Functional Logic Overloading

Matthias Neubauer Peter Thiemann
Universität Freiburg
{neubauer,thiemann}@informatik.uni-freiburg.de

Martin Gasbichler Michael Sperber
Universität Tübingen
{gasbichl,sperber}@informatik.uni-tuebingen.de

ABSTRACT

Functional logic overloading is a novel approach to user-defined overloading that extends Haskell's concept of type classes in significant ways. Whereas type classes are conceptually predicates on types in standard Haskell, they are *type functions* in our approach. Thus, we can base type inference on the evaluation of functional logic programs. Functional logic programming provides a solid theoretical foundation for type functions and, at the same time, allows for programmable overloading resolution strategies by choosing different evaluation strategies for functional logic programs. Type inference with type functions is an instance of type inference with constrained types, where the underlying constraint system is defined by a functional logic program. We have designed a variant of Haskell which supports our approach to overloading, and implemented a prototype front-end for the language.

1. INTRODUCTION

Since the invention of type classes more than a decade ago [48], every year has seen astonishing new applications and interesting extensions of the original idea. Among these extensions are *constructor classes* [26], *multi-parameter type classes* [42], *implicit parameters* [35], and *functional dependencies* [31]. A number of applications of type classes critically depend on extensions to properly resolve ambiguities [28, 47]. Moreover, the various Haskell implementations feature a number of additional ad-hoc extensions.

Among the extensions, functional dependencies effectively allow writing logic programs at the type level [16, 36] and running them at type-inference time. However, these type-level programs are often awkward to write, and the introduction of functional dependencies poses additional pragmatic problems such as the resolution of overlapping instances which interact poorly with functional dependencies [37].

The upshot is that no single approach to constrained polymorphism and overloading is sufficiently general and effective to be applicable in all situations. Therefore, we propose making the overloading machinery programmable. The main device in our proposal is a conceptual shift from type-classes-as-predicates to type-classes-as-functions. Therefore, constraints in the resulting type system are now constraints in a functional logic language [18] at the type level. Thus, the study of type-level programs can immediately benefit from the large body of published knowledge on functional logic programming. This is in sharp contrast to the implemented extensions of Haskell's type classes, which have been developed incrementally and sometimes without careful integration. For instance, some systems implement functional dependencies as well as overlapping instances even though there is no theoretical work that investigates their interaction.

Functional logic programming, just like functional programming, features a variety of evaluation strategies. There are two main approaches, *residuation* and *narrowing*, which exist in many different variants, each of which has specific tradeoffs [18]. Fortunately, Hanus's evaluation model based on *definitional trees* suits all evaluation strategies [19]. Since there are similarly diverging requirements in type-level programming for resolution of overloading, it seems natural to employ a parameterized model to specify its semantics.

Contributions. We extend Haskell's type class system with the following features:

- The resolution of overloading is specified using functions at the type level.
- Type functions are defined by a terminating (conditional) term rewriting system augmented with (member) value definitions. The responsibility of making the term rewriting system terminating rests with the programmer, as with most approaches that make type checking programmable.
- The type-level semantics is clearly specified by attaching a deterministic evaluation strategy for functional logic programs to each type function. This has two important consequences:
    1. Overloading resolution is deterministic. Thus, no coherence problems arise.
    2. Different flavors of overloading resolution are programmable by choosing an evaluation strategy.
- Our system extends the scope of Haskell-style type class systems by treating overlapping instances in combination with functions (which has a number of significant applications [28]) and by being able to resolve Ada-style overloading.

POPL '02, Jan. 16-18, 2002 Portland, OR USA

On the technical side, our system is based on HM(X) [40], a framework for constrained type inference. We instantiate HM(X) with a suitable term constraint system and define constraint simplification rules, that extend the rule systems for E-unification [11]. We show that simplification is sound and complete. The resulting type inference algorithm is inherently incomplete because not all evaluation strategies (residuation, in particular) yield complete E-unification procedures [17]. A companion technical report [12] further extends our instance of HM(X) towards overloading resolution by formalizing a dictionary translation for it.

On the practical side, we have implemented a compiler frontend for a dialect of Haskell. Using our implementation, we have validated a number of applications (see Section 2) ranging from simple dependent types through polymorphic extensible records with first-class labels to Ada-style overloading. Each application requires a particular type-level evaluation strategy (outermost residuation, residuation with most-specific matching, and outermost narrowing), which demonstrates the usefulness of a parameterized strategy.

Overview. The following section presents a number of examples for our variant of Haskell which make intrinsic use of type-level functions. Section 3 introduces some notational preliminaries. Section 4 gives an overview of the HM(X) framework. The following section describes the constraint system which forms the basis of our type system. Section 6 discusses possible evaluation strategies for type-level function applications which occur during constraint simplification. It also shows how these strategies evaluate the examples from Section 2. Section 7 contains some notes on our prototype implementation. Finally, we discuss related work in Section 8 and conclude.

2. EXAMPLES

Four simple examples demonstrate our type system and our variant of Haskell: a type-safe `sprintf` function, an implementation of records with first-class labels, a function encoding type equality, and the Ada-style overloading of the operator `+`. We assume basic familiarity with Haskell. The standard resolution strategy is based on outermost residuation (see Section 6.1).

2.1 Format

The `sprintf` function from the C standard library is an example of an unsafe function that could be made type-safe by a dependent type [4]: The first parameter of `sprintf` is a format specifier which determines the number and type of the remaining parameters. The type-level function `SPRINTF` computes the type of the remaining parameters from the format specifier [37]. In our variant of Haskell, format specifiers cannot simply be strings with control characters but rather `String` constants or values from the following datatypes:[1]

```
data I f = I f
data C f = C f
data S f = S String f
```

The format specifier (using `$`) for function application)

```
fmt :: S (I (S (C String)))
fmt =  S "Int: " $ I $ S ", Char: " $ C $ "."
```

[1]This particular application is also implementable using only ML-style polymorphism [9].

means "The literal string '`Int: `' followed by an integer followed by the literal string '`, Char: `' followed by a character and terminated by a period." The following declaration specifies the type-level function `SPRINTF` together with an associated *member value* `sprintf1`. The overloaded `sprintf1` function accepts a prefix string, a format specifier, and corresponding further arguments:

```
class SPRINTF (f :: *) :: * where
  sprintf1 :: String -> f -> SPRINTF f
instance SPRINTF String = String where
  sprintf1 prefix str = prefix ++ str
instance SPRINTF (I a) = Int -> SPRINTF a where
  sprintf1 prefix (I a) =
    \i -> sprintf1 (prefix ++ show i) a
instance SPRINTF (C a) = Char -> SPRINTF a where
  sprintf1 prefix (C a) =
    \c -> sprintf1 (prefix ++ [c]) a
instance SPRINTF (S a) = SPRINTF a where
  sprintf1 prefix (S str a) =
    sprintf1 (prefix ++ str) a
```

The first line says that `SPRINTF` maps types of format specifiers, `f`, of kind `*` to results of kind `*`. The lines following the `where` keyword are type declarations of member values associated to `SPRINTF`. There is only one member function here, `sprintf1`.

The first `instance` states that a format specifier of type `String` leads to a member value with the same result type. The value-level member declaration of `sprintf1` for this case follows. In case of a format specifier starting with the `I` type constructor, the type-level function is defined recursively: the result type of `sprintf1` is a function, the parameter of this function is of type `Int`, and the return value type arises from another recursive call to `SPRINTF`. Again, the corresponding member value follows. The two remaining cases for type expressions starting with `C` and `S` are analogous.

The main entry point, `sprintf`, supplies an empty prefix to `sprintf1`:

```
sprintf :: (SPRINTF m =:= o) => m -> o
sprintf = sprintf1 ""
```

It is only applicable to values of types `m` and `o` so that the constraint `SPRINTF m =:= o` is satisfied. The operator `=:=` specifies strict equality of its left-hand side, the application of the type-level function `SPRINTF` to `m`, and its right-hand side, `o`.

The type of `sprintf fmt` is now `Int -> Char -> String`. When applied to an integer[2] and a character, the result is

```
> sprintf fmt 42 'x'
"Int: 42, Char: x."
```

2.2 Polymorphic Extensible Records with First-Class Labels

We implement records as heterogeneous lists on the type and on the data level. (Since order is important, the programmer must normalize records upon construction.) Two type constructors `EMPTY` and `CONS` create record types. A record type is an association list at the type level: it maps a record label (also a type) to the type of the corresponding record field. The representation at the value level is analogous: two *value* constructors `EMPTY` and `CONS` create record *values*; the representation of a record is an association list

[2]Throughout, we assume that numeric literals have type `Int`.

mapping record labels to field values. Thus, the structures of record types and record values as well as of record type labels and record value labels are identical. As this kind of situation is typical in type-level programming, there is a special `lifted` declaration which creates a type constructor and a value constructor at the same time (the comments show the expansions):

```
lifted EMPTY     -- data EMPTY = EMPTY
lifted CONS x xs -- data CONS x xs = CONS x xs

lifted Lx        -- data Lx = Lx
lifted Ly        -- data Ly = Ly
```

`Lx` and `Ly` are two possible labels, modeled by singleton types.

