

# Type Inference for First-Class Messages with *Match-Functions*

Paritosh Shroff Scott F. Smith

Johns Hopkins University  
{*pari, scott*}@cs.jhu.edu

## Abstract

Messages that can be treated as first-class entities are called *first-class messages*. We present a sound unification-based type inference system for first-class messages. The main contribution of the paper is the introduction of an extended form of function called a *match-function* to type first-class messages. Match-functions can be given simple dependent types: the return type depends on the type of the argument. We encode objects as match-functions and messages as polymorphic variants, thus reducing message-passing to simple function application. We feel the resulting system is significantly simpler than previous systems for typing first-class messages, and may reasonably be added to a language design without overly complicating the type system.

## Keywords

First-class messages, polymorphic variants, object-oriented programming, constraint-based type inference, unification, object-encoding.

## 1 Introduction

First-class messages are messages that can be treated as first-class values and bound to program variables. Hence, we can write  $obj \leftarrow x$ , where  $obj$  is an object,  $x$  a program variable and  $\leftarrow$  represents message passing. Since the message assigned to  $x$  can be varied at run-time, we can change the message sent to  $obj$  dynamically. This is the exact dual of *dynamic dispatch*, where we change the *object* at run-time—here we want to be able to change the *message* at runtime.

First-class messages are useful in typing delegate objects, which forward messages to other objects. For example, in

```
let m = fmessage () in
  let o1 = {forward(x) = o2 ← x} in
    o1 ← forward(m)
```

$o1$  is a delegate object where  $forward(x)$  is the method that delegates, *i.e.* forwards,  $x$  to  $o2$ .  $fmessage()$  is the first-class message which gets assigned to  $m$ , then  $x$ , and finally forwarded to  $o2$ . Such delegate objects are ubiquitous in distributed systems such as proxy servers giving access to remote services (e.g. ftp) beyond a firewall. The following is an example of a proxy server cited by Müller [MN00]:

```
let ProxyServer = {new(o) = {send(m) = o ← m}}
```

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

takes an abstract first-class message  $m$  as a argument and forwards it to  $o$ . We can create a Ftp proxy server as:

```
let FtpProxy = ProxyServer ← new(ftp)
```

where  $ftp$  is a *Ftp* object. A typical use of this new proxy is

```
FtpProxy ← send(get('paper.ps.gz'))
```

*get* is a method in *ftp*. Delegation of abstract messages cannot be easily expressed without first-class messages, since the message can be changed at run-time and hence must be abstracted by a variable  $m$ . [MN00] and [Nis98] further discuss how first-class messages can exploit locality by abstracting remote computations in distributed object-oriented computing.

Modern typed object-oriented programming languages (e.g. Java, C++) do not support first-class messages. Smalltalk [GR89] and Objective-C [PW91] provide untyped first-class messages.

The static type-checking of first-class messages presents two main difficulties:

1. Typing first-class messages that can be placed in variables and passed as values, *e.g.*  $m = fmessage()$ : we need a way to express and type standalone messages.
2. Typing abstract message-passing to objects, such as  $o \leftarrow x$ : since  $x$  can be any message in  $o$  and these different messages can have different return types,  $o \leftarrow x$  doesn't have a single return type but rather its type depends upon the type of  $x$ . We need to be able to encode its static type such that it depends on the abstract type of  $x$ .

We solve the first problem by encoding messages as polymorphic variants which have standalone types, *i.e.*, a polymorphic variant  $fmessage()$  has type  $fmessage()$ .

The second problem is more challenging, and the most important contribution of this paper is the simplicity of its solution. We introduce an extended form of function (inspired by SML/NJ's pattern matching functions) called a *match-function*, which always takes a variant as argument and has a return type that can depend on the argument variant. We call these dependent types *match-types*. We encode objects as match-functions, thus reducing message sending to simple function application. There are several other solutions to typing first-class messages in the literature [Wak93, Nis98, MN00, Pot00]; the main advantage of our approach is its simplicity.

We organize the rest of the paper as follows. We review OCaml's polymorphic variants in Section 1.1, and review the duality of

records and variants in Section 1.2. In Section 2 we define *DV*, a core language with match-functions. In Section 3 we present a type system for *DV* that includes match-types, and show it is sound. In Section 4 we present a constraint-based type inference system for inferring match-types along with an algorithm for simplifying the constrained types to human-readable constraint-free types and prove its soundness. The net result is a type inference algorithm for the original *DV* type system. Section 5 illustrates all aspects of the system with a series of examples. In Section 6 we show how objects with first-class messages can be faithfully encoded with match-functions. Section 7 discusses the related work.

Some portions of the paper are grayed out. They represent optional extensions to our system which enhance its expressiveness at the expense of added complexity. The paper can be read assuming the gray portions don't exist. We recommend the readers skip these grayed out portions during their initial readings to get a better understanding of the core system.

## 1.1 Review of Polymorphic Variants

Variant expressions and types are well known as a cornerstone of functional programming languages. For instance in OCaml we may declare a type as:

```
type feeling = Love of string | Hate of string
             | Happy | Depressed
```

Polymorphic variants [Gar98, Gar00], implemented as part of Objective Caml [LDG<sup>+</sup>02], allow inference of variant types, so type declarations like the above are not needed: we can directly write expressions like `'Hate("Fred")` or `'Happy`.

We use the Objective Caml [LDG<sup>+</sup>02] syntax for polymorphic variants in which each variant name is prefixed with `'`. For example in OCaml 3.07 we have,

```
let v = 'Hate ("Fred");;
val v : [> 'Hate of string] = 'Hate "Fred"
```

`[> 'Hate of string]` is the type of the polymorphic variant `'Hate ("Fred")`. The `>` at the left means the type is read as “these cases or *more*”. The “or more” part means these types are polymorphic, it can match with a type of more cases.

Correspondingly, pattern matching is also given a partially specified type. For example,

```
let f = fun 'Love s -> s | 'Hate s -> s
-: val f : [< 'Love of string | 'Hate of string] -> string
```

`[< 'Love of string | 'Hate of string]` is the inferred variant type. The `<` at the left means the type can be read “these cases or *less*”, and since `'Hate ("Fred")` has type `[> 'Hate of string]`, `f 'Hate ("Fred")` will typecheck.

Polymorphic variants are expressible without subtyping, and are thus easily incorporated into unification-based type inference algorithms. They can be viewed as a simplified version of [R89, Oho95].

Our type system incorporates a generalization of Garrigue’s notion of polymorphic variants that explicitly maps the variant coming in to a function to the variant going out. This generalization is useful in functional programming, but is particularly useful for us in that it allows objects with first-class messages to be expressed using only variants and matching, something that is not possible in any of the

existing polymorphic variant type systems above.

## 1.2 The Duality of Variants and Records

It is well-known that variants are duals of records in the same manner as logical “or” is dual to “and”. A variant is this field *or* that field *or* that field ...; a record is this field *and* that field *and* that field ... Since they are duals, defining a record is related to using a variant, and defining a variant is like using a record. In a programming analogy of DeMorgan’s Laws, variants can directly encode records and vice-versa.

A variant can be encoded using a record as follows:

$$\begin{aligned} \text{match } s \text{ with } 'n_1(x_1) \rightarrow e_1 \mid \dots \mid 'n_m(x_m) \rightarrow e_m &\equiv \\ s \{n_1 = \text{fun } x_1 \rightarrow e_1, \dots, n_m = \text{fun } x_m \rightarrow e_m\} & \\ 'n(e) \equiv (\text{fun } x \rightarrow (\text{fun } r \rightarrow r.n)x) e & \end{aligned}$$

Similarly, a record can be encoded in terms of variants as follows:

$$\begin{aligned} \{l_1 = e_1, \dots, l_m = e_m\} &\equiv \\ \text{fun } s \rightarrow \text{match } s \text{ with } 'l_1(x) \rightarrow e_1 \mid \dots \mid 'l_m(x) \rightarrow e_m & \end{aligned}$$

where  $x$  is any new variable. The corresponding selection encoding is:

$$e.l_k \equiv e 'l_k(-)$$

where `-` could be any value.

One interesting aspect about the duality between records and variants is that *both* records and variants can encode objects. Traditionally, objects have been understood by encoding them as records, but a variant encoding of objects also is possible: *A variant is a message, and an object is a case on the message*. In the variant encoding, a nice added side-effect is it is easy to pass around messages as *first-class entities*.

The problems with the above encodings, however, is neither is complete in the context of the type systems commonly used for records and variants: for example, if an ML variant is used to encode objects, all the “methods” (cases of the match) must return the same type! This is why objects are usually encoded as records. But if the variant encoding could be made to work, it would give first-class status to messages, something not possible in the record system.

In this paper we introduce *match-functions*, which are essentially ML-style pattern match functions, but match-functions in addition support different return types for different argument types via *match-types*. A match-function-encoding of objects is as powerful as a record encoding, but with additional advantage of allowing first-class messages to typecheck.

## 2 The DV Language

*DV* {“Diminutive” pure functional programming language with Polymorphic Variants} is the core language we study. The grammar is as follows:

$$\begin{aligned} \text{Name} &\ni n \\ \text{Val} &\ni v ::= x \mid i \mid 'n(v) \mid \overline{\lambda_f 'n_k(x_k) \rightarrow e_k} \\ \text{Exp} &\ni e ::= v \mid e \mid 'n(e) \mid \mathbf{let } x = e_1 \text{ in } e_2 \\ \text{Num} &\ni i ::= \dots -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots \end{aligned}$$

The “vector notation”  $\overline{'n_k(x_k) \rightarrow e_k}$  is shorthand for  $'n_1(x_1) \rightarrow e_1 \mid \dots \mid 'n_k(x_k) \rightarrow e_k$  for some  $k$ .  $'n(e)$  is a polymorphic variant with an argument  $e$ . For simplicity, variants take only a single argument here; multiple argument support can be easily added.  $\lambda_f \overline{'n_i(x_k) \rightarrow e_k}$

is an extended form of  $\lambda$ -abstraction, inspired by Standard ML style function definitions which also perform pattern matching on the argument type. The  $f$  in  $\lambda_f$  is the name of the function for use in recursion, as with `let rec`. We call these *match-functions*. Each match-function can also be thought of as *collection* of one or more (sub-)functions. For example, a match-function which checks whether a number is positive, negative or zero could be written as:

$$f = \lambda_f \begin{array}{l} \text{'positive}(x) \rightarrow (x > 0) \\ \text{'negative}(x) \rightarrow (x < 0) \\ \text{'zero}(x) \rightarrow (x == 0) \end{array}$$

and corresponding application would be:

$$f \text{ ('positive}(5))$$

where `'positive(5)` is the argument to the above match-function.

A match-function need not have a single return type, it can depend on the type of the argument. Thus in the above example, `'positive`, `'negative` and `'zero` could have had different return types. The main technical contribution of this paper is a simple type system for the static typechecking of match-functions.

Regular functions can be easily encoded using match-functions as:

$$\lambda_f x. e \equiv \lambda_f \text{'_}(x) \rightarrow e$$

where `'_` is a fixed polymorphic variant name; and corresponding application as:

$$f e' \equiv f \text{ ('_}(e'))$$

## 2.1 Operational Semantics

Figure 1 presents the operational semantics for *DV*. Computation is defined via a single-step relation  $\longrightarrow_1$  between closed expressions.  $e[v/x]$  is the substitution of  $v$  for  $x$  in  $e$ . The only interesting case is function application, which is a combined  $\beta$ -reduction and pattern-match.

