Contractual Types

Oscar Nierstrasz

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-03-004

August 20, 2003

Abstract

Real software systems are open and evolving. It is a constant challenge in such environments to ensure that software components are safely composed in the face of changing dependencies and incomplete knowledge. To address this problem, we propose a new kind of type system which allows us to infer not only the type *provided* by a software component in an open system, but also the type it *requires* of its environment, subject to certain constraints. The *contractual type* we infer for components can then be statically checked when components are composed. To illustrate our approach, we introduce the *form calculus*, a calculus of explicit environments, and we present a type system that infers types for form expressions.

CR Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory—*Semantics*; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*Type structure*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*Lambda calculus and related systems*

1 Introduction

Real software systems evolve with time. Individual components are often developed and adapted without complete knowledge of the rest of the system. In order for one to determine whether a system is safely composed, all the pieces must typically be frozen and available. Changes to the interfaces offered by components can have far-reaching effects, and can require extensive analysis to ensure that no dependencies are broken. We propose to alleviate this problem by means of a type system that expresses not only what a software component *provides*, but also what it *requires* from an environment in which it will be used. Software components can then be safely composed when provided types *satisfy* the corresponding required types.

Consider, for example, the following code written in a Java-like language:

```
import com.dohickeys.*;
class Gadget extends Widget {
  FooBar tinker(Whatsit w) {
    return this.munge(new Thing(w));
  }
}
```

Suppose now that we would like to reason about this software component without over-specifying the components imported from com.dohickeys. (Let us assume, for simplicity, that nothing is implicitly imported from anywhere else.) Based on a static analysis of this code, we might express what it *provides* as follows:

$$\mathsf{Gadget}{:}() \rightarrow (\mathsf{munge}{:}\, \mathit{Thing} \rightarrow \mathit{FooBar} \cdot \mathsf{tinker}{:}\, \mathit{Whatsit} \rightarrow \mathit{FooBar})$$

i.e., a default constructor called Gadget, which yields an object with (at least) methods munge and tinker. About the argument and return types of these methods, we can say nothing further.

We can, however, say something more about the assumptions this component places on its environment. In particular, the component will safely provide what it does if and only if the environment satisfies the following requirement:

$$\mathsf{com{:}dohickeys{:}}(\mathsf{Widget}{:}() \rightarrow \mathsf{munge}{:}\, \mathit{Thing} \rightarrow \mathit{FooBar} \wedge \mathsf{Thing}{:}\, \mathit{Whatsit} \rightarrow \mathit{Thing})$$

i.e., within the environment com there must be an environment dohickeys which in turn must provide suitable constructors for Widget and Thing, and a Widget must provide a munge method.

The key idea is to express as a *required type* the requirements posed by the free variables of our software component on the environment. If we *close* our Gadget component by composing it with a concrete dohickeys component, we must check that the services *provided* by dohickeys satisfy the required type of Gadget. If we later substitute another version of dohickeys, we do *not* require that these two versions be in a subtype relation with each other, merely that both satisfy the required type of Gadget, and nothing more.

We propose a type system in which the type $T\{S\}$ of a software component specifies a *contract*: it will provide the services specified in T if the environment satisfies the required

type S. The type of our component might then be given as:

$$\begin{array}{l}\mathsf{Gadget}{:}()\rightarrow(\mathsf{munge}{:}\,\mathit{Thing}\rightarrow\mathit{FooBar}\cdot\mathsf{tinker}{:}\,\mathit{Whatsit}\rightarrow\mathit{FooBar})\ \{\\ \quad\mathsf{com{:}dohickeys}{:}(\mathsf{Widget}{:}()\rightarrow\mathsf{munge}{:}\,\mathit{Thing}\rightarrow\mathit{FooBar}\wedge\mathsf{Thing}{:}\,\mathit{Whatsit}\rightarrow\mathit{Thing})\\ \}\end{array}$$

Note that the type names *Thing*, *FooBar*, and *Whatsit* are unconstrained type variables. The environment may freely bind them to any concrete type. In general, type variables may have to satisfy certain constraints of the form $P\hookrightarrow R$, which express that a provided type P must *satisfy* the required type R. As we shall see, there is a close affinity between the contractual type $P\{R\}$ of an open component, and the functional type $R\rightarrow P$ of a closed expression that abstracts over the environment in which that component will be instantiated.

From this example it should be clear that we wish to manipulate, compose, type and typecheck environments. For this reason we introduce the pure *form calculus*, a simple, untyped calculus of first-class environments. The form calculus is essentially a λ calculus of explicit substitutions [1] in which substitutions are first-class values [19].

The contributions of this paper are:

- Syntax and semantics of the pure *form calculus*,
- a proposed system of *contractual types* for form expressions,
- type inference rules that generate contractual type expressions, with satisfaction constraints, for both closed and open form expressions, and
- partial type-checking rules for validating type constraints.

In section 2 we introduce the pure *form calculus*. In section 3 we introduce *contractual types* for the form calculus. We evaluate the type system with the help of a series of examples in section 4. In section 5 we discuss related work. Finally, in section 6, we outline ongoing and future work, and present our conclusions.

2 The Form Calculus

It is now generally accepted that a software component is "a unit of independent deployment, a unit of third-party composition [that] has no persistent state" [22]. A key aspect in this definition, however, is the term *composition*, which suggests that deployment may not always be so independent. In fact, a useful alternative definition of component is "a static abstraction with plugs" [17], which emphasizes that a component not only *provides*, but also *requires* services. In what way, however, should we characterize what components provide and require, *independently* of concrete implementations of other third-party components?

We claim that composition of software components can be understood in terms of *composition of namespaces*, and we therefore propose to model components with a calculus of first-class environment.

We seek to model software components that both provide and require sets of services. We are not concerned here with issues of concurrency or reflection, and therefore limit our attention here to purely functional composition without side effects. We are also not concerned here with non-uniform service availability, and so do not consider issues of specifying or

checking *protocols* over provided services. In order to study the composition of software components, we propose a simple calculus of first-class environments, or *forms*.

We introduce the word "form" rather than using "record" or "environment" to emphasize the the multiple roles that forms play in the calculus:

- A form may provide a *service* which can be invoked;
- a form may contain zero or more *bindings* of labels to values;
- a form may be used as the *environment* in which an expression is evaluated;
- a form is itself a *first-class value* that may be bound to a label, or passed as an argument to a service.

We similarly adopt the term "service" rather than "function" to emphasize the fact that our goal is to model composition of software components, not composition of pure functions.

Table 1 presents the syntax and semantics of the form calculus. Application is left-associative and $\cdot$ and ; are right associative. The terms are listed in order of precedence, with ; binding loosest. Closed form expressions may be evaluated and, if they terminate, reduce to the sublanguage of form values. Evaluation consists in reducing applications (function calls) and sandbox expressions (lookup of names in an environment). Pre-values may contain such expressions only underneath a lambda. A form value is a closed pre-value. ($x{=}y$ is a pre-value, but not a value since $\mathit{free}(x{=}y) = \{y\}$.)

Note that the form calculus contains the usual untyped λ calculus as a sublanguage. Our intent is that the embedded λ calculus should have the usual interpretation, except that substitutions are made explicit through forms. The new constructs are the following: $()$ represents an empty form providing neither a service to be invoked, nor any bindings that can be looked up. $x{=}F$ represents a binding of the label x to the form expression F. $E{\cdot}F$ extends the form E by F. Bindings and services present in F *override* those present in E. Finally, $E;F$ evaluates F within the environment defined by the form E. Note that this implies that $E;F$ *closes* F. Any free labels occurring in F must be bound in E, or they will lead to an error. In particular, $();x$ is an erroneous term since x is not bound in the environment $()$.