A type class `EQUAL` models equality of record labels:

```
lifted TRUE
lifted FALSE

class EQUAL (l1 :: *) (l2 :: *) :: *
reifying equal
instance EQUAL Lx Lx = TRUE
instance EQUAL Lx Ly = FALSE
instance EQUAL Ly Lx = FALSE
instance EQUAL Ly Ly = TRUE
```

The first line specifies the kinds of the arguments to `EQUAL` as well as the return kind. The `reifying` clause says that `EQUAL` has a member `equal` which mirrors the `EQUAL` function at the value level; this is possible as all types involved have `lifted` declarations.

The explicit definition of `equal` looks like this:

```
class EQUAL (l1 :: *) (l2 :: *) :: * where
  equal :: l1 -> l2 -> EQUAL l1 l2

instance EQUAL Lx Lx = TRUE where
  equal Lx Lx = TRUE
instance EQUAL Lx Ly = FALSE where
  equal Lx Ly = FALSE
instance EQUAL Ly Lx = FALSE where
  equal Ly Lx = FALSE
instance EQUAL Ly Ly = TRUE
  equal Ly Ly = TRUE
```

The `SELECT` class performs field selection. It relies on a type-level version of the `Maybe a` type (`NOTHING` and `JUST x`) as well as on a type-level conditional (`COND`).

```
lifted NOTHING
lifted JUST x

class SELECT (f :: *) (r :: *) :: *
reifying select
instance SELECT f EMPTY = NOTHING
instance SELECT f (CONS (f', v) r) =
  COND (EQUAL f f') (JUST v) (SELECT f r)

class COND (t :: *) (c :: *) (a :: *) :: *
reifying cond
instance COND TRUE x y = x
instance COND FALSE x y = y
```

The `unwrap` function allows us to define a modified field selection function:

```
unwrap (JUST t) = t

sel :: (SELECT f r =:= JUST t) => f -> r -> t
sel f r = unwrap (select f r)
```

Again, the `sel` function carries a *constrained type*: It is only applicable for types `f`, `r`, and `t` which satisfy the constraint `SELECT f r =:= JUST t`.

The `SELECT` type-function checks whether a given field is present. The `add` operation adds a new field to a record. It requires that the field is not yet present.

```
add :: (SELECT f r =:= NOTHING)
    => r -> f -> v -> CONS (f, v) r
add record field value = CONS (field, value) record
```

The `remove` class removes a field from a record:

```
class REMOVE (f :: *) (r :: *) :: *
reifying remove
instance REMOVE f EMPTY = EMPTY
instance REMOVE f (CONS (f', v) r) =
  COND (EQUAL f f') r (CONS (f', v) (REMOVE f r))
```

With this definition, removing a field from a record always succeeds, regardless of the actual presence or absence of such a field. If the presence of the field is required to remove it, then the following typing should be used with the remaining definitions unchanged:

```
rmv :: (SELECT f r =:= JUST t) => f -> r -> REMOVE f r
rmv = remove
```

An example script demonstrates the encoding. Record fields are polymorphic.

```
> add EMPTY Lx 42       :: CONS (Lx, Int)  EMPTY
> add EMPTY Lx 'c'      :: CONS (Lx, Char) EMPTY
```

Existing fields cannot be added:

```
> add (add EMPTY Lx 42) Lx 0
  -- ***ERROR***
```

Field labels are first class. The constraint `(SELECT ...)` checks that field labels are different. Application to different labels succeeds and application to equal labels fails.

```
> \l1 l2 -> add (add EMPTY l1 42) l2 "x" ::
  (SELECT f (CONS (f1, Int) EMPTY) =:= NOTHING) =>
  f1 -> f -> CONS (f, [Char]) (CONS (f1, Int) EMPTY)
> (\l1 l2 -> add (add EMPTY l1 42) l2 "x") Lx Ly ::
  CONS (Ly, [Char]) (CONS (Lx, Int) EMPTY)
> (\l1 l2 -> add (add EMPTY l1 42) l2 "x") Lx Lx
  -- ***ERROR***
```

2.3 Overlapping Instances

The definition of the function `EQUAL` in the previous subsection is tedious and restrictive: equality is only available for a fixed, predefined set of types and the number of `instance` declarations grows quadratically with the size of this set. Here is the code in our Haskell variant:

```
class EQUAL (l1 :: *) (l2 :: *) :: * where
  equal :: l1 -> l2 -> EQUAL l1 l2
with specificity

instance EQUAL a a = TRUE where
  equal a b = TRUE
instance EQUAL a b = FALSE where
  equal a b = FALSE
```

The Haskell type system would reject an equivalent definition because the two instance declarations overlap. (`EQUAL a a` is an instance of `EQUAL a b`.) Our Haskell dialect deals with this declaration by rewriting of the type function `EQUAL` with residuation and most-specific matching. That is, a match on a instance declaration will not be reduced until it is clear that no other instance declaration can match.

In particular, `EQUAL t1 t2` suspends until either `t1` and `t2` are fully instantiated to equal types or sufficiently instantiated to determine that they cannot be equal.

2.4 Ada-style Overloading

Another typical scenario is Ada-style overloading which corresponds to type classes with fixed, finite sets of instances. Consider overloading the `+` operator with addition on `Int` and `Float` as well as with list concatenation:

```
class PLUS (a :: *) (b :: *) :: * where
  (+) :: a -> b -> PLUS a b
with closed narrowing
  instance PLUS Int   Int   = Int   where ...
  instance PLUS Float Float = Float where ...
  instance PLUS [a]   [a]   = [a]   where ...
```

The phrase `with closed narrowing` stipulates the use of a narrowing semantics for `PLUS` and indicates that the following list of instances is complete. The resulting overloading resolution works similarly to the operator overloading found in the Ada programming languages. It is even more general because it does not require that overloading is locally resolved.

As an example, consider the following code:

```
f x y = (x + y) + 42
```

The `specificity` strategy from the previous subsection simply defers the resolution of overloading to the point where the types of `x` and `y` become known. The resulting type is

```
f :: (PLUS x y =:= z & PLUS z Int =:= w) => x -> y -> w
```

Using the `narrowing` semantics, the type system can resolve the overloading locally to `f :: Int -> Int -> Int`. The narrowing semantics explores all overloaded alternatives at the same time and prunes those that do not match.

2.5 Embedding of Standard Type Classes

It is easy to encode standard type classes in our system of type functions. Here is an encoding with an excerpt of the Haskell 98 type class `Eq` that characterizes types with an equality function.

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  (==) = primEqInt
instance (Eq a) => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False
instance (Eq a, Eq b) => Eq (a,b) where
  (x1, x2) == (y1, y2) = x1 == y1 && x2 == y2
```

A translation of these declarations into type-level functions takes the same route as the embedding of predicates into functional logic languages via the special singleton kind, `Success`, with element `Success` [20].

