Type Inference for First-Class Messages with Feature Constraints

Martin Müller[1] and Susumu Nishimura[2]

[1] Universität des Saarlandes, 66041 Saarbrücken, Germany
mmueller@ps.uni-sb.de
[2] RIMS, Kyoto University, Sakyo-ku, Kyoto 606-8502, Japan
nisimura@kurims.kyoto-u.ac.jp

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, 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 in an important special case. We compare OF with Treinen's constraint 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. 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 that is desirable but makes type inference NP-hard.
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 refers to the computation triggered by the corresponding method call, while a functional argument represents the computation executed on application. For example, a **map** method can be defined by means of first-class messages as follows

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

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

First-class messages are more common and crucial in distributed object-oriented programming. 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. The following delegate object defines simple proxy server:

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

This creates an object ProxyServer with a method **new** that receives an object o. The method returns a second 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

let FtpProxy = ProxyServer $\leftarrow$ **new**(ftp);

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

FtpProxy $\leftarrow$ **send**(**get**(`'paper.ps.gz'`))

Delegation cannot be easily expressed without first-class messages, since the requested 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 fn $x = y.x$; to select the field x from record y. Neither first-class messages nor first-class record fields can be type checked in languages from the ML family such as SML [14] or the objective ML dialect O'Caml [24].

Recently, the second author has proposed an extension to the ML type system that can deal with first-class messages [18]. He defines a type inference procedure in terms of kinded unification [20] and proves it correct. This procedure is, however, 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 [18] that considerably simplifies the original formulation, and we settle its complexity. For this purpose, 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 [4, 5, 27, 30] by a new tailor-made constraint: this new constraint is motivated by the type inference of a message sending statement o $\leftarrow$ 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, and in time $O(n^2)$ for an important special case. We also show that the satisfiability problem for positive and negative OF constraints is NP-hard, by comparing OF with Treinen's feature constraint system EF [30].

Based on OF, we define monomorphic type inference for first-class messages. Our formulation considerably simplifies the original one based on kinded unification. A key difference between both is 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.

From our complexity analysis of OF we obtain that monomorphic type inference for first-class messages can be done in polynomial time. Incrementality is important for modular program analysis without loss of efficiency in comparison to global program analysis. Our constraint-based setup of type inference allows us to explain ML-style polymorphic type inference [10, 13] as an instance HM(OF) of the HM(X) scheme [29]: Given a monomorphic type system based on constraint system X, the authors give a generic construction of HM(X), *i. e.*, type inference for ML-style polymorphic constrained types. Type inference for the polymorphic system remains DEXPTIME-complete, of course [11].

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 a given object type. An object type is a labeled collection of method types (*i. e.*, a product of function types distinguished by labels) marked by obj. *E. g.*, the object

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

implements two methods **pos** and **neg** that behave like functions from integer and boolean to boolean, respectively. Hence, it has an object type $\mathsf{obj}(\mathbf{pos}{:}\mathsf{int} \to \mathsf{bool}, \mathbf{neg}{:}\mathsf{bool} \to \mathsf{bool})$.[1] When a message $f(M)$ is sent to an object, the corresponding method is selected according to the message label f and then applied to the message argument M. Since a message parameter may refer to a variety of specific messages at run-time, it has a message type marked by msg that collects the corresponding types (as a sum of types distinguished by labels). For example, the expression

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

defines, depending on b, a message m of message type $\mathsf{msg}(\mathbf{pos}{:}\mathsf{int}, \mathbf{neg}{:}\mathsf{bool})$. 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 unique even though the message type is underspecified.

These are the crucial intuitions underlying Nishimura's type system [18]. Our type inferences captures these intuitions fully. Formally, however, our type inference implements a type system that does not exactly match the original one: Ours is slightly weaker and hence accepts more programs than Nishimura's. This weakness is crucial in order to achieve polynomial time complexity of type inference. However, type inference for a stronger system that fills this gap would require both positive and negative OF constraints and thus make type inference NP-hard.

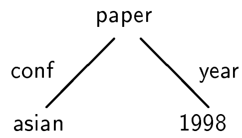
1.2 Constraint-based Type Inference

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.* [21, 32]). Constraints were also instrumental in generalizing the ML-type system towards record polymorphism [20, 23, 33], overloading [6, 19] and subtyping [1, 8] (see also [29]).