Free labels are defined in the usual way. The only surprising rule is perhaps the last one: $\mathit{free}(E;F) = \mathit{free}(E)$. This makes clear that F is closed by E. In the expression $\mathsf{com};\mathsf{dohickeys};\mathsf{Widget}$, for example, the free name Widget must be bound in $\mathsf{dohickeys}$, which in turn must be looked up in com.

Only closed form expressions may be reduced. The rule *Apply* shows how β-reduction is modeled by evaluating the body of an abstraction in an environment in which the formal parameter is bound to the argument. Note that U and V must be form values since syntactically they are pre-values, and only closed expressions may be reduced. Evaluation is therefore strict in the form calculus.

The rule *Apply error* states that an error may occur during application only if the form being applied contains no service. The rule *Substitute* resolves all free names of an open expression E by binding them in the environment U. Finally, closed subexpressions may also be reduced within any context $E[\cdot]$, except underneath a lambda.

The identity function, for example, works as in the usual untyped λ calculus, but requires

Syntax of form expressions and pre-values

$$
\begin{array}{rcll}
E, F, G & ::= & () & \textit{Empty form} \\
& | & x & \textit{Label lookup} \\
& | & FE & \textit{Service application} \\
& | & \lambda x.F & \textit{Service definition} \\
& | & x{=}F & \textit{Label binding} \\
& | & E{\cdot}F & \textit{Form extension} \\
& | & E; F & \textit{Sandbox}
\end{array}
\qquad
\begin{array}{rcl}
U, V, W & ::= & () \\
& | & x \\
& | & \lambda x.F \\
& | & x{=}V \\
& | & U{\cdot}V
\end{array}
$$

Free names

$$
\begin{array}{rcl}
\textit{free}(()) & = & \emptyset \\
\textit{free}(x) & = & \{x\} \\
\textit{free}(x{=}F) & = & \textit{free}(F) \\
\textit{free}(\lambda x.F) & = & \textit{free}(F) - \{x\}
\end{array}
\qquad
\begin{array}{rcl}
\textit{free}(FE) & = & \textit{free}(F) \cup \textit{free}(E) \\
\textit{free}(E{\cdot}F) & = & \textit{free}(E) \cup \textit{free}(F) \\
\textit{free}(E; F) & = & \textit{free}(E)
\end{array}
$$

Reduction of closed form expressions

$$
\begin{array}{rcll}
U\ V & \rightarrow & x{=}V; F & \textit{if } U_\lambda = \lambda x.F \qquad \textit{Apply} \\
U\ V & \rightarrow & \bot & \textit{if } U_\lambda = \bot \qquad \textit{Apply error} \\
U; E & \rightarrow & U[\![E]\!] & \textit{Substitute} \\
E[F] & \rightarrow & E[F'] & \textit{if } F \rightarrow F' \textit{ and } E[\cdot] \neq \lambda x.[\cdot] \qquad \textit{Nested reduction}
\end{array}
$$

Lookup of services and labels

$$
U_\lambda = \begin{cases} \lambda x.F & \textit{if } U \equiv V{\cdot}\lambda x.F \\ \bot & \textit{otherwise} \end{cases}
\qquad
U_x = \begin{cases} W & \textit{if } U \equiv V{\cdot}x{=}W \\ \bot & \textit{otherwise} \end{cases}
$$

Structural equivalence of form values

$$
\begin{array}{rcl}
()\cdot U & \equiv & U \\
U\cdot() & \equiv & U \\
U\cdot(V\cdot W) & \equiv & (U\cdot V)\cdot W
\end{array}
\qquad
\begin{array}{rcll}
x{=}V{\cdot}x{=}W & \equiv & x{=}W & \\
x{=}V{\cdot}y{=}W & \equiv & y{=}W{\cdot}x{=}V & \textit{if } x \neq y \\
\lambda x.E{\cdot}\lambda y.F & \equiv & \lambda y.F & \\
x{=}V{\cdot}\lambda y.F & \equiv & \lambda y.F{\cdot}x{=}V &
\end{array}
$$

Substitution

$$
\begin{array}{rcll}
U[\![()]\!] & = & () & \textit{Empty} \\
U[\![x]\!] & = & U_x & \textit{Lookup} \\
U[\![x{=}E]\!] & = & x{=}U[\![E]\!] & \textit{Close binding} \\
U[\![\lambda x.F]\!] & = & \lambda x.((U{\cdot}x{=}x)[\![F]\!]) & \textit{Close service} \\
U[\![FE]\!] & = & U[\![F]\!]\ U[\![E]\!] & \textit{Close application} \\
U[\![F{\cdot}E]\!] & = & U[\![F]\!]{\cdot}U[\![E]\!] & \textit{Close extension} \\
U[\![E; F]\!] & = & U[\![E]\!]; F & \textit{Close sandbox}
\end{array}
$$

Table 1: Pure Form Calculus — Syntax and Semantics

an extra reduction step:

$$\begin{array}{rcl}(\lambda x.x)\ () & \rightarrow & x{=}(); x \\ & = & ()\end{array}$$

Forms may contain both ordinary label bindings as well as services, so we have lookup functions for each of these. Note that a *Substitute* reduction may result in an error if the label x being looked up is not bound in the environment U. So errors may arise only when applying a form without a service, or when looking up a label that is not bound. The definition of lookup makes use of structural equivalence for form pre-values.

$U[\![E]\!]$ looks up all free names in E and replaces them by their bindings in the pre-value U. For example,

$$x{=}()[\![x]\!] \ = \ ()$$

The only unusual equation is *Close service*. Note that in $U[\![\lambda x.F]\!]$ we want all free labels in F to be looked up in U, *except* for x, which must be captured by λx. This is achieved by the translation to $\lambda x.(U{\cdot}x{=}x)[\![F]\!]$, which overrides any binding for x that may occur in U by a binding to the enclosing λx. Consider, for example,

$$\begin{array}{rcll}x{=}()[\![\lambda x.x]\!] & = & \lambda x.((x{=}(){\cdot}x{=}x)[\![x]\!]) & \\ & = & \lambda x.x & \text{(and not } \lambda x.() \text{ !)}\end{array}$$

This example illustrates why (open) pre-values, like $x{=}x$, and not just (closed) values can be used as environments for substitution.

2.1 Errors

Errors arise when at attempt is made to invoke a form expression that does not contain a service, or when a label is looked up in a form that does not contain a binding for it. Here *getb* is bound to a service that looks up b in its argument form. When we apply it to x, which contains a binding for a, but not for b, an error occurs:

$$\begin{array}{rcl}x{=}a{=}(){\cdot}getb{=}\lambda y.(y;b);\ getb\ x & \rightarrow & (\lambda y.(y;b))\ (a{=}()) \\ & \rightarrow & y{=}a{=}(); y; b \\ & \rightarrow & a{=}(); b \\ & \rightarrow & \bot\end{array}$$

By the same token, if we try to apply x to *getb*, this will result in an error, because x contains no service:

$$\begin{array}{rcl}x{=}a{=}(){\cdot}getb{=}\lambda y.(y;b); x\ \ getb & \rightarrow & (a{=}())\ (\lambda y.(y;b)) \\ & \rightarrow & \bot\end{array}$$

A type system for open form expressions should not only express what is provided and required, but should be able to determine which composite expressions are erroneous.