```
class Eq :: (a :: *) :: Success where
  (==) :: a -> a -> Bool

instance Eq Int = Success where {...}
instance Eq [a] = Eq a where {...}
instance Eq (a, b) = Eq a & Eq b where {...}
```

Predicates can be combined by conjunction `&`.

In Haskell, type classes can form a hierarchy where new classes can inherit operations from existing superclasses. We do not consider superclasses here because they can be expanded to sets of classes [5, 38].

3. PRELIMINARIES

A ranked alphabet $\mathcal{A}$ is a finite set of symbols with associated arities. Let further $\mathcal{X}$ be a set of variables. The set $T_{\mathcal{A}}(\mathcal{X})$ is the set of terms over alphabet $\mathcal{A}$ and variables $\mathcal{X}$. That is, it is the smallest set with $\mathcal{X} \subseteq T_{\mathcal{A}}(\mathcal{X})$ and, whenever $f \in \mathcal{A}$ with arity $n \in \mathbf{N}$ and $t_1, \ldots, t_n \in T_{\mathcal{A}}(\mathcal{X})$, then $f\ t_1 \ldots t_n \in T_{\mathcal{A}}(\mathcal{X})$. A term that does not contain variables is a *ground term*. Let $\mathcal{C}$ be a ranked alphabet of *constructor symbols* and let $\mathcal{F}$ be a ranked alphabet of *function symbols*. A *constructor term* is a term in $T_{\mathcal{C}}(\mathcal{X})$.

A substitution σ is a mapping from variables to terms which is the identity on almost all variables. A substitution extends homomorphically to terms. A term t' is a *(substitution) instance* of t if there is a substitution σ such that $t' = \sigma(t)$. In this case, we write $\sigma : t \leq t'$ or just $t \leq t'$ if σ does not matter. Two terms t and t' are *disjoint* if neither $t \leq t'$ nor $t' \leq t$. These notions extend to tuples of terms and to substitutions in the natural way. We often write $\overline{t}$ for a tuple of terms and analogously for other entities.

4. THE HM(X) FRAMEWORK

This section gives a short account of the HM(X) framework that extends the Hindley/Milner type system with constraints [40]. In particular, HM(X) provides a generic type inference algorithm that computes principal types if the underlying constraint system has certain properties.

e	$::=$	x	data variables
	$\mid$	$\lambda x.e$	lambda abstraction
	$\mid$	$e\ e$	application
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	let expression
$\Theta \ni \tau$	$::=$	a	type variables
	$\mid$	$\tau \to \tau$	function types

Figure 1: Syntactic domains for type inference

Figure 1 defines the syntactic domains. The term language is a lambda calculus with let [8]. The core type language is also standard; an applied type language would include additional type constructors.

To express principal types, HM(X) defines a notion of *constrained type scheme* which combines universal quantification with a constraint u on the type variables in $\overline{a}$.

$$s ::= \forall \overline{a}.u \Rightarrow \tau \qquad \text{type schemes}$$

$$(e\text{-}var)\frac{\Gamma(x) = s}{u \mid \Gamma \vdash x : s}$$

$$(e\text{-}lam)\frac{u \mid \Gamma\{x \mapsto \tau\} \vdash e : \tau'}{u \mid \Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$(e\text{-}app)\frac{u \mid \Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad u \mid \Gamma \vdash e_2 : \tau_2}{u \mid \Gamma \vdash e_1\ e_2 : \tau_1}$$

$$(e\text{-}spec)\frac{u \mid \Gamma \vdash e : \forall \overline{a}.u' \Rightarrow \tau' \quad u \Vdash u'[\overline{a} \mapsto \overline{\tau}]}{u \mid \Gamma \vdash e : \tau'[\overline{a} \mapsto \overline{\tau}]}$$

$$(e\text{-}gen)\frac{u \,\&\, u' \mid \Gamma \vdash e : \tau \quad \overline{a} \notin FV(u) \cup FV(\Gamma)}{u \,\&\, \exists \overline{a}.u' \mid \Gamma \vdash e : \forall \overline{a}.u' \Rightarrow \tau}$$

$$(e\text{-}let)\frac{u \mid \Gamma \vdash e_1 : s \quad u \mid \Gamma\{x \mapsto s\} \vdash e_2 : \tau}{u \mid \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$(e\text{-}conv)\frac{u \mid \Gamma \vdash e : \tau \quad u \Vdash \tau \doteq \tau'}{u \mid \Gamma \vdash e : \tau'}$$

Figure 2: Logical type system for HM(X)

Figure 2 shows the inference rules for HM(X)'s typing judgement $u \mid \Gamma \vdash e : s$ (u is a constraint and Γ a type assumption). The rule (*u-conv*) replaces the subtyping rule of HM(X). It relies on constraint entailment $\Vdash$, which is defined in the next section.

5. CONSTRAINTS

$$
\begin{array}{rcl}
U \ni u & ::= & t \doteq t \mid u \,\&\, u \mid \exists a.u \mid u\,;u \mid \mathit{SUCCESS} \mid \mathit{FAIL} \\
T \ni t & ::= & a \mid c\,\bar{t} \mid f\,\bar{t}
\end{array}
$$

Figure 3: Grammar of constraints

This section describes a constraint system for use in the HM(X) framework. A constraint specifies a unification problem involving applications of type-level functions. The grammar in Figure 3 defines the constraint language.

The primitive constraint is *strict equality* $t \doteq t'$ between two type-level terms t and t' [13]. $t \doteq t'$ is satisfied if t and t' are reducible to the same term.[3]

Type-level terms t are taken from $T = T_{\mathcal{C}\cup\mathcal{F}}(\mathcal{X})$ where $\mathcal{X}$ is the set of type variables. The type language Θ is implicitly included in T—'$\rightarrow$' is just another binary type constructor.

For convenience, the concrete syntax allows "extended type schemes" of the form $s' ::= \forall\overline{a}.u \Rightarrow t$ where the types might contain applications of type functions. In this case, s' denotes a corresponding proper type scheme $s \in \Theta$, which is obtained from s' by replacing each function application $f\ t_1 \dots t_n$ in t by a fresh type variable a and adding an equation $a \doteq f\ t_1 \dots t_n$ to the constraint.

The & operator is for constraint conjunction: $u_1 \,\&\, u_2$ is satisfied if both u_1 and u_2 is satisfied. It is commutative and associative. Existential quantification $\exists a.u$ restricts a local variable a. Its scope extends as far to the right as possible. The constraint $u_1\,; u_2$ expresses a disjunction.

$$
\begin{array}{ll}
\sigma \models t_1 \doteq t_2 & \text{iff } \sigma t_1 \Downarrow t_0 \text{ and } \sigma t_2 \Downarrow t_0 \text{ for some } t_0 \\
\sigma \models u_1 \,\&\, u_2 & \text{iff } \sigma \models u_1 \text{ and } \sigma \models u_2 \\
\sigma \models \exists a.u & \text{iff } \sigma[a \mapsto t] \models u \text{ for some } t \\
\sigma \models u_1\,; u_2 & \text{iff } \sigma \models u_1 \text{ or } \sigma \models u_2 \\
\sigma \models \mathit{SUCCESS} &
\end{array}
$$

Figure 4: Semantics of constraints

The semantics of a constraint u is the set of ground constructor substitutions that solve the constraint. A substitution σ solves u whenever $\sigma \models u$ is derivable using the axioms and rules in Figure 4.

Since the terms in equality constraints can contain function symbols, the definition of $\models$ is parameterized with an evaluation relation $t \Downarrow t'$ that evaluates term t to term t'. We defer the definition of this relation to Section 6.

The notation $\sigma \not\models u$ means that no proof exists for $\sigma \models u$.

Simplification of constraints corresponds to standard formulations of E-unification [20] with a few extensions, notably the explicit treatment of choice and existential quantification as required by HM(X). In this section, we only consider the core constraints. Rewriting steps arising from function applications are treated in Section 6. The rules in Figure 5 use a standard definition of the *free variables* $FV(u)$ of a constraint u, which treats $\exists a$ as a binding construct and uses $FV(t)$ to yield the set of variables occurring in a type-level term.

[3] "Strict equality" is a standard term in functional logic programming. It corresponds to the standard notion of equality in functional programming.

$$
\begin{array}{ll}
(\textit{u-constr-dec}) & c\ t_1 \dots t_n \doteq c\ t'_1 \dots t'_n \leadsto \\
