Pragmatic Subtyping in Polymorphic Languages

Johan Nordlander

Department of Computing Science
Chalmers University of Technology
S - 412 96 Göteborg, Sweden
Email: nordland@cs.chalmers.se

Abstract

We present a subtyping extension to the Hindley/Milner type system that is based on *name inequivalence*. This approach allows the subtype relation to be defined by incremental construction of polymorphic records and datatypes, in a way that subsumes the basic type systems of both languages like ML and Java. As the main contribution of the paper, we describe a partial type inference algorithm for the extended system which favours succinctness over generality, in the sense that it never infers types with subtype constraints. The algorithm is based on an efficient approximating constraint solver, and is able to type a wide range of programs that utilize subtyping and polymorphism in a non-trivial way. Since constrained types are not inferred, the algorithm cannot be complete; however, we provide a completeness result w. r. t. the Hindley/Milner type system as a form of characterizing lower bound.

1 Introduction

The combination of subtyping with polymorphic type inference has been under intensive study for more than a decade [Mit84, FM90, FM89, Mit91, Kae92, Smi94, AW93, EST95, Hen96, Reh97, MW97]. From the outset, this line of research has focused on *complete* inference algorithms, and the related notion of principal types. This direction is not hard to justify considering the evident merits of polymorphic languages like ML or Haskell: program fragments can be typed independently of their context, and programmers may rest assured that any absent type information will be filled in with types that are at least as general, and at least as succinct, as any information the programmer would have come up with.

Still, although complete algorithms for polymorphic subtype inference exist, practical language implementations that take advantage of these are in short supply. The main reason seems to be that "the algorithms are inefficient and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read" [HM95]. Many attempts have been made at simplifying the output from these algorithms, but they have only partially succeeded, since the problem in its generality seems to be intractable, both in theory and practice [HM95, Reh97].

However, even if type simplification were not an issue, there is an inherent conflict between generality and succinctness in polymorphic subtyping that is not present in the original ML type system. While the principal type of an ML expression is also the syntactically shortest type, the existence of subtype constraints in polymorphic subtyping generally makes a principal type *longer* than its instances. In particular, the principal type for a given expression may be substantially more complex than the simplest type general enough to cover an *intended* set of instances! Thus, type annotations, which give the programmer direct control over the types of expressions, are likely to play a more active role in languages with polymorphic subtyping than they do in ML, irrespective of advances in simplification technology.

In this paper we take a pragmatic standpoint and embrace type annotations as a fact of life. This enables us to focus on the much simpler problem of *partial* polymorphic subtype inference. As the main contribution of the paper, we present an inference algorithm that always infers types without subtype constraints, if it succeeds. This is a particularly interesting compromise between implicit and explicit typing, since such types possess the ML property of being syntactically shortest among their instances, even though they might not be most general. We might say that the algorithm favours readability over generality, leaving it to the programmer to put in type annotations where this strategy is not appropriate. An exact characterization of which programs our algorithm actually is able to accept is still an open problem, but we prove, as a lower bound, that it is complete with respect to the ML type system. We will also provide some evidence why the algorithm is likely to work very well in practice.

A second contribution of the paper is presumably the subtype relation itself, which is based on *name inequivalence*, in contrast to the structural ditto that dominate the standard literature [CW85, Mit84]. By structural we mean the common practice of defining special subtyping rules for functions, records, and variants, etc, assuming a given partial order over a set of nullary base types. We will instead assume that type constants can be of any arity, and extend the partial order on base types to a relation on fully saturated applications of these constants. Functions, records, and variants can then be seen as special cases in this general framework.

We believe that working with named and parameterized types, whose subtype relationships are defined by declaration, has the immediate benefit of giving the programmer full control over the type structure of a program. This can be a valuable tool in the design phase of a large system, and it also offers greater protection from certain logical errors. Furthermore, name inequivalence seems more in line with both the way datatypes are treated in functional languages, and with the object-oriented notion of a named class. And not the least, the ability to refer to types by name has a big impact on the readability of the output from our inference algorithm.

The rest of the paper is organized as follows. Next section continues with some motivating examples, before the formal type system is introduced in section 3. In section 4 the inference algorithm is described and its theoretical properties are investigated. In section 5 we show how records and datatypes fit into our framework. Section 6 contains a more practical exploration of the algorithm, by means of some programming examples. In section 7, type checking in the presence of incomplete type inference is addressed. Related work is surveyed in section 8, and section 9 concludes.

2 Motivating examples

In polymorphic subtyping, principal types generally require subtype constraints, as is easily demonstrated by the following archetypical example:

```
twice f x = f (f x)
```

Assuming `Int` is a subtype of `Real`, this function can have the type `(Real->Int)->Real->Int`, as well as `(Int->Int)->Int->Int`, which forces the principal type of `twice` to be as general as `(a->b)->a->b | b < a`.[1]

However, `twice` is certainly not a good representative of subtyping as it is used in common object-oriented languages. Here the type systems are mostly monomorphic, which simplifies the typing problem considerably. Still, the elegance of subtyping is captured perfectly well in these languages: great flexibility is achieved without introducing any additional burden on the programmer writing flexible code.

Our approach to polymorphic subtype inference is to let functions like `twice` retain their original Hindley/Milner type, and, in the spirit of object-oriented languages, support subtyping only when types involved are known. This choice can be justified on the grounds that `(a->a)->a->a` is still likely to be a sufficiently general type for `twice` in most situations, and that the benefit of a consistently readable output from the inference algorithm will arguably outweigh the inconvenience of having to supply a type annotation when this is not the case. We certainly do not want to prohibit exploration of the more elaborate areas of polymorphic subtyping that need constraints, but considering the cost involved, we think it is reasonable to expect the programmer to supply the type information in these cases.