Along this line, we adopt *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 at the same node must be pairwise different.

[1] Notice that 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 the annotation of their types, but rather the field names from the associated type components. This notation is common in the literature on feature trees and record typing.

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 *resp.* 1998 at the leaves.



Feature trees can naturally model objects, records, and messages as compound data types with labeled components. 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 of “domain” and “range”.

Feature trees are the interpretation domain for a class of constraint languages called *feature constraints* [4, 5, 16, 27, 30]. 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* [22, 25]. More recently, they have been used to model record structures in constraint programming languages [2, 26, 27].

The constraint language of our system OF is this one:

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

The first three constraints are the usual ones: 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 y. For a 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 [27] 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 more features at the root than τ_y, 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. For example the ProxyServer above gets the following polymorphic constrained type:

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

Using notation from [29], this 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 return type γ; the type expresses the additional constraint that α be an object type, β be a message type appropriate for α, and the corresponding method type in α has return type γ.

Plan. Section 2 defines the constraint system OF, considers the complexity of its satisfiability problem, and compares OF with the feature constraint systems from the literature. Section 3 applies OF to recast the type inference for first-class messages and compares it with the original system [18]. Section 4 concludes the paper.

Some of the proofs in this paper are only sketched for lack of space. The complete proofs are found in an appendix of the full paper [17].

2 The Constraint System OF

2.1 Syntax and Semantics

The constraint system OF is defined as a class of constraints along with their interpretation over feature trees. We assume two infinite sets $\mathcal{V}$ of *variables* $x, y, z, \ldots$, and $\mathcal{F}$ of *features* $f, \ldots$, where $\mathcal{F}$ contains at least d and r, and a set $\mathcal{L}$ of *labels* $a, b, \ldots$ that contains at least $\rightarrow$: 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 pairs $x_1{:}y_1, \ldots, x_n{:}y_n$. We use similar notation for other syntactic categories.

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''. 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* $L : D \rightarrow \mathcal{L}$. 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 $D_{\tau.\pi} = \{\pi' \mid \pi\pi' \in D_\tau\}$ 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 $a(f_1{:}\tau_1, \ldots, f_n{:}\tau_n)$ the feature tree whose domain is $\bigcup_{i=1}^n \{f_i\pi \mid \pi \in D_{\tau_i}\}$ and whose labeling is $\{(\varepsilon, a)\} \cup \bigcup_{i=1}^n \{(f_i\pi, b) \mid (\pi, b) \in L_{\tau_i}\}$. We use $\tau_1 \rightarrow \tau_2$ to denote the feature tree τ with $L_\tau = (\varepsilon, \rightarrow)$, $\mathsf{ar}(\tau) = \{d, r\}$, $\tau.d = \tau_1$, and $\tau.r = \tau_2$.

Syntax. An *OF constraint* φ is defined as a conjunction of the following *primitive* constraints:

$x = y$	(Equality)
$a(x)$	(Labeling)
$x[f]y$	(Selection)
$F(x)$	(Arity Bound)
$x\langle y\rangle z$	(Object Selection)

Conjunction is denoted by $\wedge$. 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 the number of variable, feature, and label symbols in φ.

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

$a(\tau)$	if	$(\varepsilon, a) \in L_\tau$
$\tau[f]\tau'$	if	$\tau.f = \tau'$
$F(\tau)$	if	$\mathsf{ar}(\tau) \subseteq F$
$\tau\langle\tau'\rangle\tau''$	if	$\forall f \in \mathsf{ar}(\tau') : f \in \mathsf{ar}(\tau)$ and $\tau.f = \tau'.f \rightarrow \tau''$

Let Φ and Φ' be first-order formulas built from OF constraints with the usual first-order connectives $\vee$, $\wedge$, $\neg$, $\rightarrow$, *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 $\Phi \rightarrow \Phi'$ is valid, and that Φ is *equivalent* to Φ' if $\Phi \leftrightarrow \Phi'$ is valid.