 & \quad t_1 \doteq t'_1 \,\&\, \dots \,\&\, t_n \doteq t'_n \\
(\textit{u-constr-fail}) & c\ t_1 \dots t_n \doteq d\ t'_1 \dots t'_m \leadsto \mathit{FAIL} \\
 & \text{if } c \neq d \text{ or } n \neq m \\
(\textit{u-exch}) & t \doteq a \leadsto a \doteq t \\
 & \text{if } t \text{ not a variable} \\
(\textit{u-taut}) & a \doteq a \leadsto \mathit{SUCCESS} \\
(\textit{u-occur}) & a \doteq t \leadsto \mathit{FAIL} \\
 & \text{if } a \in CV(t) \text{ and } t \neq a \\
(\textit{u-subst}) & a \doteq t \,\&\, u \leadsto a \doteq t \,\&\, u[a \mapsto t] \\
 & \text{if } t = c\ a_1 \dots a_n \text{ and } a \notin \{a_1, \dots, a_n\} \\
(\textit{u-subst-var}) & a \doteq b \,\&\, u \leadsto a \doteq b \,\&\, u[a \mapsto b] \\
 & \text{if } a \in FV(u) \\
(\textit{u-peel}) & a \doteq c\ t_1 \dots t_n \leadsto \\
 & \quad \exists a_1 \dots a_n. \\
 & \qquad a \doteq c\ a_1 \dots a_n \,\& \\
 & \qquad\quad a_1 \doteq t_1 \,\&\, \dots \,\&\, a_n \doteq t_n \\
 & \text{if } a \notin CV(t_i) \text{ and } \exists i.t_i \text{ not a variable} \\
(\textit{u-and-success}) & \mathit{SUCCESS} \,\&\, u \leadsto u \\
(\textit{u-and-fail}) & \mathit{FAIL} \,\&\, u \leadsto \mathit{FAIL} \\
(\textit{u-choice-and}) & u \,\&(u_1\,; u_2) \leadsto (u \,\&\, u_1)\,;(u \,\&\, u_2) \\
(\textit{u-choice-fail-1}) & \mathit{FAIL}\,; u \leadsto u \\
(\textit{u-choice-fail-2}) & u\,; \mathit{FAIL} \leadsto u \\
(\textit{u-choice-success-1}) & \mathit{SUCCESS}\,; u \leadsto \mathit{SUCCESS} \\
(\textit{u-choice-success-2}) & u\,; \mathit{SUCCESS} \leadsto \mathit{SUCCESS} \\
(\textit{u-exists-drop}) & \exists a.u \leadsto u \\
 & \text{if } a \notin FV(u) \\
(\textit{u-and-exists}) & u_1 \,\&\, \exists a.u_2 \leadsto \exists a.u_1 \,\&\, u_2 \\
 & \text{if } a \notin FV(u_1) \\
(\textit{u-and-context-1}) & \dfrac{u_1 \leadsto u'_1}{u_1 \,\&\, u_2 \leadsto u'_1 \,\&\, u_2} \\
(\textit{u-exists-context}) & \dfrac{u \leadsto u'}{\exists a.u \leadsto \exists a.u'} \\
(\textit{u-choice-context-1}) & \dfrac{u_1 \leadsto u'_1}{u_1\,; u_2 \leadsto u'_1\,; u_2}
\end{array}
$$

Figure 5: Simplification of constraints

For a sound occur check in E-unification, it is necessary to consider the set $CV(t)$ of the *critical variables* of t. A variable is critical unless it is protected by a function symbol [20]:

$$
\begin{array}{lcl}
CV(a) & = & \{a\} \\
CV(c\ t_1 \dots t_n) & = & CV(t_1) \cup \dots \cup CV(t_n) \\
CV(f\ t_1 \dots t_n) & = & \emptyset
\end{array}
$$

The first four rules of Figure 5 are standard: The (*u-constr-dec*) rule performs term decomposition and pushes the equality of two terms with matching top-level constructors to the immediate subterms. The (*u-constr-fail*) rule signals failure if either the top-level constructors do not match or if the number of subtrees does not agree. The (*u-exch*) rule orients equations so that variables appear on the left side. The rule (*u-taut*) removes trivial equations.

The (*u-occur*) rule performs the occurs-check restricted to critical variables as explained above.

The rule (*u-subst*) applies a constructor substitution and (*u-subst-var*) substitutes a variable by an other variable. The rule (*u-peel*) peels a constructor substitution from the top of a term, potentially exposing a function call or making it possible to apply (*u-occur*) on a subterm.

The (*u-and-success*) and (*u-and-fail*) rules specify how conjunction interacts with success and failure, respectively.

The (*u-and-exists*) rule allows an existential quantifier to float out of a conjunction unless variable capture forbids it. The (*u-exists-drop*) rule drops an existential quantification if the bound variable does not appear in the constraint.

The (*u-choice-and*) rule is a distributive law of conjunction over choice. The rules (*u-choice-fail-**) remove a failing constraint from a choice operator. The (*u-choice-success-**) rules select the first succeeding alternative of a choice.

The last group of rules determines the context in which transformations may occur: The rules (*u-and-context-**) allow transformation in both arguments of a conjunction (it is commutative). Rule (*u-exists-context*) performs transformation under existential quantification and rules (*u-choice-context-**) enables transformation of each individual choice.

Here are some properties of constraint simplification:

PROPOSITION 1 (EQUIVALENCE). *Suppose that* $u \rightsquigarrow u'$. *Then* $\sigma \models u$ *if and only if* $\sigma \models u'$.

PROPOSITION 2. *Constraint simplification is confluent.*

DEFINITION 1. *A constraint* u *is* normalized *iff* $u = FAIL$ *or* $u = SUCCESS$ *or* $u = \exists\overline{a}.u_1 \,\&\ldots\&\, u_n$ *where* $\{\overline{a}\} = FV(u_1 \,\&\ldots\&\, u_n)$ *and each* u_i *is in one of the following forms:*

- $a \doteq b$ *where* a *and* b *are different variables;*
- $a \doteq C\ a_1 \ldots a_k$ *a constructor substitution where* $a \notin \{a_1 \ldots a_k\}$*;*
- $a \doteq F\ t_1 \ldots t_k$ *where* $t_1, \ldots, t_k \in T_C(\mathcal{X})$*;*
- $u'_1; \ldots; u'_m$ *where* $u'_1, \ldots, u'_m$ *are normalized, but neither FAIL nor SUCCESS.*

Moreover, u *is in* solved form *iff* u *is normalized and each variable* a *occurs at most once in a constraint* $a \doteq \ldots$*;*

PROPOSITION 3. *If the rule (u-peel) is restricted to apply at most once to any particular constraint, then constraint simplification terminates with a normalized constraint.*

Finally, we define the entailment relation.

DEFINITION 2. Entailment *is a relation* $\Vdash \subseteq U \times U$ *defined by* $u \Vdash u'$ *iff,* $\forall\sigma$, $\sigma \models u$ *implies* $\sigma \models u'$.

The constraint system specified in this section is in a form suitable for HM(X) [40]. The formal statements and proofs are in the full version of the paper [12].

6. EVALUATION STRATEGIES

The constraint system of the previous section deliberately does not address the evaluation of function applications in constraints: Function applications are evaluated as terms in the sense of functional logic programming. In turn, functional logic programming knows a variety of different evaluation strategies with different tradeoffs [18, 19]. Thus, the evaluation strategy is a parameter to simplification.

The choice of evaluation strategy must take a number of pragmatic issues into account:

- For type-level functions reified at the value level the evaluation strategy should be the same at both levels.
- The `with closed` clause specifies that a given type class is closed by giving a fixed, final set of instances.
- There must be support for open type classes.[4]
- There may be no obvious order among the instances of a class (ruling out a sequential strategy).

We consider three evaluation strategies, each of which corresponds to a different strategy for resolving overloading:

outermost residuation with sequential matching
This strategy corresponds most closely to Haskell's constraint reduction with underlying open-world semantics. With a closed-world assumption, it is the strategy of choice for functions lifted from the data level.

outermost residuation with most-specific matching
This strategy is most useful for resolving Haskell-style overlapping instances.

outermost narrowing This strategy is useful for modeling Ada-style overloading. It also assumes a closed world.

As a prerequisite we assume that each function on the type level is defined by a terminating term rewriting system with rules of the form $f^\theta\ p_1 \ldots p_n = r$. θ is a strategy annotation and is one of the letters r, s, and n for "residuation," "specificity," and "narrowing," respectively. By convention p (with decoration) always stands for a constructor term. Moreover, write $f^\theta\ p_{j1} \ldots p_{jn} = r_j$ for the jth rule of the definition for function f^θ, and that m_{f^θ} is the number of rules for f^θ. Clearly, there is no t' so that $f^\theta\ t_1 \ldots t_n \xrightarrow{j} t'$ if $j > m_{f^\theta}$.