As an example of where the lack of inferred subtype constraints might seem more unfortunate than in the typing of `twice`, consider the function

```
min x y = if x < y then x else y
```

which, assuming `<` is a relation on `Real`s, will be assigned the type `Real -> Real -> Real` by our algorithm. A more useful choice would probably have been `a->a->a | a < Real` here, but as we have indicated, such a constrained type can only be attained by means of an explicit type annotation in our system. On the other hand, note that the principal type for `min`, `a->b->c | a < Real, b < Real, a < c, b < c`, is a yet more complicated type, and presumably an overkill in any realistic context.

So, an informal characterization of our inference algorithm is that it, in contrast to ML-style type inference, allows subtyping steps at application nodes when the types are known, as in e. g. `sin 1`. In addition, the algorithm computes least upper bounds for instantiation variables when required, so that for example the list `[7, 3.14]` will get the type `[Real]`. Greatest lower bounds for anonymous function arguments will also be found, resulting in the inferred type `Int -> (Bool,Int)` for the term `\x -> (x < 3.14, x + 1)`. Notice, though, that the algorithm assigns constraint-free types to *all* subterms of an expression, hence a compound expression might receive a less general type even though its principal type has no constraints. One example of this is

```
let twice f x = f (f x) in twice trunc 3.14
```

which is assigned the type `Real`, not the principal type `Int`.

As a demonstration of what name inequivalence can look like in practice, we define two polymorphic record types that capture a minimal set of equality and comparison operations, respectively.

```
struct Eq a = {eq :: a -> a -> Bool}
struct Ord a < Eq a = {less :: a -> a -> Bool}
```

The relation declared here states that for all types `a`, an object of type `Ord a` also supports the operations of `Eq a`. Since `a` only occurs contravariantly in `Ord`, depth subtyping can furthermore be used to derive that `Ord Real` a subtype of `Ord Int` (which in turn is a subtype of `Eq Int`). It may be worthwhile to have this example in mind when the subtype relation is introduced in the next section.

3 Type system

The starting point of our work is the let-polymorphic type system of ML (a.k.a the Hindley/Milner system), extended with subtype constraints, a subtype relation, and a subsumption rule. Such extensions are well represented in the literature [Kae92, Smi94, AW93, EST95]; in this paper we will adopt a formulation by Henglein [Hen96], which has the interesting merit of being "generic" in its subtyping theory. Since our system is obtained by "instantiation" with a particular form of subtyping theory, the results of [Hen96] directly carry over to our case. The generic part of this type system is shown in figure 1.

In our formulation, type variables are ranged over by α and β, while type constants are ranged over by t and s. The *arity* of a type constant t is denoted n_t. Type constants are considered drawn from some finite set that contains at least the function type constructor $(\to)$ with arity 2. Substitutions are ranged over by Φ. For notational convenience, we will make use of the following abbreviations:

$$\begin{array}{rcl} \tau \to \rho & \equiv & (\to)\ \tau\ \rho \\ \tau|D & \equiv & \forall\emptyset.\tau|D \\ \tau & \equiv & \tau|\emptyset \\ C \vdash_P D & \equiv & C \vdash_P \tau \leq \rho \text{ for all } \tau \leq \rho \in D \end{array}$$

We will also assume that terms have been renamed so that all bound variables are distinct and not equal to any free variables.

[1] The concrete syntax used here separates a type and its constraints by the symbol `|`.

Term language:

$$\begin{array}{llll} e & ::= & x & \textit{variable} \\ & | & e\,e' & \textit{application} \\ & | & \lambda x.e & \textit{abstraction} \\ & | & \mathtt{let}\,x = e\,\mathtt{in}\,e' & \textit{local definition} \end{array}$$

Type language:

$$\begin{array}{lll} \tau,\rho & ::= & \alpha \mid t\,\tau_1 \ldots \tau_{n_t} \\ \sigma & ::= & \forall\overline{\alpha}.(\tau|C) \\ C, D & ::= & \{\tau \leq \rho\}^* \\ \Gamma & ::= & \{x : \sigma\}^* \end{array}$$

Typing rules:

$$\frac{}{C, \Gamma \cup \{x : \sigma\} \vdash_P x : \sigma}\ (\textsc{var})$$

$$\frac{C, \Gamma \vdash_P e : \tau' \to \tau \quad C, \Gamma \vdash_P e' : \tau'}{C, \Gamma \vdash_P e\,e' : \tau}\ (\textsc{app})$$

$$\frac{C, \Gamma \cup \{x : \tau'\} \vdash_P e : \tau}{C, \Gamma \vdash_P \lambda x.e : \tau' \to \tau}\ (\textsc{abs})$$

$$\frac{C, \Gamma \vdash_P e : \sigma \quad C, \Gamma \cup \{x : \sigma\} \vdash_P e' : \tau}{C, \Gamma \vdash_P \mathtt{let}\,x = e\,\mathtt{in}\,e' : \tau}\ (\textsc{let})$$

$$\frac{C \cup D, \Gamma \vdash_P e : \tau \quad \overline{\alpha} \notin fv(C, \Gamma)}{C, \Gamma \vdash_P e : \forall\overline{\alpha}.\tau|D}\ (\textsc{gen})$$

$$\frac{C, \Gamma \vdash_P e : \forall\overline{\alpha}.\tau|D \quad C \vdash_P [\overline{\tau}/\overline{\alpha}]D}{C, \Gamma \vdash_P e : [\overline{\tau}/\overline{\alpha}]\tau}\ (\textsc{inst})$$

$$\frac{C, \Gamma \vdash_P e : \tau \quad C \vdash_P \tau \leq \tau'}{C, \Gamma \vdash_P e : \tau'}\ (\textsc{sub})$$

Figure 1: The basic type system

It can easily be verified that if $C, \Gamma \vdash_P e : \forall\overline{\alpha}.\tau|D$ is derivable without the use of rule ($\textsc{sub}$), and if all type schemes in Γ have empty constraint sets, then $\emptyset, \Gamma \vdash_P e : \forall\overline{\alpha}.\tau$ is derivable without rule ($\textsc{sub}$) and with empty constraint sets throughout (i. e. the judgement is also derivable in the original Hindley/Milner type system). Let $\Gamma \vdash^{HM} e : \sigma$ denote such a derivation. In section 4 we will prove a completeness result w. r. t. derivations of this limited form.