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

$$x[f]y \wedge x[f]y' \rightarrow y = y' \tag{1}$$

$$x\langle y\rangle z \wedge x\langle y\rangle z' \not\rightarrow z = z' \tag{2}$$

The reason for the second equation not to hold is that y may have no subtrees: In this case, the constraint $x\langle y\rangle z$ does not constrain z at all. *I. e.*, this implication holds:

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

If, however, y is known to have at least one feature at the root, then selecting both z and z' by y from x implies equality of z and z':

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

OF cannot express that y has a non-empty arity; rather, to express that y has some feature it must provide a concrete witness. Using negation, this can be expressed as $\neg\{\}(x)$. However, while satisfiability for OF is polynomial, it becomes NP-hard if it is extended such that $\neg\{\}(x)$ can be expressed (see Section 2.3).

Feature Terms. For convenience, we will occasionally use feature terms [3] as a generalization of first-order terms: Feature terms t are built from variables by feature tree construction like $a(f_1{:}t_1,\ldots,f_n{:}t_n)$, where again 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 abbreviation $x = y \rightarrow z$ for the equation $x = \rightarrow(d{:}y, r{:}z)$.

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 simplification as a rewriting system on constraints in Figure 1. The theorem follows from Propositions 1, 2 and 3 below. Rules (Substitution), (Selection), (Label Clash), and (Arity Clash) are standard. Rules (Arity Propagation I/II) reflect the fact that a constraint $x\langle y\rangle z$ implies the arity bound on x to subsume the one on y. (Arity Intersection) normalizes a constraint to contain at most one arity bound per variable. (Object Selection I) reflects that $x\langle y\rangle z$ implies all features necessary for y to be also necessary for x, and (Object Selection II) establishes the relation of x, y, and z at a joint feature f.

$$\frac{\varphi \wedge x = y}{\varphi[y/x] \wedge x = y} \qquad \text{if } x \in fv(\varphi) \qquad \text{(Substitution)}$$

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

$$\frac{\varphi \wedge x\langle y\rangle z \wedge F(x)}{\varphi \wedge x\langle y\rangle z \wedge F(x) \wedge F(y)} \qquad \text{if not exists } F' : F'(y) \in \varphi \qquad \text{(Arity Propagation I)}$$

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

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

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

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

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

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

Fig. 1. Constraint Solving Rules

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 simplification. For the subsequent analysis, it is convenient to think of the fresh variables as fixed in advance. Hence, we define the finite set : $V'(\varphi) =_{def} V(\varphi) \cup \{v_{x,f} \in \mathcal{V} \mid x \in V(\varphi), f \in F(\varphi), v_{x,f} \text{ fresh}\}$.

Remark 1. In addition to the rules in Figure 1, there are two additional rules justified by implications (3) and (4):

$$\frac{\varphi \wedge x\langle y\rangle z}{\varphi} \qquad \text{if } \{\}(y) \in \varphi \qquad \text{(Empty Message)}$$

$$\frac{\varphi \wedge x\langle y\rangle z \wedge x\langle y\rangle z'}{\varphi \wedge x\langle y\rangle z \wedge z = z'} \qquad \text{if } y[f]y' \in \varphi \qquad \text{(Non-empty Message)}$$

The first one is just a simplification rule that does not have an impact on the satisfiability check. It helps reducing the size of a solved constraint and therefore saves space and time. Secondly, compact presentation of a solved constraint can be crucial in the type inference application where solved constraints must be understood by programmers. The second one is a derived rule that should be given priority over rule (Object Selection II).

Proposition 1. *The rewrite system in Figure 1 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 $x = y \wedge \varphi$ if $x \notin V(\varphi)$. We use the constraint measure (O_1, O_2, A, E, S) defined by

(O_1) 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) number of tuples (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'.
(A) number of non-eliminated variables $x \in V'(\varphi)$ for which no arity bound $F(x) \in \varphi$ exists.
(E) number of non-eliminated variables.
(S) size of constraint as defined above.