Each of the following subsections discusses one of the strategies. Each starts off with an exposition of the rewriting strategy. A definition of the corresponding rewriting rule is next, and finally we explain the applicability of this particular strategy in terms of one of the examples.

6.1 Residuation with Sequential Matching

Our first strategy is based on residuation [46]. It is less powerful than narrowing (it is incomplete) but it gives rise to a deterministic evaluation strategy.

We present a complete formulation of residuation with sequential matching. It corresponds to the typical way that functional programming languages interpret pattern matching. The strategy tries to match the equations in textual order and commits to the first matching equation. On the type level, this is the strategy of choice for function lifted from the data level to ensure that their type-level semantics are the same as their data-level semantics.

Even though residuation with sequential matching is a standard strategy [19], we present it in full because the presentation of specificity-based matching in Section 6.2 builds upon the definitions here.

Our notion of reduction is specified as a rewriting relation on type terms $t \to t$ and is shown in Figure 6. In addition to

[4] Haskell in particular, restricts instance declarations so that for each top-level type constructor there is at most one instance declaration. This restriction guarantees that reduction of predicates (aka reduction of type functions) is deterministic because addition of new instances neither changes the typing nor the meaning of an existing definition. The overlapping instances extension of Haskell has severe problems in connection with Haskell's open-world assumption.

$$\frac{f^r\ t_1 \dots t_n \xrightarrow{1} t'}{f^r\ t_1 \dots t_n \rightarrow t'}$$

$$\frac{\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Succ}\ \sigma}{f^r\ t_1 \dots t_n \xrightarrow{j} \sigma r_j} \quad \text{if } j \leq m_{f^r}$$

$$\frac{\begin{array}{c}\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Fail} \\ f^r\ t_1 \dots t_n \xrightarrow{j+1} t'\end{array}}{f^r\ t_1 \dots t_n \xrightarrow{j} t'} \quad \text{if } j \leq m_{f^r}$$

$$\frac{\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Red}\ t'_1 \dots t'_n}{f^r\ t_1 \dots t_n \xrightarrow{j} f\ t'_1 \dots t'_n} \quad \text{if } j \leq m_{f^r}$$

Figure 6: Sequential residuation strategy

the rules of a standard outermost reduction strategy, there are additional cases for dealing with logical variables. Evaluation regards logical variables as indeterminate values.

The definition of an outermost residuation step, $t \rightarrow t'$, rests on an auxiliary notion of reduction, $t \xrightarrow{j} t'$. Both relations are at most defined for terms t of the form $f^r\ t_1 \dots t_n$. The latter relation, $t \xrightarrow{j} t'$, holds if t rewrites to t' using rule number j or higher. The definition of $\rightarrow$ relies on a matching function $\mathcal{M}(\overline{t}, \overline{p})$ that takes a tuple of terms, $\overline{t}$, and a tuple of patterns, $\overline{p}$, both of the same length, and produces a match result. A match result is either

- *Succ* σ indicating a match with substitution σ;
- *Fail* indicating a match failure;
- *Suspend* indicating that $\overline{t}$ is not sufficiently instantiated to decide matching with p;
- *Red* $\overline{t}'$ indicating that the attempt to match $\overline{t}$ against $\overline{p}$ has forced an evaluation step from $\overline{t}$ to $\overline{t}'$ in one of the components.

Note that there is no t' such that $f^r\ t_1 \dots t_n \xrightarrow{j} t'$ if the matching function yields $\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Suspend}$.

Matching makes use of the subsidiary demand function $\mathcal{D}(t, p)$ that tries to evaluate the term t sufficiently so that syntactical matching with p is possible. Since $\mathcal{D}$ just drives the evaluation, it does not return a substitution but only signals with *Succ* that the term is sufficiently evaluated for matching to proceed syntactically.

For the correct and exhaustive definition of $\mathcal{D}$, we rely on an automaton that implements left-to-right matching. The states of the automaton form the following set Q:

$$\begin{array}{rcl} Q & = & \{(i, \mathit{Succ}) \mid i \in \mathbf{N}\} \cup \{\mathit{Suspend}, \mathit{Fail}\} \\ & \cup & \{(i, \mathit{Red}\ t) \mid i \in \mathbf{N}, t \in T_{\mathcal{C}\cup\mathcal{F}}(\mathcal{X})\} \end{array}$$

The meaning of *Succ*, *Suspend*, *Fail*, and *Red* t is as described above. The additional index i paired with *Succ* and *Red* determines a position in a tuple. In particular, $(i, \mathit{Red}\ t)$ means that the ith subterm has been reduced to t and must be replaced accordingly.

$q \setminus x$	*Succ*	*Suspend*	*Fail*	*Red* t'
(i, Succ)	$(i+1, \mathit{Succ})$	*Suspend*	*Fail*	$(i+1, \mathit{Red}\ t')$
Suspend	*Suspend*	*Suspend*	*Suspend*	*Suspend*
Fail	*Fail*	*Fail*	*Fail*	*Fail*
$(i, \mathit{Red}\ t)$	$(i, \mathit{Red}\ t)$	$(i, \mathit{Red}\ t)$	$(i, \mathit{Red}\ t)$	$(i, \mathit{Red}\ t)$

Figure 7: Demand state transition function

The table shown in Figure 7 defines the function $\delta(q, x)$, for $q \in Q$ and a match result x that accumulates the demand state of a list of terms. Typically, this list is the list of subterms of a particular term. The state remains *Succ* as long as the input symbol is *Succ*. At the same time, the index keeps track of the position in a list of terms. The input *Suspend* or *Fail* changes the state to *Suspend* or *Fail*. On input *Red* t, the state changes to $(i, \mathit{Red}\ t)$ to register the position where the reduction should happen. We write $\delta^*(q, \overline{\mathcal{D}(t_i, p_i)})$ for $\delta(\dots(\delta(q, \mathcal{D}(t_1, p_1)), \dots), \mathcal{D}(t_n, p_n))$.

$$\begin{array}{lcl}
\mathcal{D}(t, a) & = & \mathit{Succ} \\
\mathcal{D}(a, c\ p_1 \dots p_n) & = & \mathit{Suspend} \\
\mathcal{D}(f^r\ t_1 \dots t_n, p) & = & \left\{ \begin{array}{l} \mathit{Red}\ t' \\ \quad \text{if } p \text{ not a variable} \\ \quad \text{and } f^r\ t_1 \dots t_n \rightarrow t' \\ \mathit{Suspend} \\ \quad \text{if } p \text{ not a variable} \\ \quad \text{but } (\nexists t')\ f^r\ t_1 \dots t_n \rightarrow t' \end{array} \right. \\
\mathcal{D}(c\ t_1 \dots t_n, c'\ p_1 \dots p_{n'}) & = & \mathit{Fail} \text{ if } c \neq c' \text{ or } n \neq n' \\
\mathcal{D}(c\ t_1 \dots t_n, c\ p_1 \dots p_n) & = & \left\{ \begin{array}{ll} \mathit{Succ} & \text{if } q' = (n, \mathit{Succ}) \\ \mathit{Fail} & \text{if } q' = \mathit{Fail} \\ \mathit{Suspend} & \text{if } q' = \mathit{Suspend} \\ \mathit{Red}\ c\ t_1 \dots t'_i \dots t_n & \\ \quad \text{if } q' = (i, \mathit{Red}\ t'_i) & \end{array} \right. \\
& & \text{where } q' = \delta^*((0, \mathit{Succ}), \overline{\mathcal{D}(t_i, p_i)})
\end{array}$$

Figure 8: Demand function

Figure 8 shows the demand function $\mathcal{D}$. If the pattern is a variable, it signals *Succ*. If the term is a variable but the pattern is not, then $\mathcal{D}$ suspends because matching cannot proceed without further instantiation of the term. If the term is a function call and the pattern is not a variable, then $\mathcal{D}$ returns *Red* t' if the function call reduces, otherwise $\mathcal{D}$ returns *Suspend*. If term and pattern start with constructors that are either different or applied to different numbers of subterms, $\mathcal{D}$ returns *Fail*. If both term and pattern start with the same constructor applied to the same number of subterms, $\mathcal{D}$ is applied recursively to all corresponding subterms and subpatterns. If all recursive calls yield *Succ*, the result is *Succ*. If the first non-*Succ* result is *Fail*, the result is *Fail*. If the first non-*Succ* result is *Suspend*, the result is *Suspend*. If the first non-*Succ* result is *Red* t, the result is also *Red* but with t correctly replaced in the reduced term.