3.1 Subtype relation

Figure 2 shows the inference rules for our subtype relation. With the exception of rules ($\textsc{depth}$) and ($\textsc{const}$), this definition directly corresponds to the one in [Hen96].

The *variance* of a type constant, utilized in rule ($\textsc{depth}$), is defined as follows:

Definition 1 (Variance) *For a type constant t, let the sets $t^+, t^- \subseteq \{1 \ldots n_t\}$ represent the argument positions that the subtype relation should treat as co- and contravariant, respectively.*

$$\frac{}{C \cup \{\tau \leq \rho\} \vdash_P \tau \leq \rho}\ (\textsc{hyp})$$

$$\frac{}{C \vdash_P \tau \leq \tau}\ (\textsc{refl})$$

$$\frac{C \vdash_P \tau \leq \rho \quad C \vdash_P \rho \leq \tau'}{C \vdash_P \tau \leq \tau'}\ (\textsc{trans})$$

$$\frac{(C \vdash_P \tau_i \leq \rho_i)^{i \in t^+} \quad (C \vdash_P \rho_j \leq \tau_j)^{j \in t^-}}{C \vdash_P t\,\tau_1 \ldots \tau_{n_t} \leq t\,\rho_1 \ldots \rho_{n_t}}\ (\textsc{depth})$$

$$\frac{\tau <_P \rho}{C \vdash_P \Phi\tau \leq \Phi\rho}\ (\textsc{const})$$

Figure 2: Subtype relation

For a built-in type, these choices must of course be consistent with the dynamic semantics of terms of that type, which for the function symbol means that $(\to)^+ = \{2\}$, and $(\to)^- = \{1\}$ (see [CW85]). Thus, rule ($\textsc{depth}$) replaces and generalizes the standard rule for functions (called ($\textsc{arrow}$) in [Hen96]). As regards types defined by the programmer, section 5 will describe how variance information can be extracted from record and datatype declarations.

Our basic subtyping theory is defined in terms of polymorphic *subtype axioms* that may be instantiated by substitution, as witnessed by rule ($\textsc{const}$) in figure 2.

Definition 2 (Subtype axiom) *If $t\,\overline{\tau}$ and $s\,\overline{\rho}$ are ground type expressions, $t \neq s$, and $\overline{\tau}$ and $\overline{\rho}$ contain no occurrences of t and s, then the term $(t\,\overline{\tau} < s\,\overline{\rho})$ is a* subtype axiom *relating t and s.*

We consider subtype axioms equivalent up to renaming of variables. To turn a subtype axiom into a judgement, we write $(\tau < \rho) \in S$, or more conveniently $\tau <_S \rho$, where S is some given set.

The interaction between multiple subtype axioms is controlled by the following definition:

Definition 3 (Subtyping theory) *A* subtyping theory *P is a set of subtype axioms such that*

1. *all axioms in P relate distinct pairs of type constants.*
2. *if $t\,\overline{\tau} <_P s\,\overline{\rho}$ and $s\,\overline{\rho}' <_P t'\,\overline{\tau}'$, then $\Phi(t\,\overline{\tau} <_P t'\,\overline{\tau}')$, where Φ is a most general unifier of $\overline{\rho}$ and $\overline{\rho}'$.*

The rationale for these restrictions is that the constraint solving problem is greatly simplified if all derivations of $t\,\overline{\tau} \leq s\,\overline{\rho}$ (with $t \neq s$) can be normalized to contain just one application of rule ($\textsc{const}$) that is not a premise of some ($\textsc{depth}$) step. The requirements on a subtype axiom furthermore outrule recursive axioms, which is vital to the termination property of our constraint solver, as well as axioms that would interfere ambiguously with depth subtyping.

Since the set of valid subtyping judgements depends on a particular subtyping theory P, we let the inference systems in figures 1 and 2 be parametric in P. This P has role analogous to the partial order on types that is assumed in [Hen96]. To see that our definition is an instance of the latter, note that for each P the relation

$$\{\Phi(\tau, \rho) \mid \tau <_P \rho \wedge \Phi \text{ is a substitution}\},$$

contains all relationships derivable by our (CONST) rule, and that its reflexive closure defines a partial order on ground type expressions as required. It is also trivial to verify that our formulation preserves the property that subtyping judgements are closed under substitution.

Our requirements on a subtyping theory are very liberal, and allow both multiple sub- and supertypes (*multiple inheritance* in object-oriented terms), and a form of rank-2 polymorphism that comes from the ability to use some type variables on just one side in a subtype axiom. In concrete programming terms, a subtyping theory expresses both the subtype relationships a language might provide as built-in (e. g. $\mathtt{Int} <_P \mathtt{Real}$, or $\mathtt{Action} <_P \mathtt{Cmd}\,\alpha\ ()$[2]), as well as the relationships that result from incremental type definitions made by the programmer (e. g. $\mathtt{ColorPoint} <_P \mathtt{Point}$, or $\mathtt{Ord}\,\alpha <_P \mathtt{Eq}\,\alpha$). Incremental definitions of records and datatypes will be further described in section 5.

3.2 Properties of typing judgements

We end this section by recapitulating some important results that are proven in general in [Hen96], and thus hold for our system in particular. For this purpose, the subtype relation in figure 2 is extended to type schemes as follows:

1. $C \vdash_P \sigma \leq \tau$ if $C, \{x : \sigma\} \vdash_P x : \tau$.
2. $C \vdash_P \sigma \leq \sigma'$ if for all D and τ such that $D \vdash_P C$ and $D \vdash_P \sigma' \leq \tau$ we have $D \vdash_P \sigma \leq \tau$.