2.2 Form Calculus — Expressiveness

The form calculus we have presented here captures the essential aspects concerning component composition that we wish to study: nested namespaces providing and requiring various services, composition of first-class namespaces, and abstraction over namespaces and composition. We ignore, on the other hand, issues of state, concurrency and synchronization.

If we reconsider the example outlined in the introduction, we might express it in the form calculus as follows:

$$\mathsf{com};\mathsf{dohickeys};\mathsf{gadget}{=}\lambda x.(\mathsf{widget}\ ()\cdot\mathsf{tinker}{=}\lambda w.(\mathsf{widget}\ ();\mathsf{munge})\ (\mathsf{thing}\ w))$$

The only free name here is com, which represents an environment containing the dohickeys namespace. Note that FooBar, Whatsit and Thing are absent, as they represent types, which are not modelled in the form calculus.

3 Contractual Types

We want to be able to type not only closed form expressions but also *open* ones. The type of an open expression will specify not only what it *provides*, but also what it *requires* from the environment. As a consequence, our type judgments will take the form:

$$\vdash F :: P\{R\} \mid C$$

where P represents what F provides, and R represents what F requires of an environment that binds its free variables. (Since we use : in the syntax of types, we will avoid confusion by using :: to indicate that a form F has a certain type $P\{R\}$.) C is a set of constraints on the free *type variables* occurring in P and R of the form $P \hookrightarrow R$, which express that provided type P must *satisfy* the requirements posed by R.

Our intention is that whenever these constraints can be satisfied, we will derive a simpler type judgment of the form $\vdash F :: P\{R\}$. If, on the other hand, we can derive $\vdash F :: \bot$, then F will be untypable, and consequently erroneous. As we shall see, the type-checking rules we propose are partial, and constraints can only be checked when enough information is available.

The syntax of contractual types is shown in table 2.

Provided types characterize form values and closed form expressions. () is the type of the empty form. α is a type variable, as in $\lambda x.x :: \alpha{\to}\alpha$. We use Greek letters to range over type variables. Later we will need to distinguish between type variables that occur *positively* (*i.e.*, with a positive superscript), representing a provided type, and those that occur *negatively*, representing a required type. Generally we will drop the superscripts for readability, as they can easily be recovered from the syntactic context of a variable, so we may write $\lambda x.x :: \alpha{\to}\alpha$ instead of $\lambda x.x :: \alpha^{-}{\to}\alpha^{+}$.

$x{:}P$ types a form with a bound label, as in $x{=}() :: x{:}()$. In the expression $P_1{\cdot}P_2$, P_2 may override P_1.

Required types are syntactically distinct from provided types, since they express conjunctions of logical requirements on an environment. Conjunctions of requirements are written $R \wedge R$ (cf. *intersection types* [11]).

Finally, contractual types characterize open expressions. If F is closed, then R is empty. So any provided type P can also be interpreted as the contractual type $P\{()\}$.

Syntax of Provided, Required and Contractual Types

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl@{\qquad}rcl}
P & ::= & () & R & ::= & () & C & ::= & \top & T & ::= & P \\
 & | & \alpha^{+} & & | & \alpha^{-} & & | & \bot & & | & P\{R\} \\
 & | & x{:}P & & | & x{:}R & & | & C, P \hookrightarrow R & & | & P\{R\} \mid C \\
 & | & P{\cdot}P & & | & R \wedge R & & & & & & \\
 & | & R{\to}P & & | & P{\to}R & & & & & &
\end{array}
$$

Type Equivalence

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
()\cdot P & \equiv & P & R{\to}P{\cdot}Q{\to}S & \equiv & Q{\to}S \\
P{\cdot}() & \equiv & P & x{:}S{\cdot}R{\to}P & \equiv & R{\to}P{\cdot}x{:}S \\
P{\cdot}(S{\cdot}Q) & \equiv & (P{\cdot}S){\cdot}Q & R \wedge Q & \equiv & Q \wedge R \\
x{:}P{\cdot}x{:}S & \equiv & x{:}S & R \wedge () & \equiv & R \\
x{:}P{\cdot}y{:}S & \equiv & y{:}S{\cdot}x{:}P \quad \textit{if } x \neq y & P & \equiv & P\{()\} \\
 & & & P\{R\} & \equiv & P\{R\} \mid \top
\end{array}
$$

Type Inference Rules

$$\frac{}{\vdash () :: ()}\ \textit{Empty} \qquad \frac{}{\vdash x :: \tau\{x{:}\tau\}}\ \textit{Lookup} \qquad \frac{\vdash E :: P\{R\} \mid C}{\vdash x{=}E :: x{:}P\{R\} \mid C}\ \textit{Label}$$

$$\frac{\vdash E :: P\{R\} \mid C_1 \qquad \vdash F :: S\{Q\} \mid C_2}{\vdash E{\cdot}F :: P{\cdot}S\{R \wedge Q\} \mid C_1, C_2}\ \textit{Extend}$$

$$\frac{\vdash E :: P\{R\} \mid C}{\vdash \lambda x.E :: R_x{\to}P\{R\backslash x\} \mid C}\ \textit{Abstract}$$

$$\frac{\vdash E :: P\{R\} \mid C_1 \qquad \vdash F :: S\{Q\} \mid C_2}{\vdash E; F :: S\{R\} \mid C_1, C_2,\ P \hookrightarrow Q}\ \textit{Close}$$

$$\frac{\vdash E :: P\{R\} \mid C_1 \qquad \vdash F :: S\{Q\} \mid C_2}{\vdash E\ F :: \beta\{R \wedge Q\} \mid C_1, C_2,\ P \hookrightarrow S{\to}\beta}\ \textit{Apply}$$

Type operators for required types

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}}
()_x & = & () & ()\backslash x & = & () \\
(x{:}R)_x & = & R & (x{:}R)\backslash x & = & () \\
(x{:}R)_y & = & () \quad x{\neq}y & (x{:}R)\backslash y & = & (x{:}R) \quad x{\neq}y \\
(Q \wedge R)_x & = & Q_x \wedge R_x & (Q \wedge R)\backslash x & = & Q\backslash x \wedge R\backslash x \\
\alpha_x & = & \bot & \alpha\backslash x & = & \bot \\
(P{\to}R)_x & = & \bot & (P{\to}R)\backslash x & = & \bot
\end{array}
$$

Table 2: Contractual Types — Syntax and Semantics

Equivalences for type expressions largely mimic those for form expressions. Only unbound type variables do not commute under extension, so, in general, $\tau{\cdot}\sigma \neq \sigma{\cdot}\tau$. ($\tau$ might later be bound to $x{:}()$ and σ might be bound to $x{:}y{:}()$, either of which could override the other.)

3.1 Type Inference

The type *inference* rules are used to infer a judgment $\vdash F :: P\{R\} \mid C$ for a form expression F. The task of checking whether the constraints in C can be satisfied is deferred to the type checking rules (section 3.2).

The rules *Empty*, *Lookup*, *Label* directly build up contractual types for empty forms, labels, and bindings. *Extend* overrides provided types and merges required types. *Abstract* makes use of type operators to lookup and remove requirements on specific labels from required types. The rules *Close* and *Apply* generate constraints to check if the provided environment P or argument satisfies its requirements R.