$$\begin{array}{l}
\mathcal{M}(t_1 \dots t_n, p_1 \dots p_n) = \\
\quad \left\{ \begin{array}{ll} \mathit{Succ}\ \sigma & \text{if } q' = (n, \mathit{Succ}) \\ & \text{and } (\forall 1 \leq i \leq n)\ \sigma p_i = t_i \\ \mathit{Fail} & \text{if } q' = (n, \mathit{Succ}) \\ & \text{and } (\forall \sigma \exists i)\ \sigma p_i \neq t_i \\ \mathit{Fail} & \text{if } q' = \mathit{Fail} \\ \mathit{Red}\ t_1 \dots t'_i \dots t_n & \text{if } q' = (i, \mathit{Red}\ t'_i) \\ \mathit{Suspend} & \text{if } q' = \mathit{Suspend} \end{array} \right. \\
\quad \text{where } q' = \delta^*((0, \mathit{Succ}), \overline{\mathcal{D}(t_i, p_i)})
\end{array}$$

Figure 9: Matching function

The matching function $\mathcal{M}$ shown in Figure 9 applies $\mathcal{D}$ recursively to all corresponding pairs, (t_i, p_i), of term and pattern. If all calls to $\mathcal{D}$ return *Succ*, then syntactic matching is possible and $\mathcal{M}$ returns *Succ* σ if $\sigma : \overline{p} \leq \overline{t}$ and *Fail* if no such substitution exists. $\mathcal{M}$ returns *Fail*, *Suspend*, and *Red* $\overline{t}$ analogously to $\mathcal{D}$.

In the definition of $\sigma \models u$ a notion of normalization $t \Downarrow t'$ is required which does not stop reduction at a constructor, but rather reduces the subterms of a data constructor as well. The definition of normalization does not distinguish partiality from non-termination: it either computes a constructor term t' or is undefined:

$$a \Downarrow a \qquad \frac{t_1 \Downarrow t_1' \quad \ldots \quad t_n \Downarrow t_n'}{c\ t_1 \ldots t_n \Downarrow c\ t_1' \ldots t_n'}$$

$$\frac{f^r\ t_1 \ldots t_n \rightarrow t' \quad t' \Downarrow t}{f^r\ t_1 \ldots t_n \Downarrow t}$$

On the basis of this rewriting relation, the constraint system can be extended by the following simplification rule for constraints:

$$\begin{array}{ll} (\textit{u-residuate}) & t_1 \doteq t_2 \rightsquigarrow t_1' \doteq t_2' \\ & \text{if } (t_1 \rightarrow t_1' \text{ and } t_2 = t_2') \\ & \text{or } (t_2 \rightarrow t_2' \text{ and } t_1 = t_1') \end{array}$$

PROPOSITION 4. *Rule (u-residuate) is sound and complete.*

This results extends Proposition 1. Given a confluent and termination term rewriting system, Propositions 2 and 3 extend, too. The structural properties of constraint entailment remain true, but we cannot hope for a principal constraint property, in general.

Figure 10 demonstrates the use of (*u-residuate*) for inferring the types of some expressions involving records.

6.2 Residuation with Most-Specific Matching

The residuation strategy with sequential matching is applicable when there is an obvious textual ordering of the instances belonging to a single class. In the context of Haskell, this is usually not the case, as the instances may be spread over several modules. Making the semantics of overloading depend on the order of the imports would be unsatisfactory. Hence, we consider a residuation strategy which is independent of the textual order of the instances and which deals directly with overlapping rules.

In the context of term rewriting, Kennaway [34] considers a *specificity rule* for ambiguous term rewriting systems: "The rule (really a rewriting strategy) stipulates that a term rewrite rule of the system can only be used to reduce a term which matches it, if that term can never match any other rule of the system which is more specific than the given rule. One rule is more specific than another if the left-hand side of the first rule is a substitution instance of the second, and the reverse is not true." This is exactly the right definition for our purposes. While Kennaway applies the rule by translating a system of equations into strongly sequential form, we embed specificity directly into our evaluation strategy.

We start off in a simplified setting. Let t be a constructor term and P be a set of patterns that appear as left-hand sides of equations. Let $U(t, P)$ be the set of patterns *unifiable* with t and $M(t, P)$ the set of patterns *matching* t.

$$\begin{array}{lcl} U(t,P) & = & \{p \in P \mid \exists t'.t \leq t' \wedge p \leq t'\} \\ M(t,P) & = & \{p \in P \mid p \leq t\} \end{array}$$

Clearly, each matching pattern is also a unifiable pattern and instantiating a term increases the set of matches.

PROPOSITION 5.

$$M(t,P) \subseteq U(t,P) \tag{1}$$

$$t \leq t' \Rightarrow M(t,P) \subseteq M(t',P) \tag{2}$$

We say that

- a pattern set P is *ambiguous* for t if $M(t, P)$ does not have a greatest element wrt. $\leq$.
- a pattern $p \in P$ is a *definite match* for t if $p \in M(t, P)$ and $\forall p' \in U(t, P).p' \leq p$.

A pattern p cannot be a definite match for t as long as there are patterns in P which are more specific than p and which are unifiable with t. Clearly, if σ_0 is a unifier of p'' and t, then $\sigma_0 t$ matches p as well as p''. Hence, the definition rules out all potential patterns like p''.

The original formulation of the specificity rule only deals with term rewriting systems. We also need to handle terms which contain residual function applications which are not sufficiently instantiated for further evaluation. Hence, we are interested in answering the following question: could the function calls in (an instance of) t eventually evaluate to something matching p? We make a very rough approximation to this property by replacing each function call in t by a fresh variable and trying to unify the resulting term with p. Hence, define $FA(t)$ as a constructor term so that

$$t = FA(t)[a_i \mapsto f_i\ \overline{t}_i'] \text{ where } a_i \notin t \text{ for all } i.$$

PROPOSITION 6.

$$M(t,P) = M(FA(t),P) \tag{1}$$

$$U(t,P) \subseteq U(FA(t),P) \tag{2}$$

$$\forall t'.(t \leq t' \wedge t' \Downarrow t'') \Rightarrow U(t'',P) \subseteq U(FA(t),P) \tag{3}$$

We extend our former definition:

- pattern $p \in P$ is a *final match* for t if $p \in M(t, P)$ and $\forall p' \in U(FA(t), P).p' \leq p$.

For defining the specificity-based strategy, we extend the definitions of M, U, and $FA()$ to tuples of terms $\overline{t}$ as well as sets of tuples of patterns $\overline{P}$ whenever the number of components of the tuples is the same throughout a set.

The specificity-based strategy tests all rewriting rules before making the decision. For a term $t = f^s\ t_1 \ldots t_n$, it computes a set L as follows:

- Each matching left-hand side contributes a pair of the form $\langle \overline{p}_j, \sigma r_j \rangle_M$ where $\overline{p}_j$ is a tuple of patterns (the left-hand side patterns of rule j) and σr_j is the instantiated right-hand side of the matching rule j.
- A left-hand side for which matching suspends contributes just its tuple of patterns $\langle \overline{p}_j \rangle_S$.

Now define $LP(L) = \{\overline{p} \mid \langle \overline{p}, r\rangle_M \in L \vee \langle \overline{p} \rangle_S \in L\}$, and let

$$\text{finalMatch}(L, \overline{t}) := \{\langle \overline{p}, t' \rangle \mid \langle \overline{p}, t' \rangle_M \in L, \overline{p} \text{ final match for } \overline{t} \text{ in } LP(L)\}.$$

Figure 11 defines the rewriting strategy as a reduction relation $t \xrightarrow{s} t'$ using an inference system for judgements of the form $L; t \xrightarrow{s,i} t'$ where L tracks the applicable matches. We write e, L for $\{e\} \cup L$.

Consider the generation of a record with one integer element at label `Lx`: `add EMPTY Lx 42`.
The trace shows how **`u-residuate`** implements Haskell's predicate reduction at the type level:

