here.

Acknowledgments

We would like to thank the members of RIMS, Andreas Rossberg and Joachim Walser for careful proofreading and feedback, as well as Martin Sulzmann for extensive discussion on HM(X). Thanks are also due to Ralf Treinen as well for providing the system EF that turned out to fit our needs so nicely. We also acknowledge helpful remarks of the referees for ASIAN98 and IJFCS.

References

1. A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture*, pp. 31–41. ACM Press, New York, June 1993.
2. H. Aït-Kaci and A. Podelski. Towards a meaning of life. *The Journal of Logic Programming*, 16(3–4):195–234, July, Aug. 1993.
3. H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, Jan. 1994.
4. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
5. R. Backofen. *Expressivity and Decidability of First-order Languages over Feature Trees*. Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, D–66041 Saarbrücken, 1994.
6. R. Backofen. A complete axiomatization of a theory with feature and arity constraints. *The Journal of Logic Programming*, 24(1–2):37–71, 1995. Special Issue on Computational Linguistics and Logic Programming.
7. R. Backofen and G. Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268, July 1995.
8. F. Bourdoncle and S. Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 302–315. ACM Press, New York, Jan. 1997.
9. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer Auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):738–761, Aug. 1994.
10. J. Dörre. *Feature-Logik und Semiunfikation*. Doctoral Dissertation. Philosophische Fakultät der Universität Stuttgart, July 1993. In German.
11. D. Duggan. Correct type explanation. In *Proceedings of ACM SIGPLAN Workshop on ML*, pp. 49–57, 1998.
12. J. Eifrig, S. Smith, and V. Trifonow. Type inference for recursively constrained types and its application to object-oriented programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.
13. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
14. B. R. Gaster. *Records, Variants and Qualified Types*. PhD thesis, University of Nottingham, 1998.
15. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.
16. P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pp. 444–478. The MIT Press,

Cambridge, MA, 1991.

17. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the Association for Computing Machinery*, 41(2):368–398, Mar. 1994.
18. K. Mehlhorn and P. Tsakalides. Data structures. In van Leeuwen [41], chapter 6, pp. 301–342.
19. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.
20. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.
21. M. Müller and J. Niehren. Ordering constraints over feature trees expressed in second-order monadic logic. In T. Nipkow, ed., *International Conference on Rewriting Techniques and Applications*, vol. 1379 of *Lecture Notes in Computer Science*, pp. 196–210. Springer-Verlag, Berlin, 1998. Full version to appear in special issue of Information and Computation on RTA'98.
22. M. Müller, J. Niehren, and A. Podelski. Inclusion constraints over non-empty sets of trees. In M. Bidoit and M. Dauchet, eds., *Proceedings of the Theory and Practice of Software Development*, vol. 1214 of *Lecture Notes in Computer Science*, pp. 345–356. Springer-Verlag, Berlin, Apr. 1997.
23. M. Müller, J. Niehren, and A. Podelski. Ordering constraints over feature trees. In G. Smolka, ed., *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, vol. 1330 of *Lecture Notes in Computer Science*, pp. 297–311. Springer-Verlag, Berlin, 1997. Full version to appear in special issue on CP'97 of Constraints – An International Journal.
24. M. Müller and S. Nishimura. Type inference for first-class messages with feature constraints. In *Proceedings of the 4th Asian Computing Science Conference*, vol. 1538 of *Lecture Notes in Computer Science*, pp. 169–187, Dec. 1998.
25. S. Nishimura. Static typing for dynamic messages. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pp. 266–278. ACM Press, New York, 1998.
26. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
27. M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proceedings of the 7th ACM Conference on Functional Programming and Computer Architecture*, pp. 135–146. ACM Press, New York, 1995.
28. A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.
29. J. Palsberg. Efficient inference of object types. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pp. 186–185. IEEE Computer Society Press, 1994.
30. S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the programming language Haskell 98: A non-strict, purely functional language. Technical report, Feb. 1999. Available at `http://www.haskell.org/definition/`.
31. C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. Cambridge University Press, Cambridge, England, 1994.
32. F. Pottier. A framework for type inference with subtyping. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pp. 228–238. ACM Press, New York, Sept. 1998.
33. Programming Systems Lab. The MOzart Programming System, 1999. Universität des Saar-

landes: `http://www.mozart-oz.org/`.

34. D. Rémy. Type checking records and variants in a natural extension of ML. In *Proceedings of the* 16th *ACM Symposium on Principles of Programming Languages*, pp. 77–87. ACM Press, New York, 1989.
35. D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.
36. W. C. Rounds. Feature logics. In J. v. Benthem and A. ter Meulen, eds., *Handbook of Logic and Language*, pp. 475–533. Elsevier Science Publishers B.V. (North Holland), 1997. Part 2: General Topics.
37. G. Smolka. The Oz Programming Model. In J. van Leeuwen, ed., *Computer Science Today*, vol. 1000 of *Lecture Notes in Computer Science*, pp. 324–343. Springer-Verlag, Berlin, 1995.
38. G. Smolka and R. Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.
39. M. Sulzmann. Proofs of properties about HM(X). Technical Report YALEU/DCS/RR-1102, Yale University, 1998.
40. R. Treinen. Feature constraints with first-class features. In A. M. Borzyszkowski and S. Sokołowski, eds., *International Symposium on Mathematical Foundations of Computer Science*, vol. 711 of *Lecture Notes in Computer Science*, pp. 734–743. Springer-Verlag, Berlin, 30 August–3 September 1993.
41. J. van Leeuwen, ed. *Handbook of Theoretical Computer Science*, vol. A (Algorithms and Complexity). The MIT Press, Cambridge, MA, 1990.
42. M. Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pp. 37–44. IEEE Computer Society Press, 1987. Corrigendum in LICS ’88, p. 132.
43. M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.