## 3 The DV Types

The types of *DV* are as follows.

$$\begin{array}{l} \text{TyVar} \quad \ni \alpha ::= 'a \mid 'b \mid \dots \\ \text{TypVariant} \ni v ::= 'n(\tau) \mid [> \alpha] \mid [< \overline{n_k(\tau_k)}] \\ \text{Typ} \quad \ni \tau ::= \alpha \mid \text{Int} \mid v \mid \overline{n_k(\tau_k) \rightarrow \tau'_k} \mid \langle \overline{v_k \rightarrow \tau_k} \rangle \mid \tau(v) \mid \mu\tau. \tau' \end{array}$$

$'n(\tau)$  is a polymorphic variant type, for variant name  $'n$  with argument type  $\tau$ .  $[> \alpha]$  is a variant-type variable which represents any polymorphic variant type.  $[< \overline{n_k(\tau_k)}]$  is an ‘‘upper bound’’ polymorphic variant type, *i.e.*, it can match  $'n_1(\tau_1)$  or  $\dots$  or  $'n_k(\tau_k)$ . This type is a part of the optional extension (grayed out portions) to our type system and we recommend readers ignore this and all the following grayed portions during the initial readings.

Our main novelty is  $\overline{n_k(\tau_k) \rightarrow \tau'_k}$ , the *match-type*, in which each variant maps to a different return type.  $\langle \overline{v_k \rightarrow \tau_k} \rangle$  is a ‘‘lower bound’’ *match-type* *i.e.* it matches any *match-type* with at least  $\overline{v_k \rightarrow \tau_k}$  cases.

$\tau(v)$  is the *app-type*. The only forms in which it will appear in our system are as  $\overline{n_k(\tau_k) \rightarrow \tau'_k} ([> \alpha])$  or  $\overline{n_k(\tau_k) \rightarrow \tau'_k} ([< 'n_i(\tau_i) \mid \dots])$ . It is used for unknown first-class messages, *i.e.*, when the type of the argument to a *known* match-function is unknown at compile-time, and the return type is also unknown and depends upon the

value assigned to the argument at run-time. In such cases, the return type has the above type, which essentially means that at run-time  $[> \alpha]$  can be any of  $'n_i(\tau_i)$  and when  $[> \alpha]$  is  $'n_i(\tau_i)$  then the corresponding  $\tau'_i$  would be the return type.

$\mu\tau. \tau'$  is the *recursive-type*. It means  $\tau$  can occur in  $\tau'$  and also  $\tau$  has the same type as  $\tau'$ . The only forms in which it will occur in our system are  $\mu\alpha. \tau$  or  $\mu[> \alpha]. \tau$ .

We now define a type subsumption relation,

DEFINITION 3.1 (TYPE SUBSUMPTION).  $\tau_1 \succeq \tau_2$  *iff*,

1.  $\tau_1 = \tau$  and  $\tau_2 = \tau$ ; or
2.  $\tau_1 = \tau$  and  $\tau_2 = [\tau''/\tau']\tau$  where  $\tau' \succeq \tau''$ ; or
3.  $\tau_1 = [> \alpha]$  and  $\tau_2 = 'n(\tau)$ ; or
4.  $\tau_1 = \langle \overline{v_k \rightarrow \tau'_k} \rangle$  and  $\tau_2 = \overline{n_{k+m}(\tau_{k+m}) \rightarrow \tau'_{k+m}}$  where  $\overline{v_k} \succeq \overline{n_k(\tau_k)}$  and  $\tau'_k \succeq \tau'_k$ ;  
or
5.  $\tau_1 = \overline{n_k(\tau_k) \rightarrow \tau'_k} ([< 'n_i(\tau_i) \mid \dots])$  and  $\tau_2 = \tau'_i$  where  $i \leq k$ ;  
or
6.  $\tau_1 = [< 'n(\tau) \mid \dots]$  and  $\tau_2 = 'n(\tau)$ .

## 3.1 Type Rules

Figure 2 gives the type rules for *DV*. We have three different types rules for application expressions  $e e'$ ; when both  $e$  and  $e'$  are known (`app`)<sub>1</sub> is applied, when  $e$  is known but  $e'$  is unknown *i.e.*  $e'$  is an abstracted argument, (`app`)<sub>2</sub> is applied, and when neither  $e$  nor  $e'$  is known *i.e.*  $e$  is an abstracted match-function and  $e'$  is an abstracted argument, (`app`)<sub>3</sub> is applied.

Type environment  $\Gamma$  is a mapping from variables to types. Given a type environment  $\Gamma$ , the type system derives a direct-type  $\tau$  for an expression  $e$ . This is written as a *type judgement*  $\Gamma \vdash e : \tau$ .  $\Gamma \parallel [x_i \mapsto \alpha_i]$  is the extension of  $\Gamma$  with  $x_i \mapsto \alpha_i$ .

## 3.2 Soundness of Type System

We prove soundness of the type system by demonstrating a subject reduction property.

LEMMA 3.1 (SUBJECT REDUCTION). *If*  $\emptyset \vdash e : \tau$  *and*  $e \longrightarrow_1 e'$  *then*  $\emptyset \vdash e' : \tau'$  *such that*  $\tau \succeq \tau'$ .

The full proof appears in Appendix A.1.

LEMMA 3.2 (SOUNDNESS OF TYPE SYSTEM). *If*  $\Gamma \vdash e : \tau$  *then*  $e$  *either computes forever or computes to a value.*

PROOF. By induction on the length of computation, using Lemma 3.1.  $\square$

## 4 Type Inference

Figure 3 gives the type inference rules. Our inference type rules follow a constrained type presentation [AW93, EST95], even though our type theory does not include subtyping. We found this formu-

(variant)	$\frac{e \rightarrow_1 e'}{n(e) \rightarrow_1 n(e')}$
(app)	$\frac{e_1 \rightarrow_1 e'_1 \quad e_2 \rightarrow_1 e'_2}{e_1 e_2 \rightarrow_1 e'_1 e'_2} \quad \frac{(\lambda_f n_k(x_k) \rightarrow e_k) n_d(v) \rightarrow_1 e_d [v/x_d] [\lambda_f n_k(x_k) \rightarrow e_k/f]}{\text{where } d \leq k}$
(let)	$\frac{e_1 \rightarrow_1 e'_1}{\text{let } x = e_1 \text{ in } e_2 \rightarrow_1 \text{let } x = e'_1 \text{ in } e_2} \quad \frac{}{\text{let } x = v_1 \text{ in } e_2 \rightarrow_1 e_2 [v_1/x]}$

Figure 1. Operational Semantic Rules

(num)	$\frac{}{\emptyset \vdash i : \text{Int}} \quad i \in \text{Num}$	(sub)	$\frac{\Gamma \vdash e : \tau \quad \tau \succeq \tau'}{\Gamma \vdash e : \tau'}$
(variant)	$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash n(e) : n(\tau)}$	(var)	$\frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}$
(abs)	$\frac{\forall i \leq k. \Gamma \parallel [f \mapsto \mu \alpha_f. \overline{n_k(\tau_k) \rightarrow \tau'_k}; x_i \mapsto \tau_i] \vdash e_i : \tau'_i}{\Gamma \vdash \lambda_f n_k(x_k) \rightarrow e_k : \mu \alpha_f. \overline{n_k(\tau_k) \rightarrow \tau'_k}}$	(let)	$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' [e/x] : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'}$
(app) <sub>1</sub>	$\frac{\Gamma \vdash e : \overline{n_k(\tau_k) \rightarrow \tau'_k} \quad \Gamma \vdash e' : n_d(\tau_d)}{\Gamma \vdash e e' : \tau'_d} \quad \text{where } d \leq k$	(app) <sub>3</sub>	$\frac{\Gamma \vdash e : \langle v_k \rightarrow \tau_k \rangle \quad \Gamma \vdash e' : v_d}{\Gamma \vdash e e' : \tau_d} \quad \text{where } d \leq k$
(app) <sub>2</sub>	$\frac{\Gamma \vdash e : \overline{n_k(\tau_k) \rightarrow \tau'_k} \quad \Gamma \vdash e' : [ > \alpha ]}{\Gamma \vdash e e' : \overline{n_k(\tau_k) \rightarrow \tau'_k} ([ > \alpha ])}$ replace $[ > \alpha ]$ above with $[ < n_i(\tau_i) \mid \dots ]$ , where $i \leq k$		

Figure 2. Type Rules

lation useful for an elegant specification of the type inference algorithm.  $\tau \setminus E$  is a constrained type, where “ $\setminus$ ” reads “where” and  $E$  is a set of equational constraints of the form  $\tau_1 = \tau_2$ .

Type environment  $\Gamma$  is a mapping from variables to types. Given a type environment  $\Gamma$ , the type inference system derives a constrained type  $\tau \setminus E$  for an expression  $e$ . This is written as a *type judgement*  $\Gamma \vdash_{\text{inf}} e : \tau \setminus E$  under the condition that  $E$  is consistent.

The following definition defines consistent and inconsistent sets.

**DEFINITION 4.1 (CONSTRAINT CONSISTENCY).** A constraint  $\tau_1 = \tau_2$  is consistent if either:

1.  $\tau_1 \in \text{TyVar}$  or  $\tau_2 \in \text{TyVar}$ ;
2.  $\tau_1 = \tau$  and  $\tau_2 = \tau$ ;
3.  $\tau_1 = n(\tau)$  and  $\tau_2 = n(\tau')$ ;
4.  $\tau_1 = n(\tau)$  and  $\tau_2 = [ > \alpha ]$  or its symmetry;
5.  $\tau_1 = [ > \alpha ]$  and  $\tau_2 = [ > \alpha' ]$ ;
6.  $\tau_1 = \overline{n_k(\tau_k) \rightarrow \tau'_k}$  and  $\tau_2 = \langle v \rightarrow \tau' \rangle$  or its symmetry;
7.  $\tau_1 = \langle v_1 \rightarrow \alpha_1 \rangle$  and  $\tau_2 = \langle v_2 \rightarrow \alpha_2 \rangle$ ;
8.  $\tau_1 = [ > \alpha ]$  and  $\tau_2 = [ < \overline{n_k(\tau_k)} ]$  or its symmetry;
9.  $\tau_1 = \mu \alpha. \overline{n_k(\tau_k) \rightarrow \tau'_k}$  and  $\tau_2 = \langle v \rightarrow \tau' \rangle$  or its symmetry;

Otherwise it is inconsistent.

A constraint set  $E$  is consistent if all the constraints in the set are

consistent.

The type inference system assigns all *DV* expressions constrained types of the form  $\tau \setminus E$  to indicate an expression of type  $\tau$  which is constrained by the constraints in  $E$ .

Following defines a closed constraint set  $E$ . A closed constraint set  $E$  will have any type errors immediately apparent in it. In the definition  $E_1 \uplus E_2$  denotes the closed union of sets of constraints: union followed by closure.  $\uplus_1$  denotes closure with respect to *Phase 1* only.

**DEFINITION 4.2 (CLOSED CONSTRAINT SET).** A set of type constraints  $E$  is closed iff

*Phase 1*

1. (Match) If  $\{\overline{n_k(\tau_k) \rightarrow \tau'_k} = \langle [ > \alpha ] \rightarrow \alpha' \rangle, [ > \alpha ] = n_d(\tau_o)\} \subseteq E$  then  $\{\alpha' = \tau'_d, \tau_o = \tau_d\} \subseteq E$  if  $d \leq k$  else fail.
2. (Variant) If  $n(\tau) = n(\tau') \in E$  then  $\tau = \tau' \in E$ .
3. (Same-Arg) If  $\langle v \rightarrow \alpha \rangle = \langle v \rightarrow \alpha' \rangle \in E$  then  $\alpha = \alpha' \in E$ .
4. (Transitivity) If  $\{\tau = \tau', \tau' = \tau''\} \subseteq E$  then  $\tau = \tau'' \in E$ .
5. (Symmetry) If  $\tau = \tau' \in E$  then  $\tau' = \tau \in E$ .