```
1. Assumption:            EMPTY            :: EMPTY
2. Assumption:            Lx               :: Lx
3. Assumption:            42               :: Int
4. Assumption:            add              :: (SELECT f r =:= NOTHING) => r -> f -> v -> CONS (f,v) r
5. (e-app) on 4 and 1:    add EMPTY        :: (SELECT f EMPTY =:= NOTHING) => f -> v -> CONS (f,v) EMPTY
6. (e-app) on 5 and 2:    add EMPTY Lx     :: (SELECT Lx EMPTY =:= NOTHING) => v -> CONS (Lx,v) EMPTY
7. (e-app) on 6 and 3:    add EMPTY Lx 42  :: (SELECT Lx EMPTY =:= NOTHING) => CONS (Lx,Int) EMPTY
8. (u-residuate) on the call to SELECT in 7 succeeds in the first step with substitution σ mapping f to Lx:
                          add EMPTY Lx 42  :: (NOTHING =:= NOTHING & f =:= Lx) => CONS (Lx,Int) EMPTY
9. (u-constr-dec) and the fact that f is not part of the type lets us get rid of the remaining constraints:
                          add EMPTY Lx 42  :: CONS (Lx,Int) EMPTY
```

Figure 10: Type Derivation for Records

$$\frac{\varnothing, f^s\ t_1 \dots t_n \stackrel{s,1}{\to} t'}{f^s\ t_1 \dots t_n \stackrel{s}{\to} t'}$$

$$\frac{\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Succ}\ \sigma \qquad \langle \overline{p}_j, \sigma r_j \rangle_M, L; f^s\ t_1 \dots t_n \stackrel{s,j+1}{\to} t'}{L; f^s\ t_1 \dots t_n \stackrel{s,j}{\to} t'} \quad \text{if } j \leq m_{f^s}$$

$$\frac{\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Fail} \qquad L; f^s\ t_1 \dots t_n \stackrel{s,j+1}{\to} t'}{L; f^s\ t_1 \dots t_n \stackrel{s,j}{\to} t'} \quad \text{if } j \leq m_{f^s}$$

$$\frac{\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Red}\ t'_1 \dots t'_n}{L; f^s\ t_1 \dots t_i \dots t_n \stackrel{s,j}{\to} f^s\ t'_1 \dots t'_n} \quad \text{if } j \leq m_{f^s}$$

$$\frac{\mathcal{M}(t_1 \dots t_n, p_{j1} \dots p_{jn}) = \mathit{Suspend} \qquad \langle \overline{p}_j \rangle_S, L; f^s\ t_1 \dots t_n \stackrel{s,j+1}{\to} t'}{L; f^s\ t_1 \dots t_n \stackrel{s,j}{\to} t'}$$

$$\frac{\langle \overline{p}, t' \rangle \in \text{finalMatch}(L, t_1 \dots t_n)}{L; f^s\ t_1 \dots t_n \stackrel{s,j}{\to} t'} \quad \text{if } j > m_{f^s}$$

Figure 11: Specificity-based strategy

We apply the overloaded `equal` member value of the `EQUAL` class from Section 2.3 to two character literals: `equal 'x' 'y'`.

1. Assumption after (*e-spec*):
   `equal :: (r =:= EQUAL Char Char) => Char -> Char -> r`
2. `'x', 'y' :: Char`
3. `equal 'x' 'y' :: (r =:= EQUAL Char Char) => r`
4. (*u-residuate-ms*) on the constraint of 3 leads to `r =:= TRUE` where
   $\langle a, a, \texttt{TRUE} \rangle \in \text{finalMatch}(\varnothing, \texttt{Char}, \texttt{Char})$
   $\varnothing, \texttt{EQUAL Char Char} \stackrel{s,1}{\to} \texttt{TRUE}$
   $\texttt{EQUAL Char Char} \stackrel{s}{\to} \texttt{TRUE}$
5. applying the substitution yields `equal 'x' 'y' :: TRUE`

Figure 12: Example with Most-Specific Matching

In the same way as for the sequential evaluation strategy, the specificity-based strategy induces an evaluation relation:

$$a \stackrel{s}{\Downarrow} a \qquad \frac{t_1 \stackrel{s}{\Downarrow} t'_1 \quad \dots \quad t_n \stackrel{s}{\Downarrow} t'_n}{c\ t_1 \dots t_n \stackrel{s}{\Downarrow} c\ t'_1 \dots t'_n}$$

$$\frac{f^s\ t_1 \dots t_n \stackrel{s,1}{\to} t' \quad t' \stackrel{s}{\Downarrow} t}{f^s\ t_1 \dots t_n \stackrel{s}{\Downarrow} t}$$

The integration of the specificity-based strategy into the type inference engine gives rise to a rule (*u-residuate-ms*) which is analogous to (*u-residuate*).

Proposition 7. *Rule (u-residuate-ms) is sound and complete with respect to* $\stackrel{s}{\Downarrow}$.

Figure 12 shows an example derivation using residuation with most-specific matching.

6.3 Narrowing

Assuming a closed world, we need not defer the expansion of a function to the point where its parameters are sufficiently known. In particular, we can derive negative information and hence determine failures earlier.

Narrowing comes in several eager and lazy variants [18]. Unfortunately, a narrowing step may introduce a choice of different alternatives. This choice is often expressed by non-deterministically rewriting a term into a term paired with a substitution. Formally, $f^n\ t_1 \dots t_n \rightsquigarrow_\sigma \sigma r$ if $f^n\ p_1 \dots p_n = r$ is a variant of a rule for f^n and σ is the most general (syntactic) unifier of $\overline{t}$ and $\overline{p}$. The non-determinism arises due to the choice among the rules which are unifiable with $\overline{p}$. In our constraints, the non-determinism appears in the form of disjunction.

Provided that the underlying term rewriting system is confluent and terminating, narrowing yields a sound and complete E-unification strategy [24]. A sufficient condition for confluence is orthogonality of the rewrite rules (the left-hand sides of the rules are pairwise disjoint).

Here is the appropriate addition to the constraint simplification system that implements the narrowing strategy ($;_{j=1}^m$ means a disjunction of constraints for $j = 1, \dots, m$):

$$(\textit{u-narrow}) \quad u \rightsquigarrow ;_{j=1}^m \exists \overline{a}_j . t_1 \doteq t'_{j1} \,\&\, \dots \,\&\, t_n \doteq t'_{jn} \,\&\, t \doteq r'_j$$
if ($u = f^n\ t_1 \dots t_n \doteq t$ or $u = t \doteq f^n\ t_1 \dots t_n$)
and t not a variable
and $f^n\ t'_{j1} \dots t'_{jn} = r'_j$ for $1 \leq j \leq m$
are fresh variants of f^n's defining rules
and $\overline{a}_j$ are the free variables in the jth rule

Proposition 8. *Rule (u-narrow) is sound and complete.*

The use of narrowing in type reduction can lead to better types and earlier error detection. On the other hand, narrowing it can also lead to complex and potentially unreadable types.

Figure 13 considers the example from Section 2.4. The interaction of the two uses of narrowing allows the type checker to "narrow down" the type to the point where it becomes unique. The effect is similar to the two-pass algorithm for overloading resolution for Ada given in the Dragon book [1]. The important difference is that our strategy does not require that an expression has a unique type, but rather defers the final elaboration by moving the remaining predicates into the type's context.

Narrowing is only applicable for mutually disjoint sets of patterns. In particular, narrowing in combination with most-specific matching can lead to problems. To see this, consider the function f defined by $f\ C = D$ and $f\ a = a$, where C and D are nullary constructors. Consider further the equation $f\ C \doteq C$. By definition, a narrowing step with this definition of f leads to $[C \doteq C \,\&\, D \doteq C; a \doteq C \,\&\, a \doteq C;]$. This predicate simplifies to $[\mathit{FAIL}; a \doteq C;]$ and then to $a \doteq C$. This outcome is wrong since the failure to unify the actual result of f with the expected result is interpreted as failure to match the argument with the expected argument. The correct simplification of $f\ C \doteq C$ would be FAIL.

7. IMPLEMENTATION NOTES

We have implemented a prototype frontend for the variant of Haskell used in the examples of this paper. In particular, we have adapted a Haskell frontend [?] to the new syntax, added a dependency analysis and a kind inference pass as well as a translation to essentially the form required by Jones's Typing-Haskell-in-Haskell type checker [30]. We have adapted the type checker to HM(X), and implemented the constraint simplification rules in Figure 5. The constraint simplifier calls an evaluation engine for function applications based on *definitional trees* [19], a representation for functional logic programs allowing fine-grain control over the evaluation strategy. Currently, we have instance translation functions implementing sequential residuation and narrowing as described in Sections 6.1 and 6.3.

8. RELATED WORK

There have been many approaches to adding overloading to languages with a Hindley/Milner style polymorphic type system, beginning with Kaes [33] and Wadler and Blott [48] and later picked up, refined, and implemented by many others [39, 5, 38, 3, 26, 41, 15, 27, 29, 42, 35, 31]. In particular, recent work is pushing hard the borders of complete and decidable type inference [44, 45]. In the face of this plethora, we only consider the most closely related work here.

The work of Jones [27] and its extension to constructor classes [26] provides a general framework for type inference with qualified types which (still) subsumes the facilities present in Haskell 98. The framework of qualified types is not sufficiently expressive for our purposes because it neither supports disjunction nor a closed-world assumption. Still, much of our inspiration comes from Jones's work on improvement of predicates [29] and from its implementation via functional dependencies [31].

The HM(X) framework [40] generalizes various constraint-based type inference systems, *e.g.*, for record types and for object types. It can be instantiated to Haskell's type classes and can handle open-world as well as closed-world theories. HM(X) provides us with a parameterized type inference engine and completeness results.

A recent proposal that employs constraint handling rules (CHR) to model type classes [14] is also based on HM(X). In this proposal, the semantics of predicate reduction is formally defined as the rewriting relation induced by the CHR. Semantic properties such as ambiguity can be decided by checking certain properties of CHRs. The authors demonstrate the generality of CHRs and their suitability to encode closed-world theories, too. Much of the thrust of our work is also on formally specifying the semantics of constraint simplification. However, a key innovation of our proposal is the customizability of the evaluation strategy.

Duggan and others [10] have proposed kinded parametric overloading for a variant of ML. They define a kind structure similar to a type class from accumulated overloaded value definitions. They have open kinds (corresponding to an open-world theory) as well as closed kinds (sic), but they provide a fixed type inference engine for their language. In contrast, we specify a parameterized modular type inference system based on HM(X).

Shields and Peyton Jones [45] discuss various ad-hoc extensions of Haskell's type system with the goal of exploring the design space. The main thrust of their extensions is the interoperability between Haskell and the object-oriented language C$^\sharp$. In particular, they present an encoding of subtyping, ad-hoc overloading in a style similar to Java, and a general notion of overlap. Technically, they develop a type inference engine and give an overview of its technical properties—termination, completeness, etc. Our approach is incomparable in power with their proposal. We can deal with most of their extensions, except the resolution of Java-style overloading. We expect that their proposed solution (define a partial order on type classes) could also be made to work with our most-specific matching strategy. On the other hand, the narrowing approach to resolve Ada-style overloading is unique to our system. In addition, our system extends simply and modularly just by specifying a new rewrite strategy.