Let us consider each of the type inference rules in turn. The rule *Empty* is straightforward. The empty form has the empty type.

The *Lookup* rule is less obvious. The judgment $x :: \tau\{x{:}\tau\}$ states that a form x provides τ if it is bound in an environment where its type is τ. If we combine this with the *Label* rule, we see that a binding $x{=}y$ has type $x{:}\sigma\{y{:}\sigma\}$:

$$\frac{\overline{\vdash y :: \sigma\{y{:}\sigma\}}}{\vdash x{=}y :: x{:}\sigma\{y{:}\sigma\}}$$

This expresses precisely what we intend: the form $x{=}y$ provides something ($x{:}\sigma$), as long as the environment satisfies the requirement $y{:}\sigma$.

The *Extend* rule forms the logical conjunction of two requirements.

$$\frac{\overline{\vdash x :: \tau\{x{:}\tau\}} \qquad \overline{\vdash y :: \sigma\{y{:}\sigma\}}}{\vdash x{\cdot}y :: \tau{\cdot}\sigma\{x{:}\tau \wedge y{:}\sigma\}}$$

This expresses clearly that $x{\cdot}y$ provides $\tau{\cdot}\sigma$ as long as the environment satisfies $x{:}\tau$ *and* $y{:}\sigma$.

The *Abstract* rule assumes that the formal parameter x may occur free in the body E. It then extracts the requirement on x from R, the requirement E places on the environment. Since $\lambda x.E$ binds x in E, the requirement on x must be removed from R. The identity function is closed, so we expect its required type to be empty:

$$\frac{\overline{\vdash x :: \tau\{x{:}\tau\}}}{\vdash \lambda x.x :: \tau{\to}\tau}\,(x{:}\tau)_x = \tau, (x{:}\tau)\backslash x = ()$$

Consider, on the other hand, the function $\lambda x.y$. Since it is open, we expect it to have a non-empty required type:

$$\frac{\overline{\vdash y :: \sigma\{y{:}\sigma\}}}{\vdash \lambda x.y :: ()\to\sigma\{y{:}\sigma\}}\,(y{:}\sigma)_x = (), (y{:}\sigma)\backslash x = y{:}\sigma$$

The resulting type says that we place no requirement on the argument x, but the environment has the requirement $y{:}\sigma$.

The remaining rules, *Close* and *Apply*, generate constraints to be checked by the type-checking rules. A sandbox expression $E; F$ provides whatever F provides, and requires whatever E requires. *Close* additionally imposes the constraint that the type P provided by E must satisfy the requirement Q of F, *i.e.*, $P \hookrightarrow Q$.

Consider, for example, that $x :: \tau\{x{:}\tau\}$ and $y :: \sigma\{y{:}\sigma\}$:

$$\frac{\vdash x :: \tau\{x{:}\tau\} \qquad \vdash y :: \sigma\{y{:}\sigma\}}{\vdash x; y :: \sigma\{x{:}\tau\} \mid \tau \hookrightarrow y{:}\sigma}$$

Here we see that $x; y$ provides type σ (the type of y) if it is evaluated in an environment where x has type τ, given the constraint that τ satisfies $y{:}\sigma$. As we shall see below, this constraint can be trivially satisfied, but this is not always the case.

The *Apply* rule merges the requirements on the environment of E and F, and furthermore checks that E satisfies the requirement that it be an arrow type accepting the type of F. Consider the open expression $x\ y$:

$$\frac{\vdash x :: \tau\{x{:}\tau\} \qquad \vdash y :: \sigma\{y{:}\sigma\}}{\vdash x\ y :: \beta\{x{:}\tau \wedge y{:}\sigma\} \mid \tau \hookrightarrow \sigma{\rightarrow}\beta}$$

The resulting type expresses that $x\ y$ provides β, requires $x{:}\tau$ and $y{:}\sigma$, and τ must satisfy $\sigma{\rightarrow}\beta$.

As we shall see, these constraints are often trivial to satisfy, but may sometimes lead to difficulties.

3.2 Type Checking

We distinguish between the type *inference* rules, which are used to infer a judgment $\vdash F :: P\{R\} \mid C$ for a form expression F, and the type *checking* rules which attempt to satisfy the constraints in C and produce a judgment of the form $\vdash F :: P\{R\}$, or alternatively fail and produce a judgment $\vdash F :: \bot$. If type checking succeeds, then we know that F can be safely evaluated in any environment that satisfies its required type R and not lead to an error. The type checking rules are partial in the sense that they do not always succeed in either satisfying C or demonstrating that C cannot be satisfied.

Type checking proceeds as follows:

- First, type constraints are simplified using the equivalences in table 3 to constraints of the form $\alpha^{+} \hookrightarrow R$ or $P \hookrightarrow \alpha^{-}$.
- Next, constraints over the same positive type variable are coalesced into a single constraint using the *Join* equivalence.
- Then, constraints are resolved by unifying type variables with type expressions using the rules *Bind right* and *Bind left*.
- Finally, iterate until all constraints are resolved ($\top$), are shown to be unresolvable ($\bot$), or no further substitutions are possible.

Equivalence of type constraints

$$
\begin{array}{rcll}
P \hookrightarrow () & \equiv & \top & \textit{Trivial} \\
P \hookrightarrow Q \wedge R & \equiv & P \hookrightarrow Q, P \hookrightarrow R & \textit{Split} \\
P{\cdot}x{:}S \hookrightarrow x{:}R & \equiv & S \hookrightarrow R & \textit{Resolve label} \\
P{\cdot}y{:}S \hookrightarrow x{:}R & \equiv & P \hookrightarrow x{:}R \quad x \neq y & \textit{Skip label} \\
() \hookrightarrow x{:}R & \equiv & \bot & \textit{No label 1} \\
Q{\rightarrow}S \hookrightarrow x{:}R & \equiv & \bot & \textit{No label 2} \\
S'{\cdot}(Q{\rightarrow}S) \hookrightarrow P{\rightarrow}R & \equiv & P \hookrightarrow Q, S \hookrightarrow R & \textit{Resolve service} \\
S'{\cdot}x{:}S \hookrightarrow P{\rightarrow}R & \equiv & S' \hookrightarrow R{\rightarrow}R & \textit{Skip label 2} \\
() \hookrightarrow R{\rightarrow}R & \equiv & \bot & \textit{No service} \\
\alpha \hookrightarrow R, \alpha \hookrightarrow Q & \equiv & \alpha \hookrightarrow R \wedge Q & \textit{Join}
\end{array}
$$

Type checking rules

$$\frac{\vdash F :: P\{R\} \mid C, \alpha^+ \hookrightarrow Q}{\vdash F :: (P\{R\} \mid C)[\alpha^- \mapsto Q]}\ \alpha^+ \text{ does not occur in } C \qquad \textit{Bind left}$$

$$\frac{\vdash F :: P\{R\} \mid C, S \hookrightarrow \alpha^-}{\vdash F :: (P\{R\} \mid C)[\alpha^+ \mapsto S]}\ \alpha^- \text{ does not occur in } C \qquad \textit{Bind right}$$

Table 3: Type-checking Rules

Substitution of type variables $T[\alpha^- \mapsto R]$ and $T[\alpha^+ \mapsto P]$ is defined in the obvious way, and we skip the formal definition.