*Phase 2*

- (Simulate) If  $\overline{n_k(\tau_k) \rightarrow \tau'_k} = \langle [ > \alpha ] \rightarrow \alpha' \rangle \in E$  and  $\neg \exists n, \tau. [ > \alpha ] = n(\tau) \in E$  then  $[ > \alpha ] = [ < n_i(\tau_i) \mid \dots ] \in E$ , such that  $i \leq k$  and  $E \uplus_1 \{ [ > \alpha ] = n_i(\tau_i) \}$  is consistent.

The closure is divided in two sequential phases. Phase 2 is computed only after Phase 1 completes.

<b>(num)</b>	$\frac{}{\emptyset \vdash_{\text{inf}} i : \text{Int} \setminus \emptyset} \quad i \in \text{Num}$	
<b>(variant)</b>	$\frac{\Gamma \vdash_{\text{inf}} e : \tau \setminus E}{\Gamma \vdash_{\text{inf}} 'n(e) : 'n(\tau) \setminus E}$	<b>(var)</b> $\frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = \tau \setminus E}{\Gamma \vdash_{\text{inf}} x : \tau \setminus E}$
<b>(abs)</b>	$\frac{\forall i \leq k. \Gamma \parallel [f \mapsto \alpha_f; x_i \mapsto \alpha_i] \vdash_{\text{inf}} e_i : \tau_i \setminus E}{\Gamma \vdash_{\text{inf}} \lambda_f \overline{'n_k(x_k)} \rightarrow e_k : \overline{'n_k(\alpha_k)} \rightarrow \tau_k \setminus E} \quad \alpha_f = \overline{'n_k(\alpha_k)} \rightarrow \tau_k \setminus E$ where $\alpha_f$ and $\overline{\alpha_k}$ are fresh type variables	
<b>(app)</b>	$\frac{\Gamma \vdash_{\text{inf}} e : \tau \setminus E \quad \Gamma \vdash_{\text{inf}} e' : \tau' \setminus E}{\Gamma \vdash_{\text{inf}} e e' : \alpha \setminus E} \quad \{\tau = \langle [ > \alpha'] \rightarrow \alpha \rangle, [ > \alpha'] = \tau' \} \subseteq E$ where $\alpha$ and $\alpha'$ are fresh type variables.	<b>(let)</b> $\frac{\Gamma \vdash_{\text{inf}} e : \tau \setminus E \quad \Gamma \vdash_{\text{inf}} e' [e/x] : \tau' \setminus E}{\Gamma \vdash_{\text{inf}} \text{let } x = e \text{ in } e' : \tau' \setminus E}$

Figure 3. Type Inference Rules

The *(Match)* rule is the crux of our type inference system. It enables match-functions to choose the return type corresponding to the argument type. The closure rule for normal functions is “if  $\tau_1 \rightarrow \tau'_1 = \tau \rightarrow \tau' \in E$  then  $\{\tau = \tau_1, \tau'_1 = \tau'\} \subseteq E$ ”. *(Match)* is the generalization of this rule to match-functions. When the argument type to the match-function is known, *i.e.* it is  $'n_d(\tau_o)$ , then *(Match)* simply selects the matching sub-function and applies the above regular function closure rule. Unknown arguments introduce no immediate type errors and so are not analyzed. If the variant is not in the match-type, there is no closure of  $E$ : closure fails.

*(Variant)* ensures that if two variants are equal they have the same argument type.

*(Same-Arg)* ensures that identical variants applied to the same match-function have identical component types.

*(Simulate)* adds precision to the type of an unknown argument applied to a known match-function. So if  $'n_k(\tau_k) \rightarrow \tau'_k = \langle [ > \alpha] \rightarrow \alpha' \rangle \in E$  but  $\neg \exists n, \tau. [ > \alpha] = 'n(\tau) \in E$  after Phase 1, then  $[ > \alpha]$  doesn't have a known concrete type. However, the above constraint does imply that  $[ > \alpha]$  could have been  $'n_1(\tau_1)$  or  $'n_2(\tau_2)$  or ... or  $'n_k(\tau_k)$ , and it would still have been consistent; anything else would have made it inconsistent. So to find all the valid  $'n_i(\tau_i)$ 's we add  $[ > \alpha] = 'n_i(\tau_i)$  to  $E$  for all  $i \leq k$  separately and compute the closure with respect to Phase 1. If the resulting closed set is consistent we know that  $'n_i(\tau_i)$  is a valid type for  $[ > \alpha]$ . At the end of all the *simulations* we add  $[ > \alpha] = \langle [ < 'n_i(\tau_i) \mid \dots] \rangle$  to  $E$  where  $'n_i(\tau_i)$  is a valid type for  $[ > \alpha]$ .

The closure is trivially seen to be computable in  $O(n^3)$  time, assuming  $K \ll n$ , where  $n = |E|$  and  $K = \max(k) \forall k. \overline{'n_k(\tau_k)} \rightarrow \tau'_k = \langle \tau \rightarrow \alpha' \rangle \in E$ . In the rare case where  $K \approx n$ , the time complexity would be  $O(n^3 K)$ . The factor  $K$  is introduced due to *(Match)* having to search through at most each of the  $K$  sub-functions to find a match (or an absence thereof).

We now define the type inference algorithm.

ALGORITHM 4.1 (TYPE INFERENCE). *Given an expression  $e$  its type  $\tau \setminus E$  (or type-error) can be inferred as follows:*

1. Produce the unique proof tree  $\emptyset \vdash e : \tau \setminus E$  via the type inference rules in Figure 3.
2. Compute  $E^+ = \text{closure}(E)$ .
3. If  $E^+$  is consistent then  $\tau \setminus E^+$  is the inferred type for  $e$ , else there is a type-error in  $e$ .

By inspection of the type inference rules in Figure 3, it is easy to see this process is deterministic, based on the structure of  $e$ , modulo the choice of fresh type variables.

We don't prove the soundness of the type inference algorithm. Rather we give a Simplification Algorithm 4.2, which simplifies the inferred constrained types to direct-types as per the type rules in Figure 2 which we have already proven sound, and prove the soundness of this simplification algorithm. However, it would not be very difficult to verify that the soundness of the type inference algorithm as well.

## 4.1 Equational Simplification

Now we present an algorithm for reducing a constrained type  $\tau \setminus E$  to an unconstrained type  $\tau_s$  which contains all the type information of  $\tau \setminus E$ , and prove its soundness. This means direct types containing the complete type information, without hard-to-digest type equations, can be presented to programmers. This improves on [Gar98] which is a lossy method.

We now give the algorithm for simplification. In the following algorithm we handle symmetry implicitly.

ALGORITHM 4.2 (SIMPLIFICATION). *A constrained type  $\tau \setminus E$  can be reduced to  $\tau_s$  by the following steps:*

1. For all constraints of the form  $\alpha = \langle \nu \rightarrow \tau' \rangle \in E$

$$E_1 = E - \{ \alpha = \langle \nu_1 \rightarrow \tau'_1 \rangle, \dots, \alpha = \langle \nu_k \rightarrow \tau'_k \rangle \} \cup \{ \alpha = \langle \nu_k \rightarrow \tau'_k \rangle \}$$

2. For all constraint sets of the form  $\{ \alpha_f = \overline{'n_k(\tau_k)} \rightarrow \tau'_k, [ > \alpha] = \tau, \alpha_f = \langle [ > \alpha] \rightarrow \alpha' \mid \dots \rangle \} \subseteq E_1$ ,

$$E_{2a} = E_1 - \{ \alpha_f = \overline{'n_k(\tau_k)} \rightarrow \tau'_k \} \\ \tau_2 \setminus E_2 = [\mu \alpha_f. \overline{'n_k(\tau_k)} \rightarrow \tau'_k / \overline{'n_k(\tau_k)} \rightarrow \tau'_k] (\tau \setminus E_{2a})$$

if  $\tau$  occurs in  $\overline{'n_k(\tau_k)} \rightarrow \tau'_k$  then:

$$\tau_3 \setminus E_3 = [\alpha_f ([ > \alpha]) / \alpha'] (\tau_2 \setminus E_2)$$

else

$$\tau_3 \setminus E_3 = \tau_2 \setminus E_2$$

$$E_4 = E_3 - \{ \alpha_f = \langle [ > \alpha] \rightarrow \alpha' \mid \dots \rangle \}$$

3. For all constraint sets of the form  $\{ [ > \alpha] = \tau, \alpha_f = \langle [ > \alpha] \rightarrow \alpha' \mid \dots \rangle \} \subseteq E_4$ ,

if  $\alpha_f$  occurs in  $\tau$  then:

$$E_5 = E_4 - \{ \alpha_f = \langle [ > \alpha] \rightarrow \alpha' \mid \dots \rangle \}$$

$$\cup \{\alpha_f = \mu \alpha_f. \langle \tau \rightarrow \alpha' \mid \dots \rangle\}$$

else  
 $E_5 = E_4$

$$4. \tau_s = \text{Unify}(E_5) (\tau_3)$$

$$\begin{aligned} & \text{Unify}(\emptyset) = \emptyset \\ & \text{Unify}(E \cup \{\alpha = \tau\}) = \text{if } \tau \equiv \alpha' \text{ then} \\ & \quad \text{if } \alpha \leq_{\text{lexicographic}} \alpha' \text{ then} \\ & \quad \quad \text{Unify}([\alpha'/\alpha]E) \circ [\alpha'/\alpha] \\ & \quad \text{else} \\ & \quad \quad \text{Unify}(E) \\ & \quad \text{else if } \alpha \text{ occurs in } \tau \text{ then} \\ & \quad \quad \text{Unify}([\mu \alpha. \tau/\alpha]E) \circ [\mu \alpha. \tau/\alpha] \\ & \quad \text{else} \\ & \quad \quad \text{Unify}([\tau/\alpha]E) \circ [\tau/\alpha] \\ & \text{Unify}(E \cup \{[> \alpha] = [> \alpha']\}) = \text{Unify}(E \cup \{\alpha = \alpha'\}) \\ & \text{Unify}(E \cup \{[> \alpha] = 'n(\tau)\}) = \\ & \quad \text{Unify}([\tau/\alpha]E) \circ [\tau/\alpha] \\ & \text{Unify}(E \cup \{[> \alpha] = 'n(\tau)/[> \alpha]\}) = \\ & \quad \text{Unify}([\tau/\alpha]E) \circ [\tau/\alpha] \\ & \text{Unify}(E \cup \{[> \alpha] = \langle v \rightarrow \tau' \rangle\}) = \\ & \quad \text{if } \tau' \equiv \alpha \text{ and } \neg \exists \tau. \alpha = \tau \in E \text{ then} \\ & \quad \quad \text{Unify}([\tau'/\alpha]E) \circ [\tau'/\alpha] \\ & \quad \quad \text{Unify}([\mu \alpha. \langle v \rightarrow \tau' \rangle/\alpha]E) \circ \\ & \quad \quad \quad [\mu \alpha. \langle v \rightarrow \tau' \rangle/\alpha] \\ & \quad \text{else} \\ & \quad \quad \text{Unify}(E) \\ & \text{Unify}(E \cup \{\tau = \tau'\}) = \text{Unify}(E) \\ & \text{Unify}(E \cup \{n(\tau) = n(\tau')\}) = \text{Unify}(E) \\ & \text{Unify}(E \cup \{v_1 \rightarrow \alpha_1 = v_2 \rightarrow \alpha_2\}) = \text{Unify}(E) \\ & \text{Unify}(E \cup \{[> \alpha] = [\langle 'n_k(\tau_k) \rangle]\}) = \\ & \quad \text{Unify}([\langle 'n_k(\tau_k) \rangle]/[> \alpha]E) \circ \\ & \quad \quad [\langle 'n_k(\tau_k) \rangle]/[> \alpha] \\ & \text{Unify}(E \cup \{[\mu \alpha. \langle v \rightarrow \tau' \rangle] = \langle v \rightarrow \tau' \rangle\}) = \\ & \quad \text{if } \tau' \equiv \alpha \text{ and } \neg \exists \tau. \alpha = \tau \in E \text{ then} \\ & \quad \quad \text{Unify}([\mu \alpha. \langle v \rightarrow \tau' \rangle/\alpha]E) \circ \\ & \quad \quad \quad [\mu \alpha. \langle v \rightarrow \tau' \rangle/\alpha] \\ & \quad \text{else} \\ & \quad \quad \text{Unify}(E) \end{aligned}$$

Figure 4.  $\text{Unify}(E)$

$\text{Unify}(E)$  defined in Figure 4 is a recursive function, which takes in a set of constraints and returns a composition of substitutions. When this composition is applied to  $\tau$  in Step 4 above, the constraint-free type results. Steps 2 and 3 generate recursive types.