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

	O_1	O_2	A	E	S
(Arity Propagation II)	$=$	$=$	$=$	$=$	$<$
(Arity Intersection)	$=$	$=$	$=$	$=$	$<$
(Selection)	$=$	$=$	$=$	$=$	$<$
(Substitution)	$\leq$	$\leq$	$\leq$	$<$	$=$
(Arity Propagation I)	$=$	$=$	$<$	$=$	$>$
(Object Selection I)	$=$	$<$	$>$	$>$	$>$
(Object Selection II)	$<$	$=$	$=$	$=$	$>$

□

Proposition 2. *We can implement the rewrite system in Figure 1 such that it uses space* $O(n^3)$ *and incremental time* $O(n^5)$*, or, if the number of features is bounded, such that it uses linear space and incremental time* $O(n^2)$*.*

Proof. 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.

For lack of space, we defer some involved parts of the proof to the full paper [17].

Define $n_i = \#V(\varphi)$, $n_v = n_i \cdot n_f = \#V'(\varphi)$, $n_l = \#L(\varphi)$, $n_f = \#F(\varphi)$. In the full paper, we define a data structure for the store that consists of a union-find data structure [12] for equations, tables for the constraints $a(x)$, $F(x)$, and $x[f]z$, a list for constraints $x\langle y\rangle z$, and two adjacency list representations of the graphs whose nodes are the initial variables, and whose edges (x,y) are given by the constraints $x\langle y\rangle z$ for the first one and $y\langle x\rangle z$ for the second one. (See appendix of the full paper [17] for details). This data structure has size

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

which is $O(n)$ if the number of features is assumed constant and $O(n^3)$ otherwise. It also allows to check in time $O(1)$ whether it contains a given primitive constraint; and to add it in quasi-constant time.[2] This is clear in the non-incremental (off-line) case where n_v, n_i, n_f,and n_s are fixed. In the incremental (on-line) case, where n_v, n_i, n_f, and n_s may grow, we can use dynamically extensible hash tables [7] to retain constant time check and update for primitive constraints.

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 simplification 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 time $O(n + n_i \cdot n_f)$.[3] (The complete analysis of this time bound can be found in appendix of the full paper [17]). We also show that every step may at most add $O(n)$ equations and $O(n_v)$ selection constraints of the form $x[f]y$. 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 execute n_v equations before all variables are equated. That is, all but $O(n_v)$ equations cost constant time.
- Amongst the new selection constraint, 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 $O(n + n_v \cdot n_f)$ steps that cost $O(n + n_i \cdot n_f)$. Each of these steps may add $O(n)$ equations and $O(n_v)$ selections each of which may add a new equation itself. Hence we have $O((n + n_v \cdot n_f) \cdot (n + n_v))$ steps that cost $O(1)$. Overall, the algorithm has the complexity

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

Since $O(n_f) = O(n)$ and $O(n_v) = O(n_i \cdot n_f) = O(n^2)$, this bound is $O(n^5)$. If the number of features is bounded, $O(n_v) = O(n_i) = O(n)$, so the bound is rather $O(n^2)$. $\square$

Notice that a constraint system of records with first-class record labels is obtained as an obvious restriction of OF and the above result implies the same time complexity bound as OF.

---

[2] All constraints except equations can be added in time $O(1)$, but addition of an equation costs amortized time $O(\alpha(n_v))$ where $\alpha(n_v)$ is the inverse of Ackermann's function. For all practical purposes, $\alpha(n_v)$ can be considered as constant.

[3] To be precise, each step costs $O(n+n_i \cdot n_f + \alpha(n_v))$ which we sloppily simplify to $O(n+n_i \cdot n_f)$.

Proposition 3. *Every OF constraint φ which is closed under the rules in Figure 1 (and hence is different from fail) is satisfiable.*

Proof. For the proof, we need to define a notion of path reachability similar to the one used in earlier work, such as [15, 16]. For all paths π and constraints φ, we define a binary relation $\overset{\varphi}{\rightsquigarrow}_\pi$, where $x \overset{\varphi}{\rightsquigarrow}_\pi y$ reads as "y is reachable from x over path π in φ":

$$\begin{array}{lll} x \overset{\varphi}{\rightsquigarrow}_\varepsilon x & \text{for every } x & \\ x \overset{\varphi}{\rightsquigarrow}_\varepsilon y & \text{if} & y = x \in \varphi \text{ or } x = y \in \varphi \\ x \overset{\varphi}{\rightsquigarrow}_f y & \text{if} & x[f]y \in \varphi \\ x \overset{\varphi}{\rightsquigarrow}_{\pi\pi'} y & \text{if} & x \overset{\varphi}{\rightsquigarrow}_\pi z \text{ and } z \overset{\varphi}{\rightsquigarrow}_{\pi'} y. \end{array}$$