Let us briefly consider the last example of section 3.1 to see how the constraint may be resolved.

$$\frac{\vdash x; y :: \sigma^+\{x{:}\tau^-\} \mid \tau^+ \hookrightarrow y{:}\sigma^-}{\vdash x; y :: \sigma^+\{x{:}y{:}\sigma^-\}}\ \textit{Bind left}$$

Here we trivially satisfy the constraint $\tau \hookrightarrow y{:}\sigma$ by substituting $[\tau \mapsto y{:}\sigma]$.

In the next example we drop the superscripts:

$$\frac{\vdash x\ y :: \beta\{x{:}\tau \wedge y{:}\sigma\} \mid \tau \hookrightarrow \sigma{\rightarrow}\beta}{x\ y :: \beta\{x{:}\sigma{\rightarrow}\beta \wedge y{:}\sigma\}}\ \textit{Bind right}$$

Again, we see that the constraint is trivially satisfied by binding $[\tau^- \mapsto \sigma^+{\rightarrow}\beta^-]$. The resulting judgment clearly states that $x\ y$ is valid in an environment where x is bound to a service and y is a suitable argument.

4 Discussion

The key feature of contractual types is that one may infer types not only for closed expressions, but also for open expressions.

$$x\ y :: \beta\{x{:}\sigma{\rightarrow}\beta \wedge y{:}\sigma\}$$

is an open expression that must be placed in an environment providing suitable bindings for x and y.

Of course we may close such an expression as follows:

$$\dfrac{\dfrac{\cdots}{\vdash \lambda x.\lambda y.x\ y :: \beta\rightarrow\gamma\rightarrow\alpha \mid \beta\hookrightarrow\gamma\rightarrow\alpha}}{\lambda x.\lambda y.x\ y :: (\gamma\rightarrow\alpha)\rightarrow\gamma\rightarrow\alpha}$$

but this fails to capture the idea that such an expression will be evaluated when it is composed with other components providing the missing bindings for x and y.

A better way of modeling this is as follows:

$$\dfrac{\dfrac{\cdots}{\vdash \lambda e.(e; x\ y) :: \alpha\rightarrow\beta \mid \gamma\hookrightarrow\delta\rightarrow\beta, \alpha\hookrightarrow x{:}\gamma\wedge y{:}\delta}}{\lambda e.(e; x\ y) :: (x{:}(\delta\rightarrow\beta)\wedge y{:}\delta)\rightarrow\beta}$$

This captures the idea that $x\ y$ will be evaluated in some environment e which is required to provide the missing bindings.

It should be clear that any open expression can be closed in the same way, so there is a natural relationship between contractual types, and ordinary functional types. It is easy to demonstrate that if $F :: P\{R\}$ then $\lambda e.(e; F) :: R\rightarrow P$

4.1 Erroneous terms

Errors may occur in the form calculus either when we attempt to lookup a non-existing label, or when we apply a non-service to a form. Let us consider the two canonical cases.

First, consider invalid lookup:

$$\vdash (); x :: \tau \mid () \hookrightarrow x{:}\tau$$

Since () cannot possibly satisfy $x{:}\tau$, we conclude that the term cannot be typed, and is therefore erroneous.

Next, consider invalid application:

$$\vdash ()\ () :: \beta \mid () \hookrightarrow ()\rightarrow\beta$$

Since () cannot possibly satisfy $()\rightarrow\beta$, we again conclude that this term cannot be typed, and is therefore erroneous.

We can similarly deduce that the example of section 2.1 is erroneous:

$$\vdash x{=}a{=}()\cdot f{=}\lambda y.(y; b); f\ x :: ()\hookrightarrow b{:}\gamma$$

Let us now reconsider the example of section 2.1:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\vdash x{=}a{=}()\cdot f{=}\lambda y.(y;b) :: x{:}a{:}()\cdot f{:}(b{:}\beta\to\beta)} \quad \dfrac{\dots}{\vdash f\ x :: \alpha\{f{:}(\gamma\to\alpha)\wedge x{:}\gamma\}}}{\vdash x{=}a{=}()\cdot f{=}\lambda y.(y;b);f\ x :: \gamma \mid \alpha\hookleftarrow b{:}\beta, \delta\hookleftarrow\eta\to\gamma, x{:}a{:}()\cdot f{:}(\alpha\to\beta)\hookleftarrow f{:}\delta\wedge x{:}\eta}}{\vdash \dots :: \gamma \mid \alpha\hookleftarrow b{:}\beta, \delta\hookleftarrow\eta\to\gamma, \alpha\to\beta\hookleftarrow\delta, a{:}()\hookleftarrow\eta}}{\vdash \dots :: \gamma \mid \delta\hookleftarrow\eta\to\gamma, b{:}\beta\to\beta\hookleftarrow\delta, a{:}()\hookleftarrow\eta}}{\vdash \dots :: \gamma \mid \eta\hookleftarrow b{:}\beta, \beta\hookleftarrow\gamma, a{:}()\hookleftarrow\eta}}{\vdash \dots :: \gamma \mid \beta\hookleftarrow\gamma, ()\hookleftarrow b{:}\beta}}{\vdash \dots :: \gamma \mid ()\hookleftarrow b{:}\gamma}}{\vdash \dots :: \bot}$$

4.2 Contractual Types for Open Modules

Let us reconsider our motivating example from the introduction, which we encoded in the form calculus as:

$$\mathsf{com};\mathsf{dohickeys};\mathsf{gadget}{=}\lambda x.(\mathsf{widget}\ ()\cdot\mathsf{tinker}{=}\lambda w.(\mathsf{widget}\ ();\mathsf{munge})\ (\mathsf{thing}\ w))$$

We can easily infer the following type for this expression:

$$\begin{array}{l}\mathsf{gadget}{:}((){\to}(\gamma\cdot\mathsf{tinker}{:}(\beta{\to}\eta))\{\\ \quad \mathsf{com}{:}\mathsf{dohickeys}{:}((\mathsf{widget}{:}((){\to}\gamma)\wedge\mathsf{widget}{:}((){\to}\mathsf{munge}{:}(\zeta{\to}\eta)))\wedge\mathsf{thing}{:}(\beta{\to}\zeta))\\ \}\end{array}$$

which is close to, if not entirely identical to, the intuitive type we derived.

We may try to factor out the duplicated instantiation of widget as follows:

$$\begin{array}{l}\mathsf{com};\mathsf{dohickeys};\mathsf{gadget}{=}\lambda x.(\\ \quad \mathsf{super}{=}\mathsf{widget}\ ();\\ \quad \mathsf{super}\cdot\mathsf{tinker}{=}\lambda w.(\mathsf{super};\mathsf{munge})\ (\mathsf{thing}\ w)\\ ) :: \bot\end{array}$$

This, however, generates the constraint $()\hookleftarrow\mathsf{thing}{:}(\beta{\to}\zeta)$, which is clearly erroneous. The problem is that thing is not bound in the environment super=widget ().

We can fix our error as follows:

$$\begin{array}{l}\mathsf{com};\mathsf{dohickeys};\mathsf{gadget}{=}\lambda x.(\\ \quad \mathsf{thing}{=}\mathsf{thing}\cdot\mathsf{super}{=}\mathsf{widget}\ ();\\ \quad \mathsf{super}\cdot\mathsf{tinker}{=}\lambda w.(\mathsf{super};\mathsf{munge})\ (\mathsf{thing}\ w)\\ )\end{array}$$

which extracts the correct binding for thing from the enclosing environment dohickeys and generates the contractual type:

$$\begin{array}{l}\mathsf{gadget}{:}((){\to}(\tau\cdot\mathsf{tinker}{:}(\gamma{\to}\sigma))\{\\ \quad \mathsf{com}{:}\mathsf{dohickeys}{:}(\mathsf{widget}{:}((){\to}(\tau\wedge\mathsf{munge}{:}(\alpha{\to}\sigma))))\wedge\mathsf{thing}{:}(\gamma{\to}\alpha)\\ \}\end{array}$$

which arguably expresses what we set out to demonstrate.