The above algorithm can be thought of as 3 phases, revolving around  $\text{Unify}$ :

1. *Pre-Unify*: which adds and removes constraints from  $E$  to generate  $E'$ . This includes Steps 1, 2 and 3.
2. *Unify*: which computes  $\text{Unify}(E')$  which generates a composition of substitutions  $s$ .
3. *Post-Unify*: which applies  $s$  to  $\tau$  to generate unconstrained direct-type  $\tau_s$  i.e. Step 4.

Note that  $\text{Unify}()$  simply throws away some constraints like  $n(\tau) = n(\tau')$  and  $\langle v_1 \rightarrow \alpha_1 \rangle = \langle v_2 \rightarrow \alpha_2 \rangle$ ; this is sound as closure would have extracted all relevant information from these constraints, hence they can be discarded safely.

The examples in Section 5 illustrate the significance of each of the steps as well  $\text{Unify}()$ .

## 4.2 Soundness of Simplification

We prove the following lemmas,

LEMMA 4.1 (TERMINATION OF SIMPLIFICATION).  $\text{simplify}(\tau \setminus E) = \tau_s$  iff  $E$  is closed and consistent.

The full proof appears in Appendix A.2. The simplification algorithm is sound in the sense that a derivation in the (non-inference) type system may be constructed using the type resulting from simplification. Thus, the type inference algorithm plus simplification amounts to a type inference algorithm for the non-inference type system.

LEMMA 4.2 (SOUNDNESS OF SIMPLIFICATION). If  $\Gamma \vdash_{\text{inf}} e : \tau \setminus E$  and  $\text{simplify}(\tau \setminus E^+) = \tau_s$ , where  $\Gamma = \overline{x_j \mapsto \alpha_j}$  and  $\text{simplify}(\alpha_j \setminus E^+) = \tau_j$ , and  $E^+$  is non-cyclic, then  $\Gamma' \vdash e : \tau_s$  where  $\Gamma' = \overline{[\tau_j/\alpha_j] \Gamma}$ .

The full proof appears in Appendix A.2.

For simplicity, we only consider non-cyclic  $E$  i.e. it doesn't contain cyclic constraints and hence no recursive types, in all the proofs.

## 5 Examples

We now illustrate our type inference system to infer constrained types and equational simplification algorithm to simplify these inferred constrained types into human-readable non-constrained types. Let us start with a basic match-function:

$$\lambda_f \text{ 'int}() \rightarrow 1 \mid \text{ 'bool}() \rightarrow \text{ 'true}() \quad (1)$$

The following would be the inferred type of (1):

$$\langle \text{ 'int}() \rightarrow \text{Int} \mid \text{ 'bool}() \rightarrow \text{ 'true}() \rangle \setminus \emptyset$$

Since the set of constraints is  $\emptyset$ , the above without  $\emptyset$  is the simplified type as well. An example of a corresponding application could be:

$$(\lambda_f \text{ 'int}() \rightarrow 1 \mid \text{ 'bool}() \rightarrow \text{ 'true}()) \text{ 'int}() \quad (2)$$

(**app**) would derive the type of (2) as:

$$'a \setminus \{ \langle \text{ 'int}() \rightarrow \text{Int} \mid \text{ 'bool}() \rightarrow \text{ 'true}() \rangle = \langle [ > 'b ] \rightarrow 'a \rangle, [ > 'b ] = \text{ 'int}() \}$$

On closure, (*Match*) would generate  $\text{Int} = 'a \in E^+$ . Had  $\text{ 'bool}()$  had been the argument, then (*Match*) would have generated  $\text{ 'true}() = 'a$ , thus achieving the desired result of tying the return type of a function to the argument type instead of the whole function. The simplified type of (2) would be  $\text{Int}$  by direct substitution of  $'a$  using the constraint  $\text{Int} = 'a$ .

Let us now look at a more interesting example, when the type of the argument is unknown:

$$\lambda_f \text{ 'redirect}(m) \rightarrow e_1 \ m \quad (3)$$

where  $e_1$  is (1) above. If  $m$  is assigned type variable  $'m$ , the inferred type of just  $e_1 \ m$  by (**app**) would be:

$$'a \setminus \{ \langle \text{ 'int}() \rightarrow \text{Int} \mid \text{ 'bool}() \rightarrow \text{ 'true}() \rangle = \langle [ > 'b ] \rightarrow 'a \rangle, [ > 'b ] = 'm \}$$

$[ > 'b ] = 'm$  constrains  $'m$  to a variant type. Hence if the above type is  $'a \setminus E'$  the type of (3) by (**abs**) would be:

$$\text{ 'redirect}('m) \rightarrow 'a \setminus E' \cup \{ f = \text{ 'redirect}('m) \rightarrow 'a \} \equiv 'a \setminus E$$

The simplified type of (3) would then be:

$\lambda_f \text{redirect}([\> 'b]) \rightarrow ('int() \rightarrow \text{Int} \mid 'bool() \rightarrow 'true()) [\> 'b]$   
or  
 $\lambda_f \text{redirect}([\< 'int() \mid 'bool()]) \rightarrow ('int() \rightarrow \text{Int} \mid 'bool() \rightarrow 'true()) [\< 'int() \mid 'bool()]$

mainly due to *Unify()* on  $'int() \rightarrow \text{Int} \mid 'bool() \rightarrow 'true() = \langle [\> 'b] \rightarrow 'a \rangle$  and Phase 2 of closure. As is clearly evident this simplified type is more precise but verbose and hence less-readable as well.

Now consider the following application with first-class message passing where  $'int()$  is the first-class message:

$$(\lambda_f \text{redirect}(m) \rightarrow e_1 m) (\text{redirect}('int())) \quad (4)$$

**(app)** would infer its type as:

$$'b \setminus E \cup \{ \text{redirect}(m) \rightarrow 'a = \langle [\> 'c] \rightarrow 'b \rangle, [\> 'c] = \text{redirect}('int()) \}$$

The simplified type of (4) would be  $\text{Int}$ . Let us go through the steps. (*Match*) closure on the above constraint generates:

$$\{ 'a = 'b, 'int() = 'm \}$$

Next,  $'int() = 'm$  with  $[\> 'b] = 'm$  on (*Transitivity*) gives  $[\> 'b] = 'm$  and  $\{ ('int() \rightarrow \text{Int} \mid 'bool() \rightarrow 'true()) = \langle [\> 'b] \rightarrow 'a \rangle, [\> 'b] = 'int() \}$  on (*Match*) generates:

$$\text{mathrmInt} = 'b$$

*Unify()* with  $\text{Int} = 'b$  generates a substitution  $[\text{Int}/'b]$  and hence the simplified type is  $\text{Int}$ .

Lets now consider the case when the match-function is unknown:

$$\lambda_f \text{dummy}(o) \rightarrow o ('zero()) \quad (5)$$

**(abs)** with **(app)** will infer the following constrained type:

$$\text{dummy}(o) \rightarrow 'b \setminus \{ 'o = \langle [\> 'c] \rightarrow 'b \rangle, [\> 'c] = 'zero() \}$$

The simplified type would be:

$$\text{dummy}(\langle 'zero() \rightarrow 'b \rangle) \rightarrow 'b$$

by Step 1 of Algorithm 4.2 and substitutions by *Unify()*.

Now consider a minor variant of (5):

$$\lambda_f \text{dummy}(o) \rightarrow o ('m()); o ('n()) \quad (6)$$

The simplified type of (6) mainly by Step 1 of Algorithm 4.2 would be:

$$\text{dummy}(\langle 'm() \rightarrow 'a \mid 'n() \rightarrow 'b \rangle) \rightarrow 'b$$

Let us look at the case when both the match-function and the argument is unknown:

$$\lambda_f \text{dummy}(o) \rightarrow \lambda_f \text{redirect}(m) \rightarrow o m \quad (7)$$

**(abs)** and **(app)** would infer the type of (7) as:

$$\text{dummy}(o) \rightarrow \text{redirect}(m) \rightarrow 'a \setminus \{ 'o = \langle [\> 'b] \rightarrow 'a \rangle, [\> 'b] = 'm \}$$

and the simplified type would be:

$$\text{dummy}(\langle [\> 'b] \rightarrow 'a \rangle) \rightarrow \text{redirect}([\> 'b]) \rightarrow 'a$$

Now lets consider the most complicated example which shows the real usefulness of (*Simulate*) closure rule:

$$\lambda_f \text{redirect}(m) \rightarrow (\lambda_f a(x) \rightarrow x > 0 \mid 'b(x) \rightarrow x + 1) m; \\ (\lambda_f 'b(x) \rightarrow x + 0 \mid 'c(x) \rightarrow x == 1) m \quad (8)$$

It can be deduced by inspection that  $m$  could only be substituted  $'b$  at run-time since it is the only variant name present in both match-functions and hence the return type of  $\lambda_f \text{redirect}$  could only be  $\text{Int}$  and never  $\text{Bool}$ .

Our system (without Closure Phase 2) will, however, generate only the following constraints even after closure:

$$\{ ('a(x) \rightarrow \text{Bool} \mid 'b(x) \rightarrow \text{Int}) = \langle [\> 'm_1] \rightarrow 'e \rangle, [\> 'm_1] = 'm \\ ('b(x) \rightarrow \text{Int} \mid 'c(x) \rightarrow \text{Bool}) = \langle [\> 'm_2] \rightarrow 'f \rangle, [\> 'm_2] = 'm \}$$

There is no constraint on return types  $'e$  and  $'f$  nor any constraint of the form  $'m = \text{Int}$ . This constraint set is still consistent since the message  $m$  in the program is not sent and hence the above code is essentially dead. The simplified type of (8), analogous to that of (3), would be:

$$\text{redirect}([\> 'm_2]) \rightarrow ('b() \rightarrow \text{Int} \mid 'c() \rightarrow \text{Bool}) [\> 'm_2]$$

If  $m$  were sent in the future as say  $'a(5)$ , then the constraints

$$\{ ('b(x) \rightarrow \text{Bool} \mid 'c(x) \rightarrow \text{Int}) = \langle [\> 'm_2] \rightarrow 'f \rangle, [\> 'm_2] = 'a(\text{Int}) \}$$

would be generated and (*Match*) would fail. Thus the program would not typecheck.

Had (*Simulate*) been computed, it would have generated

$$[\> 'm_1] = [\< 'b()], [\> 'm_2] = [\< 'b()]$$

resulting in the simplified type of (8) to be:

$$\text{redirect}([\< 'b()]) \rightarrow ('b() \rightarrow \text{Int} \mid 'c() \rightarrow \text{Bool}) [\< 'b()]$$

which clearly implies that  $\text{Int}$  is the only possible return type for  $\text{redirect}$ .