Define relations $x \overset{\varphi}{\rightsquigarrow}_\pi a$ meaning that "label a can be reached from x over path π in φ":

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

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

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

It remains to be shown that α defines a mapping into feature trees for closed constraints φ, and that α indeed satisfies φ. This can be done by a straightforward induction over paths π. $\square$

2.3 Relation to Feature Constraint Systems

We compare OF with feature constraint systems in the literature: Given a two-sorted signature with variables $x, y, z, \ldots$ and $u, v, w, \ldots$ ranging over feature trees and features, *resp.*, collections of feature constraints from the following list have, amongst others, been considered [3, 27, 30]:

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

The constraints $x = y$, $a(x)$, and $x[f]y$ are the ones of OF. The arity bound Fx (where F is a finite set of features) states that x has *exactly* the features in F at the root.

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

Apparently, both arity constraints are interreducible by means 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 [27] extends FT by Fx, and EF [30] contains the constraints $x{=}y$, $a(x)$, $x = f$, Fx, and $x[u]y$.

The satisfiability problems for FT and CFT are quasi-linear [27]. In contrast, the satisfiability problem for EF is NP-hard, as Treinen shows by reducing the minimal cover problem to it [9, 30]. Crucial in this proof is the following implication

$$\{f_1, \ldots, f_n\}x \wedge x[u]y \quad \rightarrow \quad \bigvee_{i=1}^{n} u = f_i$$

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

$$\{f_1,\dots,f_n\}(x) \wedge x\langle y\rangle z \wedge \neg\{\}(y) \quad \rightarrow \quad \bigvee_{i=1}^{n} \exists z_i\, y[f_i]z_i$$

With this new constraint we reduce the satisfiability check for EF to the one for OF.

Proposition 4. *There is an embedding $[\![\cdot]\!]$ from EF constraints into OF with negative constraints of the form $\neg\{\}(x)$ such that every EF constraint φ is satisfiable iff $[\![\varphi]\!]$ is.*

Proof. Labeling $a(x)$ and equality $x = y$ translate trivially. Now assume two special labels unit and lab; we use these to represent labels f in EF by feature trees $\mathsf{lab}(f{:}\mathsf{unit})$.

$$\begin{array}{rcl} [\![u = f]\!] & = & \exists x\,(\mathsf{lab}(u) \wedge \{f\}(u) \wedge u[f]x \wedge \mathsf{unit}(x) \wedge \{\}(x)) \\ [\![x[u]y]\!] & = & x\langle u\rangle y \wedge \neg\{\}(u) \\ [\![\{f_1,\dots,f_n\}x]\!] & = & \{f_1,\dots,f_n\}(x) \wedge \bigwedge_{i=1}^{n} \exists y\, x[f_i]y \end{array}$$

To show that satisfiability of an EF constraint φ implies satisfiability of the OF constraint $[\![\varphi]\!]$ we map every EF solution of φ to an OF solution of $[\![\varphi]\!]$ by replacing every feature f by $\mathsf{lab}(f{:}\mathsf{unit})$ and every feature tree of the form $a(f{:}\tau\dots)$ by $a(f{:}\mathsf{unit} \rightarrow \tau\dots)$.

For the inverse, we take a satisfiable OF constraint $[\![\varphi]\!]$ and construct an OF solution of $[\![\varphi]\!]$ which maps all variables u in selector position in $x\langle u\rangle y$ to a feature tree with exactly one feature. From this solution we derive an EF solution of φ by replacing these singleton-feature trees by their unique feature. Notice that the solution constructed in the proof of Proposition 3 does not suffice since it may map u to a feature tree without any feature.