Proposition 1 *Let $\overline{\alpha} \notin fv(C, \Gamma)$. Then $C, \Gamma \vdash_P e : \forall\alpha.\tau|D$ iff $C \cup D, \Gamma \vdash_P e : \tau$.*

Proposition 2 *Let $C \vdash_P \sigma \leq \sigma'$. Then:*

1. *If $C, \Gamma \vdash_P e : \sigma$ then $C, \Gamma \vdash_P e : \sigma'$.*
2. *If $C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma''$ then $C, \Gamma \cup \{x : \sigma\} \vdash_P e : \sigma''$.*

Theorem 1 (Principal types) *Let $fv(e) \subseteq dom(\Gamma)$. Then there exists a σ such that:*

1. *$\emptyset, \Gamma \vdash_P e : \sigma$*[3]
2. *for all σ', if $\emptyset, \Gamma \vdash_P e : \sigma'$ then $\emptyset, \Gamma \vdash_P \sigma \leq \sigma'$.*

Theorem 2 (Subject reduction)
Let $\longrightarrow$ stand for $\beta\eta$let-reduction, and let P be such that $C \vdash_P \tau \to \rho \leq \tau' \to \rho'$ only if $C \vdash_P \tau' \leq \tau$ and $C \vdash_P \rho \leq \rho'$. Then:
If $C, \Gamma \vdash_P e : \sigma$ and $e \longrightarrow e'$ then $C, \Gamma \vdash_P e' : \sigma$.

4 The inference algorithm

We will now turn to the partial inference algorithm we consider our main result. The core of this algorithm is an approximating subtype constraint solver, that has the merit of being simple and efficient – in fact it is defined as a small extension to the very efficient unification algorithm of Martelli and Montanari [MM82]. The main characteristic of our solver is that it approximates constraints of the form $\alpha \leq \beta$ as equality constraints, and resorts to unification in these cases. When all such constraints are removed from a constraint set, computation of least upper bounds or greatest lower bounds for the remaining variables becomes straightforward, and the algorithm can continue (just like in ordinary unification) by solving any constraints induced by the arguments of the bounding type expressions.

This strategy is not very refined, however, and is bound to fail in certain situations. Consider the constraint set

$$\{\alpha \leq \mathtt{Int}, \alpha \leq \beta, \mathtt{Real} \leq \beta\}$$

With our strategy, α would be unified with β, resulting in the unsolvable constraint set $\{\alpha \leq \mathtt{Int}, \mathtt{Real} \leq \alpha\}$. In this case, unifying α with $\mathtt{Int}$ would have been a better way to proceed.

So, if we want to stick to our simple strategy (which we really do, considering its attractively close relationship to well-known unification techniques), it becomes vital to feed as small constraint sets as possible into the constraint solver, in order to minimize the "damage" that variable/variable constraints can give rise to. For example, if the constraint set above actually was generated as the union of the sets $\{\alpha \leq \mathtt{Int}\}$ and $\{\alpha \leq \beta, \mathtt{Real} \leq \beta\}$, solving the first set separately would lead to success even with our simple strategy.

This requirement actually puts us closer to the standard inference algorithm W [Mil78] than what is customary in the literature on subtype inference. As we have indicated, we will have good reasons for solving constraints as soon as they are generated, rather than propagating them upwards in the syntax tree as input to some final simplification/solving pass. The complication we run into is of course that constraints involving variables free in the assumption environment cannot be solved immediately if we want the output of the algorithm to be predictable. However, we will postpone the discussion on how we address this problem until we have presented the constraint solver.

4.1 Solving constraints

The definition of the constraint solver is given in figure 3. It is presented as an inference system for judgements on the form $\Phi \models_P C$, which should have the operational interpretation "given a constraint set C and subtyping theory P, return substitution Φ". If a constraint set does not match any of the five clauses, the result of the algorithm is considered to be `failure`.

This algorithm, as well as the subsequent inference algorithm, depends crucially on the ability to generate fresh type variables. Instead of burdening the presentation with unessential details concerning name supplies, we use the following convention: the symbol ν always represents a type variable that is distinct from any other variable in the derivation or its context, except for other occurrences of ν in the same rule. Likewise, we let $\overline{\nu}$ represent a vector of zero or more unique variables, equal only to the variables denoted by other occurrences of $\overline{\nu}$ in the same rule.

Clauses (A) to (D) in figure 3 essentially constitute the unification algorithm of Martelli and Montanari, or more precisely, their nondeterministic *specification* of the algorithm. We prefer to use this abstract formulation here because of its conciseness; the reader is referred to [MM82] for concrete information on how to make an efficient implementation, especially on how to avoid the costly occurs-check in clause (D).

The computation of least upper bounds / greatest lower bounds in clause (D) is of course added by us; the reader should note, though, that our formulation degenerates to

[2] The latter axiom expresses that a value of type `Action` can be promoted to the monad of commands that execute in a local state α and return values of type (). See [NC97] for more information on this monad.