Now lets take a look at expressions which result in cyclic types i.e.  $\mu \alpha. \tau$ . Neither type derivation nor closure generate cyclic types; they are only generated during *simplify()*.

$$\lambda_f \text{this}() \rightarrow f \quad (9)$$

The simplified type would be:

$$\mu f. \text{this}() \rightarrow f$$

mainly due to Step 2 of Algorithm 4.2.

Consider a similar example:

$$\lambda_f \text{this}(x) \rightarrow f x \quad (10)$$

The simplified type would be:

$$\mu f. \text{this}([\> 'a]) \rightarrow (f) [\> 'a]$$

and with (*Simulate*)

$$\mu f. \text{this}(\mu [\> 'a]. [\< \text{this}([\> 'a])]) \rightarrow 'f (\mu [\> 'a]. [\< \text{this}([\> 'a])])$$

mainly due to Step 2 of Algorithm 4.2 and *Unify* on cyclic constraint  $'m = [\< \text{this}(m)]$  such that  $'m = [\> 'a]$  is in the generated set of constraints.

Lets consider an example of self-application:

$$\lambda \text{dummy}(x) \rightarrow x ('self(x)) \quad (11)$$

The simplified type will be:

$$\text{dummy}(\mu s. (\text{self}(s) \rightarrow 'a)) \rightarrow 'a$$

mally due to Step 3 of Algorithm 4.2.

## 6 Encoding Objects and Other Features

We now show how a collection of simple macros allow object-based programs to be faithfully encoded into *DV*.

The basic idea is to encode classes and objects as match-functions and messages as polymorphic variants, thus reducing message-sending to simple function application. There is nothing unusual in the translation itself, the main feature is the expressiveness of its typing: match types allow arbitrary objects to typecheck encoding messages as variants.

**DEFINITION 6.1.** *The object syntax is defined by the following translation.*

```
(class)   $\llbracket \text{class } (\lambda_{this} \overline{n_k(x_k)} \rightarrow e_k) \rrbracket = \lambda_{this} \_ \rightarrow \lambda_{this} \overline{n_k(x_k)} \rightarrow \llbracket e_k \rrbracket$ 
(new)     $\llbracket \text{new } e \rrbracket = \llbracket e \rrbracket ()$ 
(send)    $\llbracket e_1 \leftarrow e_2 \rrbracket = \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket$ 
(message)  $\llbracket 'n(e) \rrbracket = 'n(\llbracket e \rrbracket)$ 
```

Since match-functions are recursive, we get recursive objects with *this* for self-reference within the object. We now illustrate this encoding with *ftpProxy* example from the introduction.

```
 $\llbracket \text{let } ftp = \text{new } (\text{class } (\lambda_f 'get () \rightarrow 1)) \text{ in}$ 
 $\text{let } ftpProxy =$ 
 $\text{new } (\text{class } (\lambda_f 'send(m) \rightarrow (ftp \leftarrow m))) \text{ in}$ 
 $ftpProxy \leftarrow 'send('get()) \rrbracket$ 
 $\Downarrow (\text{translation to } DV)$ 
 $\text{let } ftp = (\lambda_f \_ \rightarrow \lambda_f 'get () \rightarrow 1) () \text{ in}$ 
 $\text{let } ftpProxy =$ 
 $(\lambda_f \_ \rightarrow \lambda_f 'send(m) \rightarrow (ftp m)) () \text{ in}$ 
 $ftpProxy 'send('get()) \rrbracket$ 
```

It creates new *ftp* and *ftpProxy* objects with *ftpProxy* delegating messages to *ftp*. For simplicity, *ftp* returns an integer in response to the *'get* request.

The simplified type of *ftpProxy* produced by our system would be:

```
'send([> 'm])  $\rightarrow$  ('get()  $\rightarrow$  Int) [> 'm]
```

and with (*Simulate*):

```
'send([< 'get()])  $\rightarrow$  ('get()  $\rightarrow$  Int) ([< 'get()])
```

which is similar to example (3) in Section 5. We do not deal with inheritance here, but the various standard approaches should apply.

We show that records and if-then-else can also be encoded with variants and match-types alone, thus defining a very expressive language.

### 6.1 Encoding Records

In Section 1.1 we discussed the duality of variants and records. We showed how the ML type system allows only records with all elements of the same type to be encoded in terms of variants. We now show how match-functions can fully and faithfully encode records. This should not be surprising given the above encoding of objects.

A match-function is essentially a sophisticated form of the ML match statement. Hence, record encoding of a match-function would be identical to that of match given in Section 1.1. The key

observation is that, since every case of the match can return a different type, it allows records with differently-typed fields to be encoded. For example, the record  $\{l_1 = 5, l_2 = 'wow(3)\}$  is encoded as  $\lambda_f 'l_1(x) \rightarrow 5 \mid 'l_2(x) \rightarrow 'wow(3)$  and has type  $('l_1('a) \rightarrow \text{Int} \mid 'l_2('b) \rightarrow 'wow(\text{Int}))$ .

### 6.2 Encoding if-then-else

**if-then-else** statements can be easily encoded via match-functions using polymorphic variants *'true()* and *'false()* to correspond to boolean values *true* and *false* respectively. This encoding has the added advantage that the two “branches” can have different return types. **if-then-else** is then encoded as

```
 $\llbracket \text{if } e \text{ then } e_1 \text{ else } e_2 \rrbracket = (\lambda_f 'true () \rightarrow e_1 \mid 'false () \rightarrow e_2) e$ 
```

## 7 Related Work

Previous papers containing static type systems for first class messages include those by Wakita [Wak93], Nishimura [Nis98], Müller & Nishimura [MN00], and Pottier [Pot00]. The main advantage of our system is it is significantly simpler. No existing programming language implementation efforts have included typed first-class messages, this is an implicit indication of a need for a more simple type system that captures their spirit.

Wakita [Wak93] presents an inter-object communication framework based on RPC-like message passing. He does not present a type system for his language so a comparison with his system is not possible.

Nishimura [Nis98] develops a second order polymorphic type system for first-class messages (referring to them as *dynamic* messages in the paper), where type information is expressed by kindings of the form  $t :: k$ , where  $t$  is a type variable indexing the type information of the object or message and  $k$  is a kind representing the type information. It has no type directly representing the type structure of objects and messages. This system is very complicated and Müller and Nishimura in [MN00] (same second author) attempt to present a simpler system.

Müller et al [MN00] present a monomorphic type inference system based on OF (*objects* and *features*) constraints. They extend traditional systems of feature constraints by a selection constraint  $x(y)z$  intended to model the behavior of a generic message-send operation. This does simplify things a little, but, is arguably still not simple enough to be implemented in a programming language.

Bugliesi and Crafa [BC99] also attempt to simplify Nishimura’s original work [Nis98]. However, they choose a higher-order type system, and abandon type inference.

Pottier [Pot00] like us does not define a type system oriented solely around first-class messages; it is a very general type system that happens to be powerful enough to also faithfully type first-class messages. His approach is in some ways similar to ours in that conditional types are used. His system is very expressive, but is also very complex and is thus more suited to program analysis than the production of human-readable types.

## 8 Implementation

We have implemented an interpreter for *DV* in OCaml. It has an OCaml-style top-loop, in which the user can feed a *DV* expression.

The interpreter will typecheck it, compute its simplified human-readable type, evaluate it to a value and then display both the value and the type.

The core of the interpreter, which includes the Type Inference (4.1) and Simplification (4.2) Algorithms, is only a few hundred lines of code. This further validates our assertion about the simplicity of the DV type system.

The source code along with documentation and examples can be found at <http://www.cs.jhu.edu/~pari/match-functions/>.

### Acknowledgements

We would like to thank the anonymous reviewers for their comments on improving the paper.

## 9 References

- [AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, 1993.
- [BC99] Michele Bugliesi and Silvia Crafa. Object calculi with dynamic messages. In *FOOL'6*, 1999.
- [EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to oop. In *Electronic Notes in Theoretical Computer Science*, 1995.
- [Gar98] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.
- [Gar00] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, 2000.
- [GR89] Adele Goldberg and David Robson. *Smalltalk-80 The Language*. Addison-Wesley, 1989.
- [LDG<sup>+</sup>02] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.07, Documentation and user's manual*. INRIA, <http://caml.inria.fr/ocaml/htmlman/>, 2002.
- [MN00] Martin Müller and Susumu Nishimura. Type inference for first-class messages with feature constraints. *International Journal of Foundations of Computer Science*, 2000.
- [Nis98] Susumu Nishimura. Static typing for dynamic messages. In *POPL'98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.
- [Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 1995.
- [Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 2000.
- [PW91] Lewis J. Pinson and Richard S. Wiener. *Objective-C: Object Oriented Programming Techniques*. Addison-Wesley, 1991.
- [R89] Didier Rémy. Type checking records and variants in a natural extension of ML. In *POPL*, 1989.
- [Wak93] Ken Wakita. First class messages as first class continuations. In *Object Technologies for Advanced Software, First JSSST International Symposium*, 1993.

## A Proofs

### A.1 Soundness of the Type System

We prove soundness of the type system by demonstrating a subject reduction property.

LEMMA A.1.1 (SUBSTITUTION). *If  $\Gamma \parallel [x \mapsto \tau_x] \vdash e : \tau$  and  $\Gamma \vdash e' : \tau'_x$  such that  $\tau_x \succeq \tau'_x$ , then  $\Gamma \vdash e[e'/x] : \tau'$  such that  $\tau \succeq \tau'$ .*

PROOF. Follows induction on the structure of  $e$ . For simplicity we ignore the (sub) case as well as recursive match-functions.

1. (num)  $e \equiv i$ . Proof is trivial.

2. (var)  $e \equiv x$  and  $x \in \text{dom}(\Gamma)$ .

By (var),  $\Gamma \vdash x : \tau$  where  $\Gamma(x) = \tau$ .

By induction hypothesis, let  $\Gamma(x[e'/x]) = \tau'$  where  $\tau \succeq \tau'$ .

Hence, by (var),  $\Gamma \vdash x[e'/x] : \tau'$ .

3. (variant)  $e \equiv 'n(e_v)$

By (variant), let  $\Gamma \vdash 'n(e_v) : 'n(\tau_v)$  where  $\Gamma \vdash e_v : \tau_v$ . Also let  $x \mapsto \tau_x \in \Gamma$ .

By induction hypothesis, let  $\Gamma' \vdash e_v[e'/x] : \tau'_v$  and  $\Gamma' \vdash e' : \tau'_x$ , where  $\Gamma' = \Gamma - \{x \mapsto \tau_x\}$  such that  $\tau_x \succeq \tau'_x$  and  $\tau_v \succeq \tau'_v$ .

Now,  $'n(e_v)[e'/x] = 'n(e_v[e'/x])$ . Hence, by (variant),  $\Gamma' \vdash 'n(e_v[e'/x]) : 'n(\tau'_v)$ . Also, by Case 6 of Definition 3.1,  $'n(\tau_v) \succeq 'n(\tau'_v)$ .

4. (abs)  $e \equiv \overline{'n_k(x_k) \rightarrow e_k}$

By (abs),  $\Gamma \vdash \overline{'n_k(x_k) \rightarrow e_k} : \overline{'n_k(\tau_k) \rightarrow \tau'_k}$  where  $\forall i \leq k. \Gamma \parallel [x_i \mapsto \tau_i] \vdash e_i : \tau'_i$ . Also let  $x \mapsto \tau_x \in \Gamma$  and, without loss of generality,  $x \notin \{\overline{x_i}\}$ .