Formally, we extend the definition of path reachability by $x \overset{\varphi}{\leadsto}_\pi F$ meaning that "arity bound F can be reached from x over path π in φ":

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

We assume an order on $F(\varphi)$, and, for non-empty F, let $min(F)$ denote the smallest feature in F wrt. this order. We define α as follows:

$$\begin{array}{rcl} D_{\alpha(x)} & = & \{\pi \mid \text{exists } y:\ x \overset{\varphi}{\leadsto}_\pi y\} \cup \{\pi f \mid x \overset{\varphi}{\leadsto}_\pi F, f = min(F)\} \\ L_{\alpha(x)} & = & \{(\pi,a) \mid x \overset{\varphi}{\leadsto}_\pi a\} \cup \{(\pi,\mathsf{unit}) \mid \pi \in D_{\alpha(x)},\ \nexists a : x \overset{\varphi}{\leadsto}_\pi a\} \end{array}$$

By the constraint $\neg\{\}(u)$ in the translation of $x\langle u\rangle y$ we know that, for closed and non-failed φ, the set F must be non-empty in the first line. Hence, α is a well-defined mapping into feature trees. It is easy to show that α satisfies φ and that α corresponds to an EF solution as sketched above. $\square$

Corollary 1. *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.

$$\frac{x : t \in \Gamma}{\varphi, \Gamma \vdash x : t} \text{ VAR} \qquad \frac{}{\varphi, \Gamma \vdash b : \mathsf{typeof}(b)} \text{ CONST} \qquad \frac{\varphi, \Gamma \vdash M : t' \quad \varphi \models_{\text{OF}} t :: \mathsf{msg}(f : t')}{\varphi, \Gamma \vdash f(M) : t} \text{ MSG}$$

$$\frac{\varphi, \Gamma; x_i : t_i \vdash M_i : t_i' \quad \text{for every } i = 1, \ldots, n}{\varphi, \Gamma \vdash \{f_1(x_1) = M_1, \ldots, f_n(x_n) = M_n\} : \mathsf{obj}(f_1{:}t_1{\to}t_n', \ldots, f_n{:}t_n{\to}t_n')} \text{ OBJ}$$

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

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

Fig. 2. The Monomorphic Type System

3 Type Inference

We reformulate the type inference of [18] in terms of OF constraints. As in that paper, we consider a tiny object-oriented programming language with this abstract syntax.[4]

M	$::=$	b	(Constant)
	$\mid$	x	(Variable)
	$\mid$	$f(M)$	(Message)
	$\mid$	$\{f_1(x_1) = M_1, \ldots, f_n(x_n) = M_n\}$	(Object)
	$\mid$	$M \leftarrow N$	(Message Passing)
	$\mid$	$\mathsf{let}\ y = M\ \mathsf{in}\ N$	(Let Binding)

The operational semantics contains no surprise (see [18]). 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 all feature trees over this signature; monomorphic *type terms* are feature terms:

t	$::=$	α	(Type variable)
	$\mid$	$\mathsf{int} \mid \mathsf{bool} \mid \ldots$	(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)

Somewhat sloppily, we allow for infinite (regular) feature terms such as to incorporate recursive types without an explicit μ notation. Recursive types are necessary for the analysis of recursive objects. We assume a mapping typeof from constants of base type to their corresponding types. We also use the *kinding* notation $x :: a(f_1{:}t_1, \ldots, f_n{:}t_n)$ to constrain x to a feature tree whose arity is underspecified, *e. g.*, $a(x) \wedge \bigwedge_{i=1}^{n} x[f_i]t_i$.

Monomorphic Type Inference. The monomorphic type system is given in Figure 2. As usual, Γ is a finite mapping from variables to type terms and $\Gamma; x : t$ extends Γ so

[4] In contrast to [18], we drop letobj and allow let to introduce recursively defined expressions.