[3] Note that this statement does not say anything about the *satisfiability* of the constraints that are generally contained in σ.

$$\frac{}{Id \models_P \emptyset} \text{ (A)}$$

$$\frac{\Phi \models_P [\beta/\alpha]C}{\Phi \circ [\beta/\alpha] \models_P C \uplus \{\alpha \leq \beta\}} \text{ (B)}$$

$$\frac{\Phi \models_P C \cup \{\tau_i \leq \rho_i\}^{i \in t^+} \cup \{\rho_j \leq \tau_j\}^{j \in t^-}}{\Phi \models_P C \uplus \{t\,\tau_1 \ldots \tau_{n_t} \leq t\,\rho_1 \ldots \rho_{n_t}\}} \text{ (C)}$$

$$\frac{\alpha \notin fv(C, \overline{\tau}_i, \overline{\rho}_j) \quad t = (\sqcup_P \overline{t_i}) \,[\!]\, (\sqcap_P \overline{s_j}) \quad \Phi \models_P C \cup \{t_i\,\overline{\tau}_i \leq t\,\overline{\nu}\}^i \cup \{t\,\overline{\nu} \leq s_j\,\overline{\rho}_j\}^j}{\Phi \circ [t\,\overline{\nu}/\alpha] \models_P C \uplus \{t_i\,\overline{\tau}_i \leq \alpha\}^i \cup \{\alpha \leq s_j\,\overline{\rho}_j\}^j} \text{ (D)}$$

$$\frac{t\,\overline{\tau}' <_P s\,\overline{\rho}' \quad \overline{\alpha} = fv(\overline{\tau}', \overline{\rho}') \quad \Phi \models_P C \cup \{t\,\overline{\tau} \leq [\overline{\nu}/\overline{\alpha}](t\,\overline{\tau}')\} \cup \{[\overline{\nu}/\overline{\alpha}](s\,\overline{\rho}') \leq s\,\overline{\rho}\}}{\Phi \models_P C \uplus \{t\,\overline{\tau} \leq s\,\overline{\rho}\}} \text{ (E)}$$

Figure 3: The constraint solver

the original one in case the trivial subtyping theory $P = \emptyset$ is given. These standard algorithms use a partial order on pure type constants derived from P, which relates t and s iff $t = s$ or $t\,\overline{\tau} <_P s\,\overline{\rho}$ for some $\overline{\tau}$ and $\overline{\rho}$. Since neither least upper bounds nor greatest lower bounds are guaranteed to exist (for one thing, $\overline{\tau}$ or $\overline{\rho}$ may be empty), both algorithms might return `failure`.

The "fatbar" operator in $(\sqcup_P \overline{t_i}) \,[\!]\, (\sqcap_P \overline{s_j})$ picks one of its arguments in a failure-avoiding manner. In case both arguments evaluate successfully this choice is assumed to be guided by the inference algorithm, so that the alternative is taken which results in the smallest inferred type. If neither alternative is better than the other we somewhat arbitrarily specify that the left argument is selected. Adding an extra parameter to the constraint solver representing the type that should be minimized is straightforward, so we leave out the details in the interest of brevity.

Clause (E) is our main extension to the unification algorithm; it has no correspondence in the original formulation since it handles the case where two distinct type constants are compared. Note that this rule is only applicable if there exists an appropriate subtype axiom in P.

The following lemma states a soundness property of the constraint solver.

Lemma 3 *If $\Phi \models_P C$ then $\emptyset \vdash_P \Phi C$.*

Since the algorithm deliberately discards certain solutions, completeness cannot not hold by definition. However, a distinctive characteristic of our approach is that it is a conservative extension of *unification*. This is made precise below.

Definition 4 *We say that a substitution Φ* unifies *a constraint set C iff $\Phi\tau = \Phi\tau'$ for all $\tau \leq \tau' \in C$.*

Lemma 4 *If Φ unifies C, then $\Phi' \models_P C$ and $\Phi = \Phi'' \circ \Phi'$ for some Φ''.*

PROOF By the same argument as in [MM82], noting that

1. In clause (D), all t_i and s_j must be equal, and since at least one of $\overline{t_i}$ and $\overline{s_j}$ must be non-empty, $(\sqcup_P \overline{t_i}) \,[\!]\, (\sqcap_P \overline{s_j})$ trivially succeeds.
2. Since a subtype axiom never relates a type constant to itself, clause (E) can never match.
3. The case where a type constant is *non-variant* (that is, $i \notin t^+ \cup t^-$ for some $i \leq n_t$) only makes the algorithm output *more* general than the unifying substitution. □

A vital property of the constraint solver is that it terminates on all input, either with a substitution, or with the implicit result `failure` if there is no matching clause. Formally,

Proposition 5 *The relation $\Phi \models_P C$ is decidable.*

PROOF We show that there can be no infinite derivations of $\Phi \models_P C$ by considering the *size* $|C|$ of a constraint set C, as defined by

$$\begin{array}{lcl}
|\{\tau_i \leq \rho_i\}^i| & = & \sum_i |\tau_i \leq \rho_i| \\
|\alpha \leq \tau| & = & |\tau| \\
|\tau \leq \alpha| & = & |\tau| \\
|t\,\overline{\tau_i} \leq t\,\overline{\rho_i}| & = & 1 + 2\sum_i |\tau_i \leq \rho_i| \\
|t\,\overline{\tau} \leq s\,\overline{\rho}| & = & 1 + |t\,\overline{\tau} \leq t\,\overline{\tau}'| + |s\,\overline{\rho}' \leq s\,\overline{\rho}| \\
 & & \text{if } t\,\overline{\tau}' <_P s\,\overline{\rho}' \\
|\overline{\tau_i}| & = & \sum_i |\tau_i| \\
|\alpha| & = & 1 \\
|t\,\overline{\tau_i}| & = & 4 + 2(|\overline{\tau_i}| + n_{\max} + k_t)
\end{array}$$

where $n_{\max}$ is the maximal arity of any t, and k_t is computed for each t by examining all axioms in P of the form $(t\,\overline{\tau} < s\,\overline{\rho})$ or $(s\,\overline{\rho} < t\,\overline{\tau})$, and taking the sum of the sizes of each $\overline{\tau}$ and $\overline{\rho}$ thus encountered. The well-foundedness of this definition follows from the requirement that the type constants related by a subtype axioms occur just once in the axiom (see definition 2).