By induction hypothesis, let  $\forall i \leq k. \Gamma' \parallel [x_i \mapsto \tau_i] \vdash e_i[e'/x] : \tau''_i$ , where  $\Gamma' \vdash e' : \tau'_x$  and  $\Gamma' = \Gamma - \{x \mapsto \tau_x\}$  such that  $\tau_x \succeq \tau'_x$  and  $\tau'_k \succeq \tau''_k$ .

Now,  $\overline{'n_k(x_k) \rightarrow e_k}[e'/x] = \overline{'n_k(x_k) \rightarrow (e_k[e'/x])}$ . Thus, by (abs),  $\Gamma' \vdash \overline{'n_k(x_k) \rightarrow (e_k[e'/x])} : \overline{'n_k(\tau_k) \rightarrow \tau''_k}$ . Also, by Case 2 of Definition 3.1,  $\overline{'n_k(\tau_k) \rightarrow \tau'_k} \succeq \overline{'n_k(\tau_k) \rightarrow \tau''_k}$ .

5. (app)  $e \equiv e_o e_v$ . There are 3 different cases:

(a) (app)<sub>1</sub>. By (app)<sub>1</sub>,  $\Gamma \vdash e_o e_v : \tau'_d$  where  $\Gamma \vdash e_o : \overline{'n_k(\tau_k) \rightarrow \tau'_k}$ ,  $\Gamma \vdash e_v : 'n_d(\tau_d)$  for  $d \leq k$ . Also, let  $x \mapsto \tau_x \in \Gamma$ .

By induction hypothesis and Case 2 of Definition 3.1, let  $\Gamma' \vdash e_o[e'/x] : \overline{'n_k(\tau''_k) \rightarrow \tau'''_k}$  and  $\Gamma' \vdash e_v[e'/x] : 'n_d(\tau''_d)$  where  $\Gamma' \vdash e' : \tau'_x$  and  $\Gamma' = \Gamma - \{x \mapsto \tau_x\}$  such that  $\tau_x \succeq \tau'_x$ ,  $\tau_k \succeq \tau''_k$  and  $\tau'_k \succeq \tau'''_k$ .

Now,  $(e_o e_v)[e'/x] = (e_o[e'/x]) (e_v[e'/x])$ . Thus, by (app)<sub>1</sub>,  $\Gamma' \vdash (e_o[e'/x]) (e_v[e'/x]) : \tau'''_d$  and by hypothesis we know  $\tau'_d \succeq \tau'''_d$ .

(b) (app)<sub>2</sub>. By (app)<sub>2</sub>,  $\Gamma \vdash e_o e_v : \overline{'n_k(\tau_k) \rightarrow \tau'_k} ([< 'n_i(\tau_i) \mid \dots])$  where  $\Gamma \vdash e_o : \overline{'n_k(\tau_k) \rightarrow \tau'_k}$ ,  $\Gamma \vdash e_v : [< 'n_i(\tau_i) \mid \dots]$  for  $i \leq k$ . Also, let  $x \mapsto \tau_x \in \Gamma$  and  $\Gamma' = \Gamma - \{x \mapsto \tau_x\}$ . Assume,  $\Gamma \vdash e' : \tau'_x$  such that  $\tau_x \succeq \tau'_x$ . Also,  $(e_o e_v)[e'/x] = (e_o[e'/x]) (e_v[e'/x])$ .

Now there two possible choices for the induction hypothesis :

i. By induction hypothesis and Case 2 of Definition 3.1, let  $\Gamma' \vdash e_o[e'/x] : \overline{'n_k(\tau''_k) \rightarrow \tau'''_k}$  and  $\Gamma' \vdash e_v[e'/x] : [< 'n_i(\tau''_i) \mid \dots]$  such that  $\tau_k \succeq \tau''_k$  and  $\tau'_k \succeq \tau'''_k$ .

Thus, by (app)<sub>2</sub>,  $\Gamma' \vdash (e_o[e'/x]) (e_v[e'/x]) : \overline{'n_k(\tau''_k) \rightarrow \tau'''_k} ([< 'n_i(\tau''_i) \mid \dots])$  and by Case 2 of Definition 3.1,  $\overline{'n_k(\tau_k) \rightarrow \tau'_k} ([< 'n_i(\tau_i) \mid \dots]) \succeq \overline{'n_k(\tau''_k) \rightarrow \tau'''_k} ([< 'n_i(\tau''_i) \mid \dots])$ .

ii. By induction hypothesis and Cases 2 and 6 of Definition 3.1, let  $\Gamma' \vdash e_o[e'/x] : \overline{'n_k(\tau''_k) \rightarrow \tau'''_k}$  and  $\Gamma' \vdash e_v[e'/x] : 'n_i(\tau''_i)$  for some  $i \leq k$  such that  $\tau_k \succeq \tau''_k$  and  $\tau'_k \succeq \tau'''_k$ .

Thus, by (app)<sub>2</sub>,  $\Gamma' \vdash (e_o [e'/x]) (e_v [e'/x]) : \tau''_i$  and by Cases 2 and 5 of Definition 3.1,  $\overline{‘n_k(\tau_k) \rightarrow \tau'_k} ([< ‘n_i(\tau_i) \mid \dots]) \succeq \overline{‘n_k(\tau''_k) \rightarrow \tau''_k} ([< ‘n_i(\tau''_i) \mid \dots]) \succeq \tau''_i$ .

- (c) (app)<sub>3</sub>. By (app)<sub>3</sub>,  $\Gamma \vdash e_o e_v : \tau_d$  where  $\Gamma \vdash e_o : \overline{‘v_k \rightarrow \tau'_k}$ ,  $\Gamma \vdash e_v : \tau'_d$  for  $d \leq k$ . Also, let  $x \mapsto \tau_x \in \Gamma$  and  $\Gamma' = \Gamma - \{x \mapsto \tau_x\}$ . Assume,  $\Gamma \vdash e' : \tau'_x$  such that  $\tau_x \succeq \tau'_x$ . Also,  $(e_o e_v)[e'/x] = (e_o [e'/x]) (e_v [e'/x])$ .

Now there two possible choices for the induction hypothesis :

- i. By induction hypothesis and Case 2 of Definition 3.1, let  $\Gamma' \vdash e_o [e'/x] : \overline{‘v'_k \rightarrow \tau'_k}$  and  $\Gamma' \vdash e_v [e'/x] : \tau'_d$  such that  $\overline{‘v_k \succeq v'_k}$  and  $\tau_k \succeq \tau'_k$ .

Now,  $(e_o e_v)[e'/x] = (e_o [e'/x]) (e_v [e'/x])$ . Thus, by (app)<sub>3</sub>,  $\Gamma' \vdash (e_o [e'/x]) (e_v [e'/x]) : \tau'_d$  and we know by hypothesis  $\tau_d \succeq \tau'_d$ .

- ii. By induction hypothesis, let  $\Gamma' \vdash e_o [e'/x] : \overline{‘n_{k+l}(\tau'_{k+l}) \rightarrow \tau''_{k+l}}$  and  $\Gamma' \vdash e_v [e'/x] : \tau'_d$  such that  $\overline{‘v_k \succeq ‘n_k(\tau'_k)}$  and  $\tau_k \succeq \tau''_k$ .

Thus, by (app)<sub>3</sub>,  $\Gamma' \vdash (e_o [e'/x]) (e_v [e'/x]) : \tau''_d$  and we know by hypothesis  $\tau_d \succeq \tau''_d$ .

6. (let)  $e \equiv \mathbf{let} y = e_1 \text{ in } e_2$ .

By (let), let  $\Gamma \vdash e : \tau_2$  where  $\Gamma \vdash e_1 : \tau_1$  and  $\Gamma \vdash e_2[e_1/y] : \tau_2$ . Also, let  $x \mapsto \tau_x \in \Gamma$ . Without loss of generality, we assume  $x \neq y$ .

By induction hypothesis, let  $\Gamma' \vdash e_1 [e'/x] : \tau'_1$ ,  $\Gamma' \vdash e_2[e_1/y][e'/x] : \tau'_2$  and  $\Gamma \vdash e' : \tau'_x$ , where  $\Gamma' = \Gamma - \{x \mapsto \tau_x\}$  such that  $\tau_x \succeq \tau'_x$ ,  $\tau_1 \succeq \tau'_1$  and  $\tau_2 \succeq \tau'_2$ .

Now,  $x \neq y$  implies  $(\mathbf{let} y = e_1 \text{ in } e_2)[e'/x] = \mathbf{let} y = e_1 [e'/x] \text{ in } e_2 [e'/x]$ . Similarly,  $e_2[e_1/y][e'/x] = e_2 [e'/x][e_1/y]$ . Hence by (let),  $\Gamma' \vdash e [e'/x] : \tau'_2$ .

□

LEMMA A.1.2 (SUBJECT REDUCTION). *If  $\emptyset \vdash e : \tau$  and  $e \longrightarrow_1 e'$  then  $\emptyset \vdash e' : \tau'$  such that  $\tau \succeq \tau'$ .*

PROOF. Follows by strong induction on the depth of the type derivation tree, *i.e.*, the induction hypothesis applies to all trees of depth  $n - 1$  or less, where  $n$  is the depth of the proof tree of  $\emptyset \vdash e : \tau$ . Hence, following are all the possible rules that can be applied as the last step of the type derivation of  $\emptyset \vdash e : \tau$ . (Note that (app)<sub>2</sub> and (app)<sub>3</sub>, will never be applied as the last step, since the argument in (app)<sub>2</sub> and both the applicand and the argument in (app)<sub>3</sub> are program variables, and hence the application expression is not closed. By the same argument (var) will also never be the last step. These cases are handled in the Substitution Lemma.)

1. (num). Proof is trivial.

2. (variant).

Hence,  $e \equiv ‘n(e_d)$  and let the last step of the type derivation be  $\emptyset \vdash ‘n(e_d) : ‘n(\tau_d)$  and  $\emptyset \vdash e_d : \tau_d$  the penultimate one.

By (variant),  $‘n(e_d) \longrightarrow_1 ‘n(e'_d)$  where  $e_d \longrightarrow_1 e'_d$ .

By induction hypothesis, let  $\emptyset \vdash e'_d : \tau'_d$  such that  $\tau_d \succeq \tau'_d$ . Hence by (variant),  $\emptyset \vdash ‘n(e'_d) : ‘n(\tau'_d)$ . We know by Case 6 of Definition 3.1,  $‘n(\tau_d) \succeq ‘n(\tau'_d)$ .

3. (abs)

Hence,  $e \equiv \overline{‘n_k(\alpha_k) \rightarrow e_k}$ . The proof in this case trivial, since a  $\overline{‘n_k(\alpha_k) \rightarrow e_k} \in Val$ , hence it evaluates to itself.

4. (app)<sub>1</sub>

Hence,  $e \equiv e_o e_v$ . The cases when  $e_o$  and  $e_v$  are not both values are analogous to Case 2. Suppose now that both  $e_o, e_v \in Val$  then  $e \equiv (\lambda_f ‘n_k(x_k) \rightarrow e'_k) ‘n(v)$ .

Hence by (app)<sub>1</sub>, let the the last step of the type derivation be  $\emptyset \vdash e : \tau'_d$  for  $d \leq k$  and,  $\emptyset \vdash \lambda_f \overline{‘n_k(x_k) \rightarrow e'_k} : \overline{‘n_k(\tau_k) \rightarrow \tau'_k}$ ,  $\forall i \leq k$ .  $\emptyset \parallel [f \mapsto \overline{‘n_k(\tau_k) \rightarrow \tau'_k}; x_i \mapsto \tau_i] \vdash e_i : \tau'_i$  and  $\emptyset \vdash ‘n(v) : ‘n_d(\tau_d)$  be the penultimate ones, where  $n \equiv n_d$ ; while be  $\emptyset \vdash v : \tau_d$  the second to last.