Further work on type-level programming indicates widespread interest in the subject. There are applications [37, 36, 16, 47] as well as theoretical investigations starting from a variety of foundations. Dependent types are certainly the ultimata ratio in type-level programming. Cayenne [4] is a Haskell dialect that builds on dependent type theory. Cayenne is much more radical than the present work because it builds on a richer type theory, where the entire term language is encorporated in the type language. While this is an interesting proposition for future work, we only allow certain term rewriting systems at present.

An approach towards integrating dependent types into a full programming language is the language DML(C) [50]. DML(C) allows for types indexed with constraints from a constraint domain C. This approach is also incomparable with our proposal. While DML(C) can incorporate semantically rich constraint theories and thus guarantee a decidable type checking algorithm, our constraint theory is in principle fixed but still variable due to the underlying term rewriting system and the choice of strategy for each function.

Intensional type analysis [21, 7, 6, 49] is an approach to defining functions by induction on the structure of types. These works are closer to generic programming [22, 25]. The commonality is that their scheme of function definition is much more rigid than with our approach. Usually, the inspection of the type structure is limited to a fold operation. In contrast, we can examine the type structure using a term rewriting system, which can even be ambiguous.

There is a plethora of different strategies for performing narrowing and residuation and each has its benefits. Definitional trees [19] serve as a mechanism for fine-grain spec-

Narrowing obtains the unambiguous typing `f :: Int -> Int -> Int` (cf. Section 2.4) via the following derivation:

1. Assumptions:	`x :: x, y :: y, (+) :: (c =:= PLUS a b) => a -> b -> c`
2. (*e-app*) on 1:	`(+ x) :: (c =:= PLUS x b) => b -> c`
3. (*e-app*) on 2 and 1:	`x + y :: (c =:= PLUS x y) => c`
4. (*e-app*) on 1 and 3:	`(+) (x + y) :: (f =:= PLUS c e & c =:= PLUS x y) => e -> f`
5. (*e-app*) on 4 and `42 :: Int`:	`(x + y) + 42 :: (f =:= PLUS c Int & c =:= PLUS x y) => f`
From now on, the term remains the same and is omitted.	
6. (*u-narrow*) for first `PLUS` on 5:	`(((c =:= Int & Int =:= Int & f =:= Int)`
	`;(c =:= Float & Int =:= Float & f =:= Float)`
	`;(exists q. c =:= [q] & Int =:= [q] & f =:= [q]))`
	`& c =:= PLUS x y) => f`
7. (*u-choice-and*) on 6:	`((c =:= Int & Int =:= Int & f =:= Int & c =:= PLUS x y)`
	`;(c =:= Float & Int =:= Float & f =:= Float & c =:= PLUS x y)`
	`;(exists q. c =:= [q] & Int =:= [q] & f =:= [q] & c =:= PLUS x y)) => f`
8. (*u-constr-dec*), (*u-constr-fail*) on 7:	`((c =:= Int & Success & f =:= Int & c =:= PLUS x y)`
	`;(c =:= Float & Fail & f =:= Float & c =:= PLUS x y)`
	`;(exists q. c =:= [q] & Fail & f =:= [q] & c =:= PLUS x y)) => f`
9. (*u-and-fail*) and (*u-exists-drop*) on 8:	`((c =:= Int & Success & f =:= Int & c =:= PLUS x y) ;Fail ;Fail) => f`
10. (*u-choice-fail*) and (*u-and-success*) on 9:	`(c =:= Int & f =:= Int & c =:= PLUS x y) => f`
11. (*u-subst*) on 10:	`(Int =:= PLUS x y) => Int`
12. (*u-narrow*) on 11:	`((x =:= Int & y =:= Int & Int =:= Int)`
	`;(x =:= Float & y =:= Float & Int =:= Float)`
	`;(exists q. x =:= [q] & y =:= [q] & Int =:= [q]) ) => Int`
13. (*u-constr-dec*), (*u-constr-fail*), (*u-and-success*), (*u-and-fail*), (*u-exists-drop*), and (*u-choice-fail*) on 12:	
	`(x =:= Int & y =:= Int) => Int`

Figure 13: An Example Derivation with Narrowing

ification of evaluation strategies. In particular, the Curry language [20] allows the programmer to annotate equations with evaluation annotations specifying the evaluation strategy, similar to our approach.

9. CONCLUSION

We have presented a programmable approach to the implementation of overloading. The introduced transition from type-classes-as-predicates to type-classes-as-functions allows for natural formulations of solutions of many practical overloading problems, among them the classic `sprintf` problem, the handling of overlapping instances as well as Ada-style overloading. We have designed a concrete variant of the Haskell language which supports functional logic overloading, and implemented a prototype frontend for it. The type system needed for handling this style of overloading is based on parameterizing the HM(X) framework with unification constraints. Constraint simplification calls an evaluation engine for functional logic programs when it encounters applications of type-level functions. The choice of an evaluation strategy for type-level functions opens a spectrum of design choices for this style of overloading. Different applications benefit from different evaluation strategies. It remains to be seen what combination of evaluation strategies is most appropriate for practical use, and what degree of control over it the language should offer the programmer.

10. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Arvind, editor. *Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press, New York.

[3] L. Augustsson. Implementing Haskell overloading. In Arvind [2], pages 65–73.

[4] L. Augustsson. Cayenne—a language with dependent types. In Hudak [23].

[5] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 170–181, San Francisco, California, USA, June 1992.

[6] K. Crary and S. Weirich. Flexible type analysis. In P. Lee, editor, *Proc. International Conference on Functional Programming 1999*, pages 233–248, Paris, France, Sept. 1999. ACM Press, New York.

[7] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In Hudak [23], pages 301–312.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[9] O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, Nov. 1998.

[10] D. Duggan, G. V. Cormack, and J. Ophel. Kinded type inference for parametric overloading. *Acta Inf.*, 33(1):21–68, 1996.

[11] J. H. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Comput. Sci.*, 67((2+3)):203–260, 1989.

[12] M. Gasbichler, M. Neubauer, M. Sperber, and P. Thiemann. Functional logic overloading. Technical Report 163, Institut für Informatik, University of Freiburg, Germany, Nov. 2001. Available from `ftp://ftp.informatik.uni-freiburg.de/documents/reports/report163/r`

[13] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.

[14] K. Glynn, P. Stuckey, and M. Sulzmann. Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming*, July 2000.

[15] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. In D. Sannella, editor, *Proc. 5th European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 241–256, Edinburgh, UK, Apr. 1994. Springer-Verlag.

[16] T. Hallgren. Fun with functional dependencies. In *Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Göteborg University*, Varberg, Sweden, Jan. 2001.

http://www.cs.chalmers.se/~hallgren/Papers/wm01.html.

[17] M. Hanus. On the completeness of residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 192–206. MIT Press, 1992.

[18] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19,20:583–628, 1994.

[19] M. Hanus. A unified computation model for functional and logic programming. In Jones [32], pages 80–93.

[20] M. Hanus. Curry – an integrated functional logic language. http://www.informatik.uni-kiel.de/~mh/curry/report.html, June 2000. Version 0.7.1.

[21] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, Jan. 1995. ACM Press.

[22] R. Hinze. A new approach to generic functional programming. In Reps [43], pages 119–132.

[23] P. Hudak, editor. *International Conference on Functional Programming*, Baltimore, USA, Sept. 1998. ACM Press, New York.

[24] J.-M. Hullot. Canonical forms and unification. In R. Kowalski, editor, *Proceedings of the Fifth International Conference on Automated Deduction (Les Arcs, France)*, number 87 in Lecture Notes in Computer Science, pages 318–334, Berlin, July 1980. Springer-Verlag.

[25] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In Jones [32], pages 470–482.

[26] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In Arvind [2], pages 52–61.

[27] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.

[28] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, May 1995.

[29] M. P. Jones. Simplifying and improving qualified types. In S. Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 160–169, La Jolla, CA, June 1995. ACM Press, New York.

[30] M. P. Jones. Typing Haskell in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical Reports, 1999. ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf.

[31] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 230–244, Berlin, Germany, Mar. 2000. Springer-Verlag.

[32] N. D. Jones, editor. *Proc. 24th Annual ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[33] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symposium on Programming 1988*, number 300 in Lecture Notes in Computer Science, pages 131–144. Springer-Verlag, 1988.

[34] R. Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In N. D. Jones, editor, *Proc. 3rd European Symposium on Programming 1990*, number 432 in Lecture Notes in Computer Science, pages 256–270, Copenhagen, Denmark, 1990. Springer-Verlag.

[35] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In Reps [43], pages 108–118.

[36] C. McBride. Faking it—simulating dependent types in Haskell. http://www.dur.ac.uk/~dcs1ctm/faking.ps, 2001.

[37] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In R. Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, 2001. to appear.

[38] T. Nipkow and C. Prehofer. Type checking type classes. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 409–418, Charleston, South Carolina, Jan. 1993. ACM Press.

[39] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In J. Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, number 523 in Lecture Notes in Computer Science, pages 1–14, Cambridge, MA, 1991. Springer-Verlag.

[40] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[41] J. Peterson and M. Jones. Implementing type classes. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 227–236, Albuquerque, New Mexico, June 1993.

[42] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In J. Launchbury, editor, *Proc. of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997. Yale University Research Report YALEU/DCS/RR-1075.

[43] T. Reps, editor. *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, Boston, MA, USA, Jan. 2000. ACM Press.

[44] M. Shields and E. Meijer. Type-indexed rows. In H. R. Nielson, editor, *Proc. 28th Annual ACM Symposium on Principles of Programming Languages*, pages 261–275, London, Jan. 2001. ACM Press.

[45] M. Shields and S. Peyton Jones. Object-oriented style overloading for Haskell. In *BABEL '01. First Workshop on Multi-Language Infrastructure and Interoperability*, Florence, Italy, Sept. 2001.

[46] P. A. Subrahmanyam and J.-H. You. FUNLOG: A computational model integrating logic programming and functional programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[47] P. Thiemann and M. Sperber. Program generation with class. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Proceedings Informatik'97*, Reihe Informatik aktuell, pages 582–592, Aachen, Sept. 1997. Springer-Verlag.

[48] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, Jan. 1989. ACM Press.

[49] S. Weirich. Encoding intensional type analysis. In D. Sands, editor, *Proc. 10th European Symposium on Programming*, Lecture Notes in Computer Science, Genova, Italy, Apr. 2001. Springer-Verlag.

[50] H. Xi and F. Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, USA, Jan. 1999. ACM Press.