It is now straightforward to show that every premise in the definition of $\Phi \models_P C$ involves a constraint set of strictly lesser size than the constraint set of the corresponding conclusion. The crucial step is rule (D), where one expands the premise using rules (E) and (C), and then utilizes the fact that for every τ and ρ, $|\tau \leq \rho|$ is less than $|\tau| + |\rho|$. □

4.2 Algorithm definition

The actual inference algorithm is shown in figure 4. Again we use a formulation in terms of an inference system, whose

$$\frac{\overline{\alpha_i} = fv(\sigma) \quad \tau = inst(\sigma)}{\{\nu_i \leq \alpha_i\}^{\alpha_i \in \sigma^-} \cup \{\alpha_i \leq \nu_i\}^{\alpha_i \in \sigma^+}, \Gamma \cup \{x : \sigma\} \models_P x : [\overline{\nu_i}/\overline{\alpha_i}]\tau} \quad (\text{VAR'})$$

$$\frac{C, \Gamma \models_P e : \tau \quad C_i, \Gamma \models_P e_i : \tau_i \quad \Phi \models_P \{\tau \leq \overline{\tau_i} \rightarrow \nu\}}{\Phi(C \cup \bigcup_i C_i), \Gamma \models_P e\, \overline{e_i} : \Phi\nu} \quad (\text{APP'})$$

$$\frac{C, \Gamma \cup \{x_i : \nu_i\}^i \models_P e : \tau \quad \Phi \models_P C \backslash C_\Gamma}{\Phi(C_\Gamma), \Gamma \models_P \lambda\overline{x_i}.e : \Phi(\overline{\nu_i} \rightarrow \tau)} \quad (\text{ABS'})$$

$$\frac{C, \Gamma \models_P e : \tau \quad \sigma = gen(C, \Gamma, \tau) \quad C', \Gamma \cup \{x : \sigma\} \models_P e' : \tau' \quad \Phi \models_P C' \backslash C'_\Gamma}{\Phi(C \cup C'_\Gamma), \Gamma \models_P \texttt{let}\ x = e\ \texttt{in}\, e' : \Phi\tau'} \quad (\text{LET'})$$

Figure 4: The inference algorithm

judgements $C, \Gamma \models_P e : \tau$ should be read "given an assumption environment Γ, a subtyping theory P, and an expression e, return type τ and constraint set C".

The existence of a constraint set in the output from our inference algorithm might at first seem contradictory to our whole approach. However, these constraint sets have a very limited form, and are there just for the same purpose as the substitutions returned by Milner's algorithm W: to propagate requirements on the free variables of the assumption environment Γ [Mil78].

But instead of deciding locally on a fixed substitution that makes Γ meet its requirements, our algorithm will effectively work on cloned copies of Γ (rule (VAR)), and return a constraint set that relates these clones to the original.[4] It is not until a free variable of Γ exits its scope that its constraints are collected and a satisfying substitution is computed (last premise of rules (ABS') and (LET')). A key element in this step is an operation which takes a constraint set and returns only those constraints which reference variables free in Γ:

$$C_\Gamma = \{\tau \leq \tau' \mid (\tau \leq \tau') \in C \wedge fv(\tau, \tau') \cap fv(\Gamma) \neq \emptyset\}$$

We formalize the role of these generated constraint sets as follows:

Definition 5 *A constraint set C is a* Γ-constraint *iff for all $\tau \leq \tau' \in C$, either $\tau \equiv \alpha$ or $\tau' \equiv \alpha$ for some $\alpha \in fv(\Gamma)$.*

Proposition 6 *If $C, \Gamma \models_P e : \tau$ then C is a Γ-constraint.*

PROOF By structural induction on e, using the fact that variables free in Γ never occur in τ. □

Corollary 7 *If $C, \Gamma \models_P e : \tau$ and $fv(\Gamma) = \emptyset$ then $C = \emptyset$.*

Thus, for expressions defined on the top-level of a program, the inference algorithm returns just a type if it succeeds.

Generalization and instantiation play the same role here as in Milner's W. We define these operations for the full type scheme syntax with constraints, even though we will not need this generality until section 7:

$$gen(C, \tau|D) = \forall\overline{\alpha}.\tau|D \text{ where } \overline{\alpha} = fv(\tau, D) \backslash fv(C)$$

$$inst(\forall\overline{\alpha}.\tau|D) = [\overline{\nu}/\overline{\alpha}](\tau|D)$$

Note that gen takes the Γ-constraint C as a parameter instead of Γ, since the free variables of $\tau|D$ and Γ will always be disjoint.

The vector notations in rule (APP') stand for nested application and function type construction, respectively. The possibility of letting more than one argument influence the type of an application expression can have a crucial impact on the result of the algorithm. For example, if $\texttt{f} : \forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$ and the numeric constants have their obvious type, then `f 7 3.14` will be assigned the type `Real`, whereas `f 7` applied to `3.14` will result in a type error (assuming `Int` $\leq$ `Real`). A similar argument, although probably less important in practice, applies to the use of nested abstractions in rule (ABS'). Thus it is implicitly understood that rules (APP') and (ABS') are matched against as large expressions as possible.

It is worth noting that the result of the algorithm is independent of the order in which subexpressions are analyzed (it is only rule (APP') that offers any degree of freedom). A related property guarantees that function arguments can be reordered without causing any other effect than a permutation of the corresponding argument types.

Detailed examples of how the algorithm works can be found in section 6. We end this section with the main technical results about our inference algorithm.

Theorem 3 (Soundness) *If $C, \Gamma \models_P e : \tau$ then $C, \Gamma \vdash_P e : \tau$.*

PROOF By structural induction on e. □

Theorem 4 (Partial completeness) *If $\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau$, then $C, \Gamma \models_P e : \rho$ and there is a Φ' such that Φ' unifies C, $\Phi'\Gamma = \Phi\Gamma$, and $\Phi'\rho = \tau$.*

PROOF By induction on the derivation of $\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau$, utilizing lemma 4 and the fact that if C is a Γ-constraint and Φ unifies C then $fv(\Phi C) \subseteq fv(\Phi\Gamma)$. □

Detailed proofs of the theorems and lemmas stated in this paper will appear in the author's forthcoming thesis.