4.3 Difficulties

Although we can infer types for any open or closed form expression, we are not always able to typecheck all the generated constraints. The simplest example that illustrates the problem is this one:

$$\vdash x{\cdot}y; z :: \gamma\{x{:}\alpha \wedge y{:}\beta\} \mid \alpha{\cdot}\beta \hookleftarrow z{:}\gamma$$

z is to be evaluated in the environment obtained by composing x and y. But since we have neither x nor y, we do not know which of the two will actually provide a binding for z. This is expressed by the constraint $\alpha{\cdot}\beta \hookleftarrow z{:}\gamma$.

Closing this expression does not help in any way:

$$\vdash \lambda x.\lambda y.(x{\cdot}y; z) :: \alpha \rightarrow \beta \rightarrow \gamma \mid \alpha{\cdot}\beta \hookleftarrow z{:}\gamma$$

We conclude that certain constraints (and this is by far the simplest example) cannot be resolved without more information.

A very different question is to ask whether *any* solution exists for a set of constraints. A complete type-checking algorithm would generate at least one solution that satisfies a set of constraints, or demonstrate that no solution exists. Our current type checking rules are far from achieving this.

A second problem is that we have not addressed the issue of encoding recursively defined services. It turns out that the usual fixed-point combinator can also be used effectively in the form calculus:

$$\mathsf{fix}{=}\lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y))$$

The type we infer for this expression, however, generates a nasty set of constraints that cannot be easily interpreted.

$$\begin{aligned} \vdash \cdots :: \quad & fix{:}(((\phi \rightarrow \delta) \rightarrow \beta \wedge (\tau \rightarrow \gamma) \rightarrow \zeta) \rightarrow \alpha) \\ & \mid \zeta \hookleftarrow \phi \rightarrow \delta, \eta \hookleftarrow \eta \rightarrow \tau \rightarrow \gamma, \eta \hookleftarrow \eta, \zeta \hookleftarrow \tau \rightarrow \gamma, (\eta \rightarrow \tau \rightarrow \gamma \wedge \eta) \rightarrow \zeta \hookleftarrow \eta, \beta \hookleftarrow \alpha \end{aligned}$$

The problem seems to arise from the (necessary) presence of self application in the fixed-point combinator, a traditional stumbling point for type systems. We suspect that a better approach would be to add an explicit fixed-point operator to the form calculus, and a corresponding notion of recursive types to the type system.

5 Related work

Other researchers have proposed type systems that explicitly document both provided and required services. Shao and Appel infer constraints on environments in order to achieve *smartest recompilations* for ML modules. Jim has proposed *principal typings* as a general approach to inferring constraints on environments for open expressions. Neither approach introduces first-class environments, but instead generates a *type environment* expressing the type assumptions. We conjecture that principal typings exist for the form calculus.

More recently, *Collaboration interfaces* [15] express *provided* and *expected* contracts for a software component, and enable components to be flexibly bound to services they require. Zenger [23] has proposed a type-safe prototype-based model for component composition as

an extension to Featherwight Java with *provides* and *requires* declarations. Numerous architectural description languages (ADLs) [21], such as *ArchJava* [5] and *Jiazzi*, [14] also model components as explicitly providing and requiring services, typically through named ports.

The inspiration for this work is the Piccola composition language [4] [3], whose formal semantics are defined in terms of forms, agents and channels. Lumpe's thesis [13] presents the $\pi\mathcal{L}$-calculus — a variant of the π calculus based on forms — together with its type system. This simple type system, however, is very restrictive, and does not support polymorphism. Schneider's thesis [20] introduces a "form calculus" that includes agents and channels, but does not treat environments as first-class. Achermann's "Piccola calculus" [2] also includes agents and channels, and can be seen as containing the pure form calculus presented here as a sublanguage. Neither Schneider's form calculus nor the Piccola calculus are typed.

Abadi's calculus of explicit substitutions [1] makes environment explicit, but does not turn them into first-class values. Dami's calculus of names, λN[8], replaces variables by names, and has a similar flavour to our form calculus, but still does not treat environments as first-class values.

Nishizaki's *environment calculus* [18] treats environments as first-class values, and supports a polymorphic, ML-style type system with a corresponding type-inference algorithm. Although the environment calculus is superficially very similar to the form calculus, it skirts the issue of when expressions are open or closed.

Sato's $\lambda\epsilon$ calculus [19] bears remarkable resemblance the form calculus, but it is a typed calculus requiring explicit type declarations, rather than an untyped calculus for which type inference is defined. Although $\lambda\epsilon$ offers a definition of free names and open expressions, it is strongly dependent on the type system. In particular, not all free names of an expression need be captured by the environment in which it is evaluated. Unlike Nishizaki's environment calculus, $\lambda\epsilon$ only supports a simple type system.

Aside from Piccola, various other languages have supported explicit environments. Scheme [9] is the best-known, and is dynamically typed. Pebble [6] is the other well-known example, and is statically typed. Symmetric Lisp [10] is dynamically typed, and was developed specifically with parallel applications in mind. Quest [7] was inspired by Pebble, and supports explicit import and export of first-class modules.

The "contracts" expressed by contractual types are technically very weak, and do not capture behavioural contracts, such as pre- or post-conditions to services, or protocols over sets of services. *Regular types* [16] were an early attempt to express service protocols as finite state processes, where protocol satisfaction was determined by deadlock freeness. It would be interesting to see if contractual types could be extended to capture some degree of behavioural contracts.

6 Ongoing and Future Work

We have introduced the *form calculus* as a platform for studying composition of software components, and we have proposed *contractual types* as a means to express what a software component not only provides, but also what it requires. We have presented a type system for inferring contractual types, together with the type satisfaction constraints over free type variables. Although the type system will infer types for arbitrary open or closed form expressions, the type checking rules can only validate certain kinds of constraints. Other constraints can only be checked when the missing environments are available.

Although we have presented an operational semantics for the form calculus, a number of important properties remain to be shown, such as confluence and faithfulness to the call-by-value lambda calculus. We similarly conjecture, without proof, soundness and a subject reduction theorem for contractual types.

We have developed an experimental prototype of the form calculus and its type system, and the derivations shown here have been generated from this implementation. The semantics given correspond closely to the rules of the Prolog implementation. A first attempt has been made to apply contractual types to Piccola, where provided and required types can be used to effectively capture the "plugs" of software components [12]. Even though Piccola is a small language, there are several important extensions needed to the form calculus to adequately express the type issues at stake, in particular, form introspection, recursive services, and root, the mechanism Piccola offers to capture the current environment.

The next step would be to apply the type system to other more mainstream languages, like Java or Smalltalk. This would require a translation of the compositional aspects of these languages to the form calculus. Although concurrency and side effects can largely be ignored for type inference and type-checking, reflective aspects of these languages may pose further difficulties.