$$
\begin{array}{lcll}
I(x,b) & = & a(x) \wedge \{\}(x) & \text{if } a = \mathsf{typeof}(b) \\
I(x,y) & = & x = y & \\
I(x,f(M)) & = & \exists y\ (\mathsf{msg}(x) \wedge x[f]y \wedge I(y,M)) & \\
I(x,\{f_1(x_1) = M_1,\ldots,f_n(x_n) = M_n\}) & = & \mathsf{obj}(x) \wedge \{f_1,\ldots,f_n\}(x) \wedge & \\
 & & \quad \bigwedge_{i=1}^{n} \exists x_i\ \exists x'\ \exists z\ (x[f_i]x' \wedge x' = x_i \to z \wedge I(z,M_i)) & \\
I(x,M \leftarrow N) & = & \exists y\ \exists z\ (y\langle z\rangle x \wedge \mathsf{obj}(y) \wedge I(y,M) \wedge \mathsf{msg}(z) \wedge I(z,N)) & \\
I(x,\mathsf{let}\ y = M\ \mathsf{in}\ N) & = & \exists y\ (\ I(y,M) \wedge I(x,N)) &
\end{array}
$$

Fig. 3. Monomorphic Type Inference for First-Class Messages with OF Constraints

that it maps variable x to t. The type system defines judgments $\varphi,\Gamma \vdash M : t$ which reads as "under the type assumptions in Γ subject to the constraint φ, the expression M has type t";[5] the constraint φ in well-formed judgements is required to be satisfiable. We do not comment further on the type system here but refer to [18] for intuitions and to [28, 29] for notation. The corresponding type inference is given in Figure 3 as a mapping I from a variable x and a program expressions M to an OF constraint such that every solution of x in $I(x,M)$ is a type of M. For ease of reading, we use the bound variables in program expressions as their corresponding type variables. Correctness of the type inference with respect to the type system is obvious, and it should be clear that soundness of the type system (with respect to the assumed operational semantics) can be shown along the lines given in [18]. The type inference generates a constraint whose size is proportional to the size of the given program expression. Hence, we know from Proposition 2 that type inference can be done in polynomial time and space.[6]

Let us give some examples. To reduce the verbosity of OF constraints, we shall freely use feature term equations as introduced above. First, the statement

let o1 = {**succ**(x)=x+1, **pos**(x)=x>0};

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

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

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

let o2 = {**redirect**(m)= o1 ← 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:

[5] 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".

[6] To be precise, we have to show that every satisfiable OF constraint derived by type inference is satisfiable in the smaller domain of types; this is easy.

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

The return type of a message passing to this object, *e. g.*,

```
let w = o2← redirect(succ(1));
```

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

$$\varphi_3 \quad \equiv \quad \exists z'\, (o_2\langle z'\rangle w \wedge z' :: \mathsf{msg}(\mathbf{redirect} : \mathsf{msg}(\mathbf{succ} : \mathsf{int}))),$$

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

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

```
let v = o2← redirect(pred(1)),
```

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

$$\varphi_4 \quad \equiv \quad \exists z'\, (o_2\langle z'\rangle w' \wedge z' :: \mathsf{msg}(\mathbf{redirect} : \mathsf{msg}(\mathbf{pred} : \mathsf{int})))$$

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

Now recall that in OF the implication $x\langle y\rangle z \wedge x\langle y\rangle z' \to z = z'$ does *not* hold. The following example demonstrates how this weakness affects typing and type inference.

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

It is easy to see that the **foo** method always returns bool, since the argument message of **foo** must be accepted by both the objects o1 and o2, which share only the method name **b**. However, type inference for this program derives (essentially) the constraint

$$o_1 = \mathsf{obj}(\mathbf{a} : \mathsf{int}{\to}\mathsf{int}, \mathbf{b} : \mathsf{int}{\to}\mathsf{bool}) \wedge o_2 = \mathsf{obj}(\mathbf{b} : \mathsf{int}{\to}\mathsf{bool}, \mathbf{c} : \mathsf{int}{\to}\mathsf{int}) \wedge$$
$$o_3 = \mathsf{obj}(\mathbf{foo} : m{\to}z) \wedge o_1\langle m\rangle z_1 \wedge o_2\langle m\rangle z_2$$

Herein, the result type z of the method **foo** is neither entailed to equal z_1 nor z_2. This is reasonable since the message m in this program is not sent and hence may safely return anything. By a similar argument, the following program can be considered acceptable:

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

One may complain that this kind of methods should be detected as a type error. Manipulating the type system and the type inference to do this is easy: One just needs to exclude types $\mathsf{msg}()$, *i. e.*, message types without any feature. However, recall that the polynomial time complexity of the analysis depends on this weakness: The corresponding clause in the type inference

$$I(x, f(M)) = \exists y\, (\neg\{\}(x) \wedge \mathsf{msg}(x) \wedge x[f]y \wedge I(y, M))$$

generates OF constraints with negation such that constraint solving (and hence, type inference) would become NP-hard.[7]

[7] The altered type inference does still not exactly correspond to the original. For example, we would not accept message sending to an empty object as in $\{\} \leftarrow$ m, while the original one does.