[4] Here we let the symbols σ^+ and σ^- stand for the free variables of σ that occur in co- and contravariant positions, respectively.

5 Records and datatypes

Subtyping in real programming languages is mostly associated with record-like structures such as classes in object-oriented languages, although it is also perfectly sensible to define subtyping for variant types, or *datatypes* as they are commonly called in functional languages [CW85]. In this section we present an application of our subtyping framework in terms of a system of incrementally definable record types and datatypes, in a style reminiscent of Haskell. Because we have chosen to work with name inequivalence, most of the work that concern extensible type hierarchies is already done in the definition of a valid subtyping theory; therefore the material in this section becomes mostly straightforward.

A distinguishing feature of this extension is that the treatment of records and datatypes is perfectly *symmetric*; that is, there is a close correspondence between record selectors and datatype constructors, between record construction and datatype selection, and between the two forms of type extension, which yields subtypes for records and supertypes for datatypes. Along this line, we will treat both record selectors and datatype constructors as *global* constants – an absolutely normal choice for what datatypes are concerned, but not so for records (see e.g. [MTH90, Gas97]). Still, we think that a symmetric treatment like ours has some interesting merits in itself, and that the ability to form hierarchies of record types alleviates most of the problems of having a common scope for all selector names. We also note that overloaded names in Haskell are given very much the same treatment, without much hassle in practice. Apart from the global scope for selectors, a hierarchy of record types in our system subsumes the essential features of the class hierarchies expressible in Java and C++.

We extend our term language with the following constructs:

$$\begin{array}{lcll} e & ::= & k & \textit{datatype constructor} \\ & | & \{k \to e\}^* & \textit{datatype selection} \\ & | & l & \textit{record selector} \\ & | & \{l = e\}^* & \textit{record construction} \\ & | & \ldots & \end{array}$$

The datatype selection syntax might seem a little unusual, since it does not contain any expression to scrutinize. We have chosen the given formulation in order to emphasize the symmetry between datatype selection and record construction. The syntax $\texttt{case}\ e\ \texttt{of}\ \{k_i \to e_i\}^i$ normally found in functional languages can then be defined as a syntactic sugar for the application $\{k_i \to e_i\}^i\, e$. A similar argument applies to the common dot-notation used for record selection.

We assume that datatypes are declared using the syntax

$$\texttt{data}\, t\,\overline{\alpha} > \{s_i\,\overline{\rho}_i\}^i = \{k_j\ \tau_j\}^j$$

Apart from the optional component $> \{s_i\,\overline{\rho}_i\}^i$, this is essentially ordinary Haskell. Implicit in the definition is the declaration of type schemes for each constructor

$$\sigma_{k_j} = \forall\overline{\alpha}.\tau_j \to t\,\overline{\alpha},$$

and a set of subtype axioms

$$\{s_i\,\overline{\rho}_i < t\,\overline{\alpha}\}^i$$

that the subtyping theory must contain. Since it only makes sense operationally to let datatypes be subtype-related to other datatypes (which are declared analogously), the transitive closure prescribed by condition 2 in definition 3 can always be successfully constructed from any given set of type declarations. However, the result of this operation is not necessarily a valid subtyping theory, which means that a language implementation must also perform a consistency check according to definitions 2 and 3 (first condition) before a new set of type declarations can be accepted.

$$\frac{}{C,\Gamma \vdash l : \sigma_l}\ (\text{SEL})$$

$$\frac{}{C,\Gamma \vdash k : \sigma_k}\ (\text{CON})$$

$$\frac{C,\Gamma \vdash l_i : \tau \to \tau_i \quad \{l_i\}^i = \widehat{\tau} \quad C,\Gamma \vdash e_i : \tau_i}{C,\Gamma \vdash \{l_i = e_i\}^i : \tau}\ (\text{REC})$$

$$\frac{C,\Gamma \vdash k_i : \tau_i \to \tau \quad \{k_i\}^i = \widehat{\tau} \quad C,\Gamma \vdash e_i : \tau_i \to \tau'}{C,\Gamma \vdash \{k_i \to e_i\}^i : \tau \to \tau'}\ (\text{ALT})$$

Figure 5: Records and datatypes

Note that since all constructor names k_j are required to be globally unique, there is no way of *modifying* the type of a constructor when a datatype is extended. Thanks to subsumption, old constructors can nonetheless be used to construct values of the new type.

For records, the declarations look very much the same, except that the new type now denotes a subtype instead of a supertype.

$$\texttt{struct}\, t\,\overline{\alpha} < \{s_i\,\overline{\rho}_i\}^i = \{l_j : \tau_j\}^j$$

Type schemes for the selectors are formed analogously

$$\sigma_{l_j} = \forall\overline{\alpha}.t\,\overline{\alpha} \to \tau_j,$$

as well as a set of subtype axioms:

$$\{t\,\overline{\alpha} < s_i\,\overline{\rho}_i\}^i$$

The argument above regarding closure generation and validity of the resulting subtyping theory applies here as well. And as for datatypes, it is the subsumption rule that will allow us to apply old selectors to values of the new record type.

Variances for user-defined types must also be calculated by a language implementation. Essentially this information follows from the way argument variables are used in constructor/selector types and sub-/supertypes; however, the fact that type declarations may be recursive causes a slight problem. Fortunately, a bit of abstract interpretation over a domain consisting of the four subsets of $\{+,-\}$ will do the job.

The typing rules for records and datatypes are given in figure 5, while figure 6 shows the necessary algorithm extensions. These definitions contain no surprises, the only difference subtyping makes in the typing rules is that we must generally rely on subsumption in order to find a common argument type for the selectors in rule (REC). The same holds for the result type of the constructors in rule (ALT).