By (app), let  $e \rightarrow_1 e_d [v/x_d] [\lambda_f \overline{n_k(x_k)} \rightarrow e'_k / f] \equiv e'$ .

Now by Lemma A.1.1,  $\emptyset \vdash e' : \tau''_d$  such that  $\tau'_d \succeq \tau''_d$ .

5. (let)

Hence,  $e \equiv \mathbf{let} x = e_1 \text{ in } e_2$ . There are two possible cases:

(a)  $e_1 \notin \text{Val}$ .

Hence by (let), let the last step of the type derivation be  $\emptyset \vdash e : \tau$  and  $\emptyset \vdash e_1 : \tau_1$  and  $\emptyset \vdash e_2 [e_1/x] : \tau$  be the penultimate ones.

By (let), let  $e \rightarrow_1 (\mathbf{let} x = e'_1 \text{ in } e_2) \equiv e'$  where  $e_1 \rightarrow_1 e'_1$ .

By induction hypothesis, let  $\emptyset \vdash e'_1 : \tau'_1$  such that  $\tau_1 \succeq \tau'_1$  and  $\emptyset \vdash e_2 [e'_1/x] : \tau'$  such that  $\tau \succeq \tau'$ . Hence by (let),  $\emptyset \vdash e' : \tau'$  where  $\tau \succeq \tau'$ .

(b)  $e_1 \in \text{Val}$ . So let  $e \equiv \mathbf{let} x = v \text{ in } e_2$ .

Hence by (let), let the last step of the type derivation be  $\emptyset \vdash e : \tau$  and  $\emptyset \vdash v : \tau_v$  and  $\emptyset \vdash e_2 [v/x] : \tau$  be the penultimate ones.

By (let), let  $e \rightarrow_1 e_2 [v/x] \equiv e'$ .

We already know,  $\emptyset \vdash e_2 [v/x] : \tau$  and by Case 1 of Definition 3.1,  $\tau \succeq \tau$ .

6. (sub)

In a type derivation we can collapse all the successive (sub)'s into one (sub). Hence, we know that the penultimate rule will *not* be a (sub), and thus by the (strong) induction hypothesis we can assume the lemma to hold up to the second to last rule and prove it for the penultimate rule via one of the above cases. The last step then follows via (sub).

□

LEMMA A.1.3 (SOUNDNESS OF TYPE SYSTEM). *If  $\Gamma \vdash e : \tau$  then  $e$  either diverges or computes to a value.*

PROOF. By induction on the length of computation, using Lemma A.1.2. □

## A.2 Soundness of Simplification

LEMMA A.2.1 (CANONICAL CONSTRAINT FORMS). *Following are the canonical constraint forms  $\tau_1 = \tau_2$  that can occur in any consistent  $E$ :*

1.  $\alpha = \tau$
2.  $\tau = \tau$ ;
3.  $\overline{n}(\tau) = \overline{n}(\tau')$
4.  $\overline{n}(\tau) = [\> \alpha]$
5.  $[\> \alpha] = [\> \alpha']$
6.  $\overline{n_k}(\tau_k) \rightarrow \tau'_k = \langle v \rightarrow \tau' \rangle$
7.  $\langle v_1 \rightarrow \alpha_1 \rangle = \langle v_2 \rightarrow \alpha_2 \rangle$
8.  $[\> \alpha] = [\< \overline{n_k}(\tau_k)]$
9.  $\mu \alpha. \overline{n_k}(\tau_k) \rightarrow \tau'_k = \langle v \rightarrow \tau' \rangle$

and their corresponding symmetric constraints.

PROOF. Directly follows from Definition 4.1. □

LEMMA A.2.2 (TERMINATION OF UNIFY). *Unify(E) terminates for all closed and consistent E.*

PROOF. *Unify(E)* is a recursive function with  $E = \emptyset$  as its base or terminating case. At each level of recursion *Unify(E)* removes one constraint, except at *Unify(E ∪ {> α} = [> α'])* when it adds  $\alpha = \alpha'$  to *E*. But it can be easily seen that  $\alpha = \alpha'$  will be removed at the next step without adding any additional constraints. Also there is a case for each canonical constraint form in any consistent *E*. Also since *E* is closed none of the intermediate substitutions will produce an inconsistent constraint. Hence ultimately *E* will be reduced to  $\emptyset$  and *Unify(E)* will terminate, returning a composition of substitutions.  $\square$

LEMMA A.2.3 (TERMINATION OF SIMPLIFICATION). *simplify(τ \ E) = τ<sub>s</sub> iff E is closed and consistent.*

PROOF. Step 4 of Simplification Algorithm 4.2 implies *simplify(τ \ E) = τ<sub>s</sub>* iff *Unify(E<sub>5</sub>)* terminates. By Lemma A.2.2 *Unify(E<sub>5</sub>)* terminates iff *E<sub>5</sub>* is closed and consistent. It can be easily seen that the previous steps of *simplify(τ \ E)* do not introduce nor remove any inconsistent constraints in *E<sub>5</sub>*. Hence *E<sub>5</sub>* is closed and consistent iff *E* is closed and consistent.  $\square$

LEMMA A.2.4 (TYPE STRUCTURE PRESERVATION BY SIMPLIFICATION). *If simplify(τ \ E) = τ<sub>s</sub> and τ ≠ α or [> α] then τ<sub>s</sub> has the same outermost structure as τ i.e. for example, simplify('n(τ') \ E<sup>+</sup>) = 'n(τ'<sub>s</sub>) for some τ'<sub>s</sub>, simplify('n<sub>k</sub>(α<sub>k</sub>) → τ'<sub>k</sub> \ E<sup>+</sup>) = 'n<sub>k</sub>(τ<sub>ks</sub>) → τ'<sub>ks</sub> for some τ<sub>ks</sub> and τ'<sub>ks</sub>, and so on.*

PROOF. *Unify(E)* only produces substitutions of the form [τ'' / α] or [τ'' / [> α]]. Hence, at Step 4 when the composition of substitutions generated by *Unify(E<sub>5</sub><sup>+</sup>)* are applied to τ only the type variables inside τ will get substituted, never τ itself, thus at the end τ will retain its outermost structure.  $\square$

LEMMA A.2.5 (SUB-SIMPLIFICATION).

1. *If simplify('n<sub>k</sub>(α<sub>k</sub>) → τ'<sub>k</sub> \ E) = 'n<sub>k</sub>(τ'<sub>k</sub>) → τ'<sub>k</sub> then simplify(α<sub>k</sub> \ E) = τ'<sub>k</sub> and simplify(τ<sub>k</sub> \ E) = τ''<sub>k</sub>.*
2. *If simplify('n(τ) \ E) = 'n(τ') then simplify(τ \ E) = τ' and vice-versa.*
3. *If simplify([< 'n<sub>i</sub>(τ<sub>i</sub>) | ...] \ E) = [< 'n<sub>i</sub>(τ'<sub>i</sub>) | ...] then simplify(τ<sub>i</sub> \ E) = τ'<sub>i</sub> and vice-versa.*

PROOF. Directly follows from the fact that *Unify(E)* only produces substitutions of the form [τ'' / α] or [τ'' / [> α]].  $\square$

LEMMA A.2.6 (PRE-UNIFY PROPERTY).

1. *If α = τ ∈ E and τ ≠ ⟨ν → τ'⟩ for any ν, τ', then α = τ ∈ Pre-Unify(E).*
2. *If α = τ ∈ E and τ = ⟨ν → τ'⟩ for some ν, τ', then α = ⟨ν → τ' | ...⟩ ∈ Pre-Unify(E).*

PROOF. Directly follows from inspection of *Pre-Unify*.  $\square$

LEMMA A.2.7 (CONFLUENCE). *If τ = τ' ∈ E, where E is closed, consistent and non-cyclic, and τ, τ' ≠ ⟨ν → τ''⟩ for any ν or τ'', then simplify(τ \ E) = simplify(τ' \ E).*

PROOF SKETCH. We observe that *Unify(E)* only produces substitutions which substitute a type for a type variable or a variant-type variable. And the simplified type is generated by applying this composition of substitutions to τ. Hence, *simplify(τ \ E) = s τ* and *simplify(τ' \ E) = s' τ'*. Now, since *E* is same both *s* and *s'* contain the exact same substitutions, but their orders might differ.

Hence, *simplify(τ \ E) ≠ simplify(τ' \ E)* implies  $s \tau \neq s' \tau'$ , which further implies that two different substitutions [τ<sub>1</sub> / α] and [τ<sub>2</sub> / α] exist in *s* and *s'* such that τ<sub>1</sub> and τ<sub>2</sub> have a different outermost structures. In a closed and consistent this is only possible with τ<sub>1</sub> = 'n<sub>k</sub>(τ<sub>k</sub>) → τ'<sub>k</sub> and τ<sub>2</sub> = ⟨ν<sub>k</sub> → τ''<sub>k</sub>⟩. However, during the Step 2 of Algorithm 4.2 we remove α = ⟨ν<sub>k</sub> → τ''<sub>k</sub>⟩, thus leaving only α = 'n<sub>k</sub>(τ<sub>k</sub>) → τ'<sub>k</sub> in the *E* passed to *Unify()*. Thus, the above case will never arise and the lemma will always hold.  $\square$

LEMMA A.2.8 (SOUNDNESS OF SIMPLIFICATION). *If Γ ⊢<sub>inf</sub> e : τ \ E and simplify(τ \ E<sup>+</sup>) = τ<sub>s</sub>, where Γ =  $\overline{x_j \mapsto \alpha_j}$  and simplify(α<sub>j</sub> \ E<sup>+</sup>) = τ<sub>j</sub> and E<sup>+</sup> is non-cyclic, then Γ' ⊢ e : τ<sub>s</sub> where Γ' =  $\overline{[\tau_j / \alpha_j]} \Gamma$ .*

PROOF. Following induction on the structure of *e*.

1. (num)  $e \equiv i$ . Proof is trivial.
2. (variant)  $e \equiv 'n(e')$ .

By (variant),  $\Gamma \vdash_{\text{inf}} 'n(e') : 'n(\tau') \setminus E$  where  $\Gamma \vdash_{\text{inf}} e' : \tau' \setminus E$ . By assumption *simplify('n(τ') \ E<sup>+</sup>) = τ<sub>s</sub>*.

As per Lemma A.2.4, let *simplify('n(τ') \ E<sup>+</sup>) = 'n(τ'<sub>s</sub>)*. Hence by Lemma A.2.5, *simplify(τ' \ E<sup>+</sup>) = τ'<sub>s</sub>*.

By induction hypothesis, let  $\Gamma' \vdash e' : \tau'_s$ . Hence by (variant),  $\Gamma' \vdash 'n(e') : 'n(\tau'_s)$ .

3. (var)  $e \equiv x$  and  $x \in \text{dom}(\Gamma)$ . (If  $x \notin \text{dom}(\Gamma)$  inference fails).

By (var),  $\Gamma \vdash_{\text{inf}} x : \tau \setminus E$  where  $\Gamma(x) = \tau \setminus E$ .

By induction hypothesis, let  $\text{simplify}(\tau \setminus E^+) = \tau_s$  such that  $\Gamma'(x) = \tau_s$ . Hence  $\text{simplify}(\tau \setminus E^+) = \tau_s$ .

By (var),  $\Gamma' \vdash x : \tau_s$ .

4. (abs)  $e \equiv \lambda_f \overline{'n_k(x_k)} \rightarrow e_k$ .