Acknowledgments We would like to thank Andrew Black, Luca Cardelli, Gerhard Jäger, Stefan Kneubühl, Mathis Kretz, Peng Liang, Tom Mens, Roel Wuyts, and Matthias Zenger for comments and suggestions, as well as anonymous reviewers who offered corrections and pointers to related work.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. Technical Report 54, DEC Systems Research Center, Palo Alto, California, February 1990.

[2] Franz Achermann. *Forms, Agents and Channels - Defining Composition Abstraction with Style*. PhD thesis, University of Berne, January 2002.

[3] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[4] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[5] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain, June 2002. Springer Verlag.

[6] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. *Information and Computation*, 76(2/3), 1984. Also appeared in Proceedings of the

International Symposium on Semantics of Data Types, Springer, LNCS (1984), and as SRC Research Report 1.

[7] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State of the Art Reports Series, pages 431–507. Springer-Verlag, 1991.

[8] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, February 1998.

[9] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, 1987.

[10] David Gelernter, Suresh Jagannathan, and Tom London. Environments as first-class objects. In *Principles of Programming Languages*. ACM, 1987.

[11] Trevor Jim. What are principal typings and what are they good for? In *Principles of Programming Languages*. ACM, 1996.

[12] Stefan Kneubuehl. Typeful compositional styles. Diploma thesis, University of Bern, April 2003.

[13] Markus Lumpe. *A Pi-Calculus Based Approach to Software Composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[14] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: New age components for old fashioned java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, October 2001.

[15] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA 2002*, pages 52–67, November 2002.

[16] Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 1–15, October 1993.

[17] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall, 1995.

[18] Shin-ya Nishizaki. A polymorphic environment calculus and its type-inference algorithm. *Higher-Order and Symbolic Computation*, 13(3):241–280, 2000.

[19] Masahiko Sato, Takafumi Sakurai, and Rod M. Burstall. Explicit environments. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *LNCS*, pages 340–354, L'Aquila, Italy, April 1999. Springer-Verlag.

[20] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[21] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[22] Clemens A. Szyperski. *Component Software.* Addison Wesley, 1998.

[23] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 470–497, Malaga, Spain, June 2002. Springer Verlag.

Appendix—Constraint unification for contractual types

Unification

Here we look at the examples in some detail, indicating in each case the *parity* of type variables, according to whether they occur in positive (provided) or negative (required) positions. For each example, we give first the plain inferred type. Then, if necessary, this type is *normalized* so that each constraint in the constraint set contains a single type variable either on the left ($\alpha \hookrightarrow R$) or on the right hand side ($P \hookrightarrow \alpha$). Note that, in general, if a variable appears positively or negatively in a constraint, it appears, respectively, negatively or positively somewhere else, either in the type or in the constraint set. After normalization, a single unification step is shown, in which either a RHS or LHS variable is unified with the provided or required type that constrains it. Constraints on positive variables are merged into a type conjunction. Negative variables are only unified if they occur uniquely.

To ease readability, the final type expression shown erases the parity. All the type expressions here have been generated by the nanola formtypes package.

Simple substitutions

In these examples, the type variable is simply replaced by its requirement.

$$
\begin{array}{rcl}
x; y & :: & \beta^+\{x{:}\alpha^-\}[\alpha^+ \hookrightarrow y{:}\beta^-] \\
 & \rightarrow & \beta^+\{x{:}y{:}\beta^-\} \\
 & = & \beta\{x{:}y{:}\beta\} \\
x\ y & :: & \alpha^+\{x{:}\beta^- \wedge y{:}\gamma^-\}[\beta^+ \hookrightarrow \gamma^+ \rightarrow \alpha^-] \\
 & \rightarrow & \alpha^+\{x{:}(\gamma^+ \rightarrow \alpha^-) \wedge y{:}\gamma^-\} \\
 & = & \alpha\{x{:}(\gamma \rightarrow \alpha) \wedge y{:}\gamma\} \\
\lambda x.(x; y) & :: & \alpha^- \rightarrow \beta^+[\alpha^+ \hookrightarrow y{:}\beta^-] \\
 & \rightarrow & y{:}\beta^- \rightarrow \beta^+ \\
 & = & y{:}\beta \rightarrow \beta \\
\lambda x.\lambda y.(x; y) & :: & \alpha^- \rightarrow () \rightarrow \beta^+[\alpha^+ \hookrightarrow y{:}\beta^-] \\
 & \rightarrow & y{:}\beta^- \rightarrow () \rightarrow \beta^+ \\
 & = & y{:}\beta \rightarrow () \rightarrow \beta \\
x\ x & :: & \alpha^+\{x{:}\beta^- \wedge x{:}\gamma^-\}[\beta^+ \hookrightarrow \gamma^+ \rightarrow \alpha^-] \\
 & \rightarrow & \alpha^+\{x{:}(\gamma^+ \rightarrow \alpha^-) \wedge x{:}\gamma^-\} \\
 & = & \alpha\{x{:}(\gamma \rightarrow \alpha) \wedge x{:}\gamma\}
\end{array}
$$

Positive and negative constraints

The first example here is interesting because α occurs twice. We must rewrite the constraint as $\alpha^+ \hookrightarrow f{:}\gamma \wedge x{:}\delta$ to perform the substitution, i.e., "undoing" the type decomposition.

$$\begin{array}{rcl}
\lambda e.(e; f\ x) & :: & \alpha^- \to \beta^+ [\gamma^+ \hookrightarrow \delta^+ \to \beta^-, \alpha^+ \hookrightarrow f{:}\gamma^- \wedge x{:}\delta^-] \\
& = & \alpha^- \to \beta^+ [\gamma^+ \hookrightarrow \delta^+ \to \beta^-, \alpha^+ \hookrightarrow f{:}\gamma^-, \alpha^+ \hookrightarrow x{:}\delta^-] \\
& \to & \alpha^- \to \beta^+ [\alpha^+ \hookrightarrow f{:}(\delta^+ \to \beta^-), \alpha^+ \hookrightarrow x{:}\delta^-] \\
& \to & (f{:}(\delta^+ \to \beta^-) \wedge x{:}\delta^-) \to \beta^+ \\
& = & (f{:}(\delta \to \beta) \wedge x{:}\delta) \to \beta
\end{array}$$

Note that we could also resolve the constraint by first binding α, which suggests that the substitution order does not matter.

In the next two examples as well, substitution order does not matter:

$$\begin{array}{rcl}
(\lambda x.x)\ (\lambda x.x) & :: & \alpha^+ [\beta^- \to \beta^+ \hookrightarrow (\gamma^- \to \gamma^+) \to \alpha^-] \\
& = & \alpha^+ [\gamma^- \to \gamma^+ \hookrightarrow \beta^-, \beta^+ \hookrightarrow \alpha^-] \\
& \to & \alpha^+ [\gamma^- \to \gamma^+ \hookrightarrow \alpha^-] \\
& \to & \gamma^- \to \gamma^+ \\
& = & \gamma \to \gamma
\end{array}$$

Here we have an example where a unification gives rise to the need for normalization.

$$\begin{array}{rcl}
(\lambda x.(x; b))\ (a{=}() \cdot b{=}()) & :: & \alpha^+ [\beta^+ \hookrightarrow b{:}\gamma^-, \beta^- \to \gamma^+ \hookrightarrow (a{:}() \cdot b{:}()) \to \alpha^-] \\
& = & \alpha^+ [\beta^+ \hookrightarrow b{:}\gamma^-, a{:}() \cdot b{:}() \hookrightarrow \beta^-, \gamma^+ \hookrightarrow \alpha^-] \\
& \to & \alpha^+ [a{:}() \cdot b{:}() \hookrightarrow b{:}\gamma^-, \gamma^+ \hookrightarrow \alpha^-] \\
& = & \alpha^+ [() \hookrightarrow \gamma^-, \gamma^+ \hookrightarrow \alpha^-] \\
& \to & \alpha^+ [() \hookrightarrow \alpha^-] \\
& \to & ()
\end{array}$$