Polymorphic Type Inference. We can obtain the polymorphic type inference by applying the scheme HM(X) [29]. The constraint system OF is a viable parameter for HM(X) since it satisfies the two required properties, called coherence and soundness. Both rely on a notion of monomorphic types, in our case, given by feature trees; it does no harm that these may be infinite. The *coherence* property requires that the considered order on types is semantically well-behaved; this is trivial in our case since we only consider a trivial order on feature trees. The *soundness* property that a solved constraint indeed has a solution follows from Proposition 3.

Comparison with Nishimura. In Nishimura's original type system [18], abbreviated 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 $x :: \{\!|y_1{\rightarrow}t_1, \ldots, y_n{\rightarrow}t_n|\!\}_F$ 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) [29] of Ohori's type system for the polymorphic record calculus: Both recast the kinding system as a constraint system. 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 the kind system in $\mathcal{D}$ and OF. In $\mathcal{D}$, kinded types may contain variables, *e. g.*, an object returning integers as a response to messages of type y receives the kind $\{\!|y{\rightarrow}\mathsf{int}|\!\}_F$. On unifying two types with kindings $\{\!|y{\rightarrow}\mathsf{int}|\!\}_F$ and $\{\!|y{\rightarrow}z|\!\}_F$, the type inference for $\mathcal{D}$ unifies 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. *E. g.*, we understand $\{\!|y{\rightarrow}\mathsf{int}|\!\}$ 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:

```
{foo(m) = (o←m) + 1; (o←m) & 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. The type system $\mathcal{D}$, however, rejects it after trying to unify int and bool during type inference. To insist that this is a syntactic argument notice that $\mathcal{D}$ accepts the following program, where o is replaced by the object constant {}:

```
{baz(m) = ({}←m) + 1; ({}←m) & true}
```

4 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 [18].

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 the additional expressiveness would render monomorphic type inference NP-hard as we have shown. Given OF, we can add ML-style polymorphism by instantiating the recent HM(X) scheme to the constraint system OF.

Acknowledgements. 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). We also acknowledge helpful remarks of the referees.

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. 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 onComputational Linguistics and Logic Programming.
5. R. Backofen and G. Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268, July 1995.
6. 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.
7. 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.
8. 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.
9. 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.
10. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.
11. H. G. Mairson. Deciding ML typebility is complete for deterministic exponential time. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pp. 382–401. ACM Press, New York, Jan. 1990.
12. K. Mehlhorn and P. Tsakalides. Data structures. In van Leeuwen [31], chapter 6, pp. 301–342.
13. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.
14. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

15. 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.
16. 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 submitted to special journal issue of CP'97.
17. M. Müller and S. Nishimura. Type inference for first-class messages with feature constraints. Technical report, Programming Systems Lab, Universität des Saarlandes, 1998. `http://www.ps.uni-sb.de/Papers/abstracts/FirstClass98.html`.
18. 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.
19. 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*. ACM Press, New York, 1995.
20. A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.
21. 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.
22. C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. Cambridge University Press, Cambridge, England, 1994.
23. 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.
24. D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 40–53. ACM Press, New York, 1997.
25. 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.
26. 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.
27. G. Smolka and R. Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.
28. M. Sulzmann. Proofs of properties about HM(X). Technical Report YALEU/DCS/RR-1102, Yale University, 1998.
29. M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types (extended abstract). In B. Pierce, ed., *Proceedings of the 4th International Workshop on Foundations of Object-oriented Programming*, Jan. 1997. Full version to appear in TAPOS, 1998.
30. 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.
31. J. van Leeuwen, ed. *Handbook of Theoretical Computer Science*, vol. A (Algorithms and Complexity). The MIT Press, Cambridge, MA, 1990.
32. 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.
33. M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.