By (abs),  $\Gamma \vdash_{\text{inf}} e : \overline{'n_k(\alpha_k) \rightarrow \tau_k} \setminus E$  where  $\forall i \leq k. \Gamma \parallel [f \mapsto \alpha_f, x_i \mapsto \alpha_i] \vdash_{\text{inf}} e_i : \tau_i \setminus E$  and  $\alpha_f = \overline{'n_k(\alpha_k) \rightarrow \tau_k} \in E$ . By assumption  $\text{simplify}(\overline{'n_k(\alpha_k) \rightarrow \tau_k} \setminus E^+) = \tau_s$ . Hence by Lemma A.2.3  $E^+$  is consistent.

By Lemma A.2.4, let  $\text{simplify}(\overline{'n_k(\alpha_k) \rightarrow \tau_k} \setminus E^+) = \tau_s = \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$  for some  $\overline{\tau'_k}$  and  $\overline{\tau''_k}$ , and then by Lemma A.2.7,  $\text{simplify}(\alpha_f \setminus E^+) = \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$ . Then by Lemma A.2.5,  $\forall i \leq k. \text{simplify}(\alpha_i \setminus E^+) = \tau'_i$  and  $\text{simplify}(\tau_i \setminus E^+) = \tau''_i$ .

Now, by induction hypothesis, let  $\forall i \leq k. \Gamma' \parallel [f \mapsto \overline{'n_k(\tau'_k) \rightarrow \tau''_k}; x_i \mapsto \tau'_i] \vdash e_i : \tau''_i$ . Hence by (abs),  $\Gamma' \vdash e : \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$ .

5. (app)  $e \equiv e_o e_v$ .

By (app),  $\Gamma \vdash_{\text{inf}} e : \alpha \setminus E$  where  $\Gamma \vdash_{\text{inf}} e_o : \tau_o \setminus E$ ,  $\Gamma \vdash_{\text{inf}} e_v : \tau_v \setminus E$  and  $\{\tau_o = \langle [ > \alpha' ] \rightarrow \alpha \rangle, [ > \alpha' ] = \tau_v\} \subseteq E$ . By assumption  $\text{simplify}(\alpha \setminus E^+) = \tau_s$ . Hence by Lemma A.2.3  $E^+$  is consistent. Now, we observe from the type inference rules in Figure 3 and Definition 4.1 that  $\tau_o \equiv \overline{'n_k(\alpha_k) \rightarrow \tau'_k}$  or  $\alpha_o$  and  $\tau_v \equiv 'n_d(\tau)$  or  $\alpha_v$ . Hence there are the following possible combinations:

(a)  $\tau_o \equiv \overline{'n_k(\alpha_k) \rightarrow \tau'_k}$  and  $\tau_v \equiv 'n_d(\tau)$ .

So  $\{\overline{'n_k(\alpha_k) \rightarrow \tau'_k} = \langle [ > \alpha' ] \rightarrow \alpha \rangle, [ > \alpha' ] = 'n_d(\tau)\} \subseteq E$ . By (Match), we know  $\{\alpha = \tau_d, \tau = \alpha_d\} \subseteq E^+$  where  $d \leq k$ .

By Lemma A.2.4, let  $\text{simplify}(\overline{'n_k(\alpha_k) \rightarrow \tau'_k} \setminus E^+) = \tau_s = \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$  for some  $\overline{\tau'_k}$  and  $\overline{\tau''_k}$ , then by Lemma A.2.5,  $\text{simplify}(\alpha_k \setminus E^+) = \tau'_k$  and  $\text{simplify}(\tau_k \setminus E^+) = \tau''_k$ , and similarly let  $\text{simplify}('n_d(\tau) \setminus E^+) = 'n_d(\tau')$ , then  $\text{simplify}(\tau \setminus E^+) = \tau'$ . By Lemma A.2.7,  $\text{simplify}(\tau \setminus E^+) = \text{simplify}(\alpha_d \setminus E^+)$  i.e.  $\tau' = \tau'_d$  and  $\text{simplify}(\alpha \setminus E^+) = \text{simplify}(\tau_d \setminus E^+)$  which implies  $\text{simplify}(\alpha \setminus E^+) = \tau''_d$ .

Now, by induction hypothesis, let  $\Gamma' \vdash e_o : \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$  and  $\Gamma' \vdash e_v : 'n_d(\tau'_d)$ . Now, by (app)<sub>1</sub>,  $\Gamma' \vdash e_o e_v : \tau''_d$ .

(b)  $\tau_o \equiv \overline{'n_k(\alpha_k) \rightarrow \tau'_k}$  and  $\tau_v \equiv \alpha_v$ .

So we know,  $\{\overline{'n_k(\alpha_k) \rightarrow \tau'_k} = \langle [ > \alpha' ] \rightarrow \alpha \rangle, [ > \alpha' ] = \alpha_v\} \subseteq E$ . Now there are 2 possible cases:

i.  $\alpha_v = 'n_d(\tau) \in E^+$ . Same as 5a.

ii.  $\alpha_v = 'n_d(\tau) \notin E^+$ .

Hence  $\neg \exists n, \tau. [ > \alpha' ] = 'n(\tau) \in E^+$ . Thus (Simulate) will ensure  $[ > \alpha' ] = [ < 'n_i(\alpha_i) \mid \dots ] \in E^+$  where  $i \leq k$ . Hence by (Transitivity),  $\alpha_v = [ < 'n_i(\alpha_i) \mid \dots ] \in E^+$ . Also, notice that since  $\alpha$  is freshly generated by (app),  $\overline{'n_k(\alpha_k) \rightarrow \tau'_k} = \langle [ > \alpha' ] \rightarrow \alpha \rangle$  is the only constraint in  $E$  that  $\alpha$  occurs; and since (Match) is the only closure rule which can generate another constraint containing  $\alpha$  in  $E^+$ , which is not applicable in this case, we can infer that  $\neg \exists \tau. \alpha = \tau \in E^+$ .

By Lemma A.2.4, let  $\text{simplify}(\overline{'n_k(\alpha_k) \rightarrow \tau'_k} \setminus E^+) = \tau_s = \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$  for some  $\overline{\tau'_k}$  and  $\overline{\tau''_k}$ , then by Lemma A.2.5,  $\text{simplify}(\alpha_k \setminus E^+) = \tau'_k$  and  $\text{simplify}(\tau_k \setminus E^+) = \tau''_k$ . And by Lemma A.2.7,  $\text{simplify}(\alpha_v \setminus E^+) = [ < 'n_i(\tau'_i) \mid \dots ]$ . Hence, without loss of generality, Unify( $\overline{'n_k(\alpha_k) \rightarrow \tau'_k} = \langle [ > \alpha' ] \rightarrow \alpha \rangle$ ) during  $\text{simplify}(\alpha \setminus E^+)$  will generate a substitution  $[\overline{'n_k(\alpha_k) \rightarrow \tau'_k} ([ > \alpha' ]) / \alpha]$ ; which would be the only substitution on  $\alpha$ . Also, Unify( $[ > \alpha' ] = [ < 'n_i(\alpha_i) \mid \dots ]$ ) will generate  $[[ < 'n_i(\alpha_i) \mid \dots ] / [ > \alpha' ]]$ . Hence,  $\text{simplify}(\alpha \setminus E^+) = \overline{'n_k(\tau'_k) \rightarrow \tau''_k} ([ < 'n_i(\tau'_i) \mid \dots ])$ .

Now, by induction hypothesis, let  $\Gamma' \vdash e_o : \overline{'n_k(\tau'_k) \rightarrow \tau''_k}$  and  $\Gamma' \vdash e_v : [ < 'n_i(\tau'_i) \mid \dots ]$ . Hence, by (app)<sub>2</sub>,  $\Gamma' \vdash e_o e_v : \overline{'n_k(\tau'_k) \rightarrow \tau''_k} ([ < 'n_i(\tau'_i) \mid \dots ])$ .

Now, suppose Phase 2 is not computed.

Hence,  $Unify()$  will generate substitution  $[[> \alpha']/\alpha_v]$  and thus  $simplify(\alpha \setminus E^+) = \overline{n(\tau'_k) \rightarrow \tau''_k}([> \alpha'])$ .

Also, by induction hypothesis, let  $\Gamma' \vdash e_o : \overline{n_k(\tau'_k) \rightarrow \tau''_k}$  and  $\Gamma' \vdash e_v : [> \alpha']$ . Hence, by (app)<sub>2</sub>,  $\Gamma' \vdash e_o e_v : \overline{n_k(\tau'_k) \rightarrow \tau''_k}([> \alpha'])$ .

$\overline{n_k(\tau'_k) \rightarrow \tau''_k}([> \alpha'])$  is not only almost as expressive as  $\overline{n_k(\tau'_k) \rightarrow \tau''_k}([\langle n_i(\tau'_i) \mid \dots \rangle])$ , but also significantly more compact and less redundant from the perspective of a human-reader. A human-reader would easily deduce that  $[> \alpha']$  could be replaced by  $\langle n_i(\tau'_i) \rangle$  for all  $i \leq k$  which is only a little less precise than  $[\langle n_i(\tau'_i) \mid \dots \rangle]$ , which can only be replaced by  $\langle n_i(\tau'_i) \rangle$ 's contained inside it where  $i \leq k$ .

(c)  $\tau_o \equiv \alpha_o$  and  $\tau_v \equiv n_d(\tau)$ .

So we know,  $\{\alpha_o = \langle [> \alpha'] \rightarrow \alpha \rangle, [> \alpha'] = n_d(\tau)\} \subseteq E$ . Again there are 2 possible cases:

- i.  $\alpha_o = \overline{n_k(\alpha_k) \rightarrow \tau_k} \in E^+$ . Same as 5a.
- ii.  $\alpha_o = \overline{n_k(\alpha_k) \rightarrow \tau_k} \notin E^+$ .

Now during  $simplify(\alpha_o \setminus E^+)$ , Step 1 of Algorithm 4.2 will add  $\alpha_o = \langle [> \alpha'] \rightarrow \alpha \mid \dots \rangle$  to  $E^+$  and remove  $\alpha_o = \langle [> \alpha'] \rightarrow \alpha \rangle$  from  $E^+$ . Hence by Lemma A.2.7, let  $simplify(\alpha_o \setminus E^+) = \langle n_d(\tau') \rightarrow \alpha \mid \dots \rangle$  and  $simplify(n_d(\tau) \setminus E^+) = n_d(\tau')$ , such that by Lemma A.2.5  $simplify(\tau \setminus E^+) = \tau'$ . However, since (Match) is not applicable in this case we can infer that  $\neg \exists \tau. \alpha = \tau \in E^+$  and thus  $simplify(\alpha \setminus E^+) = \alpha$ .

Now, by induction hypothesis, let  $\Gamma' \vdash e_o : \langle n_d(\tau') \rightarrow \alpha \mid \dots \rangle$  and  $\Gamma' \vdash e_v : n_d(\tau')$ . Hence, by (app)<sub>3</sub>,  $\Gamma' \vdash e_o e_v : \alpha$ .

(d)  $\tau_o \equiv \alpha_o$  and  $\tau_v \equiv \alpha_v$ . Same as 5c.

6. (let) **let**  $x = e_1$  in  $e_2$ .

By (let),  $\Gamma \vdash_{\text{inf}} e : \tau_2 \setminus E$  where  $\Gamma \vdash_{\text{inf}} e_1 : \tau_1 \setminus E$  and  $\Gamma \vdash_{\text{inf}} e_2[e_1/x] : \tau_2 \setminus E$ .

Now, by induction hypothesis, let  $simplify(\tau_1 \setminus E) = \tau'_1$  and  $simplify(\tau_2 \setminus E) = \tau'_2$  such that  $\Gamma' \vdash e_1 : \tau'_1$  and  $\Gamma' \vdash e_2[e_1/x] : \tau'_2$ . Hence by (let),  $\Gamma' \vdash e : \tau'_2$ .

□