Again, various substitution orders do not make a difference:

$$\begin{array}{rcl}
(\lambda x.(x; b))\ z & :: & \alpha^+ \{z{:}\delta^-\} [\beta^+ \hookrightarrow b{:}\gamma^-, \beta^- \to \gamma^+ \hookrightarrow \delta^+ \to \alpha^-] \\
& = & \alpha^+ \{z{:}\delta^-\} [\beta^+ \hookrightarrow b{:}\gamma^-, \delta^+ \hookrightarrow \beta^-, \gamma^+ \hookrightarrow \alpha^-] \\
& \to & \alpha^+ \{z{:}\delta^-\} [\delta^+ \hookrightarrow b{:}\gamma^-, \gamma^+ \hookrightarrow \alpha^-] \\
& \to & \alpha^+ \{z{:}b{:}\gamma^-\} [\gamma^+ \hookrightarrow \alpha^-] \\
& \to & \alpha^+ \{z{:}b{:}\alpha^-\} \\
& = & \alpha \{z{:}b{:}\alpha\}
\end{array}$$

Erroneous terms are easily recognized, since their constraint sets cannot reduce to normal form ($[() \hookrightarrow x : \tau]$ and $[() \hookrightarrow () \to \beta]$).

$$\begin{array}{rcl}
(); x & :: & \alpha^+ [() \hookrightarrow x{:}\alpha^-] \\
& \to & \bot \\
()\ () & :: & \alpha^+ [() \hookrightarrow () \to \alpha^-] \\
& \to & \bot
\end{array}$$

Difficulties

Here is another case where unification leads to a complex constraint, which must then be normalized to make further progress. We reach an impasse, since it is not clear how we should unify α and β due to mixed parities.

$$\begin{aligned}
x{=}\lambda y.y; x\ x \quad &:: \quad \beta^+[\gamma^+ \hookrightarrow \delta^+ \rightarrow \beta^-, x{:}(\alpha^- \rightarrow \alpha^+) \hookrightarrow x{:}\gamma^- \wedge x{:}\delta^-] \\
&= \quad \beta^+[\gamma^+ \hookrightarrow \delta^+ \rightarrow \beta^-, \alpha^- \rightarrow \alpha^+ \hookrightarrow \gamma^-, \alpha^- \rightarrow \alpha^+ \hookrightarrow \delta^-] \\
&\rightarrow \quad \beta^+[\alpha^- \rightarrow \alpha^+ \hookrightarrow \delta^+ \rightarrow \beta^-, \alpha^- \rightarrow \alpha^+ \hookrightarrow \delta^-] \\
&= \quad \beta^+[\delta^+ \hookrightarrow \alpha^-, \alpha^+ \hookrightarrow \beta^-, \alpha^- \rightarrow \alpha^+ \hookrightarrow \delta^-] \\
&\rightarrow \quad \beta^+[\alpha^+ \hookrightarrow \beta^-, \alpha^- \rightarrow \alpha^+ \hookrightarrow \alpha^-] \\
&= \quad \beta[\alpha \hookrightarrow \beta, \alpha \rightarrow \alpha \hookrightarrow \alpha]
\end{aligned}$$

This expression seems to pose no problems:

$$\begin{aligned}
\lambda x.x\ x \quad &:: \quad (\beta^- \wedge \gamma^-) \rightarrow \alpha^+[\beta^+ \hookrightarrow \gamma^+ \rightarrow \alpha^-] \\
&\rightarrow \quad (\gamma^+ \rightarrow \alpha^- \wedge \gamma^-) \rightarrow \alpha^+ \\
&= \quad (\gamma \rightarrow \alpha \wedge \gamma) \rightarrow \alpha
\end{aligned}$$

Self-application of this expression (AKA Ω) seems to be untypable. In this example, neither σ nor η occur as unifiable variables in the initial normal form, but only during later normalization steps.

$$\begin{aligned}
& (\lambda x.x\ x)\ (\lambda x.x\ x) \\
:: \quad & \alpha^+[\gamma^+ \hookrightarrow \delta^+ \rightarrow \beta^-, \tau^+ \hookrightarrow \sigma^+ \rightarrow \eta^-, (\gamma^- \wedge \delta^-) \rightarrow \beta^+ \hookrightarrow ((\tau^- \wedge \sigma^-) \rightarrow \eta^+) \rightarrow \alpha^-] \\
= \quad & \alpha^+[\gamma^+ \hookrightarrow \delta^+ \rightarrow \beta^-, \tau^+ \hookrightarrow \sigma^+ \rightarrow \eta^-, (\tau^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \gamma^-, (\tau^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \delta^-, \beta^+ \hookrightarrow \alpha^-] \\
\rightarrow \quad & \alpha^+[\tau^+ \hookrightarrow \sigma^+ \rightarrow \eta^-, (\tau^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \delta^+ \rightarrow \beta^-, (\tau^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \delta^-, \beta^+ \hookrightarrow \alpha^-] \\
= \quad & \alpha^+[\tau^+ \hookrightarrow \sigma^+ \rightarrow \eta^-, \delta^+ \hookrightarrow \tau^-, \delta^+ \hookrightarrow \sigma^-, \eta^+ \hookrightarrow \beta^-, (\tau^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \delta^-, \beta^+ \hookrightarrow \alpha^-] \\
\rightarrow \quad & \alpha^+[\delta^+ \hookrightarrow \sigma^+ \rightarrow \eta^-, \delta^+ \hookrightarrow \sigma^-, \eta^+ \hookrightarrow \beta^-, (\sigma^+ \rightarrow \eta^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \delta^-, \beta^+ \hookrightarrow \alpha^-] \\
\rightarrow \quad & \alpha^+[\eta^+ \hookrightarrow \beta^-, (\sigma^+ \rightarrow \eta^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \sigma^+ \rightarrow \eta^- \wedge \sigma^-, \beta^+ \hookrightarrow \alpha^-] \\
= \quad & \alpha^+[\eta^+ \hookrightarrow \beta^-, \sigma^+ \hookrightarrow \sigma^+ \rightarrow \eta^-, \sigma^+ \hookrightarrow \sigma^-, \eta^+ \hookrightarrow \eta^-, (\sigma^+ \rightarrow \eta^- \wedge \sigma^-) \rightarrow \eta^+ \hookrightarrow \sigma^-, \beta^+ \hookrightarrow \alpha^-] \\
= \quad & \alpha[\eta \hookrightarrow \beta, \sigma \hookrightarrow \sigma \rightarrow \eta, \sigma \hookrightarrow \sigma, \eta \hookrightarrow \eta, (\sigma \rightarrow \eta \wedge \sigma) \rightarrow \eta \hookrightarrow \sigma, \beta \hookrightarrow \alpha]
\end{aligned}$$

At this point we are stuck, and no further progress seems to be possible. Negative variables occur multiply on the RHS of constraints, and other strange constraints appear. In the usual typed polymorphic lambda calculus, only terminating expressions are typable. It is not clear what should be the case here.