Since a type constant contains no information about its own declaration, checking that datatype selection / record

$$\frac{}{\emptyset, \Gamma \models_P l : inst(\sigma_l)} \text{ (SEL')}$$

$$\frac{}{\emptyset, \Gamma \models_P k : inst(\sigma_k)} \text{ (CON')}$$

$$\frac{\tau_i \to \tau_i' = inst(\sigma_{l_i}) \quad \Phi \models_P \{\nu \leq \tau_i\}^i \quad \{l_i\}^i = \widehat{\Phi\nu} \quad C_i, \Gamma \models_P e_i : \rho_i \quad \Phi' \models_P \{\rho_i \leq \Phi\tau_i'\}^i}{\Phi' \bigcup_i C_i, \Gamma \models_P \{l_i = e_i\}^i : \Phi'\Phi\nu} \text{ (REC')}$$

$$\frac{\tau_i' \to \tau_i = inst(\sigma_{k_i}) \quad \Phi \models_P \{\tau_i \leq \nu\}^i \quad \{k_i\}^i = \widehat{\Phi\nu} \quad C_i, \Gamma \models_P e_i : \rho_i \quad \Phi' \models_P \{\rho_i \leq \Phi\tau_i' \to \nu'\}^i}{\Phi' \bigcup_i C_i, \Gamma \models_P \{k_i \to e_i\}^i : \Phi'\Phi(\nu \to \nu')} \text{ (ALT')}$$

Figure 6: Type inference for datatypes and records

construction is exhaustive must be done using an additional attribute associated with each type constant. This attribute is written $\widehat{t}$, and is defined as the set of constructor/selector names directly introduced in the declaration of t, plus all the names associated with the declared sub/super-types of t. We tacitly lift this attribute to a partial function on type expressions in the obvious way.

We have assumed here that both kinds of type declarations only mention variables that appear in the argument list $\overline{\alpha}$. As has been described by Läufer and Jones among others, lifting this restriction for a constructor type naturally leads to a system with support for local existential quantification, while the corresponding generalization for selector types is best interpreted as local universal quantification [Läu92, Jon96, Jon97]. Taking this idea further, by also letting the declared sub-/supertypes contain free variables, opens up some interesting possibilities to explore (recall that the implied subtype axioms remain valid in spite of this change). For one thing, a term can now be assigned an existentially quantified type simply by means of a type annotation, or even just by using the term in the right context. However, although this additional feature is natural and carries no extra implementational cost, it is not clear how useful it really is in practice.

6 More examples

In this section we will discuss the behaviour of our inference algorithm by means of some programming examples, that we have tested on a prototype implementation running under Hugs 1.4. The concrete syntax we use follows the syntax of Haskell. Input to the prototype is an expression written after the prompt >. Output follows after the :: sign, and is either a type or an error message. We will assume an environment where Int $\leq$ Real, and where the following declarations are in scope:

```
struct Box a =
  fst   :: a
struct Couple a < Box a =
  snd   :: a
struct Couple' < Box Int =
  snd'  :: Bool
rdict   :: Ord Real
idict   :: Ord Int
==      :: Int -> Int -> Bool
```

Here `Ord` is assumed to be defined as in section 2.

Some basic examples of how the algorithm works have also been given in section 2. Below follows some examples that need slightly more involved computations, including least upper bounds, greatest lower bounds, and depth subtyping with contravariance (due to type `Ord`).

```
>  a = {fst = 1}
:: Box Int
>  b = {fst = 1; snd = 3.14}
:: Couple Real
>  c = {fst = idict; snd = rdict}
:: Couple (Ord Int)
>  d = {fst = 1; snd' = False}
:: Couple'
>  e = {fst = 3.14; snd' = False}
:: ### Type error
```

These types are all inferred using rule (REC'). We give the derivation used in example `d` as an illustration; the other derivations follow the same pattern.

$$\frac{\begin{array}{c}\mathtt{Box}\,\alpha \to \alpha = inst(\sigma_{\mathtt{fst}}) \quad \mathtt{Couple'} \to \mathtt{Bool} = inst(\sigma_{\mathtt{snd'}}) \\ [\mathtt{Int}/\alpha, \mathtt{Couple'}/\nu] \models_P \{\nu \leq \mathtt{Couple'}, \nu \leq \mathtt{Box}\,\alpha\} \\ \{\mathtt{fst}, \mathtt{snd'}\} = \widehat{\mathtt{Couple'}} \\ \emptyset, \Gamma \models_P 1 : \mathtt{Int} \quad \emptyset, \Gamma \models_P \mathtt{False} : \mathtt{Bool} \\ Id \models_P \{\mathtt{Int} \leq \mathtt{Int}, \mathtt{Bool} \leq \mathtt{Bool}\}\end{array}}{\emptyset, \Gamma \models_P \{\mathtt{fst} = 1, \mathtt{snd'} = \mathtt{False}\} : \mathtt{Couple'}}$$

The first constraint solving problem above is non-trivial, so we write it out as well. It also illustrates most of the clauses that constitute the constraint solver. Some obvious premises are left out for space economy reasons, though, most notably `Couple'` $<_P$ `Box Int` in clause (E), and `Couple'` $=$ `Couple'` $\sqcap_P$ `Box` in the bottom application of clause (D).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Id \models_P \emptyset}\text{ (A)}}{[\mathtt{Int}/\alpha] \models_P \{\mathtt{Int} \leq \alpha\}}\text{ (D)}}{[\mathtt{Int}/\alpha] \models_P \{\mathtt{Box\,Int} \leq \mathtt{Box}\,\alpha\}}\text{ (C)}}{[\mathtt{Int}/\alpha] \models_P \{\mathtt{Couple'} \leq \mathtt{Couple'}, \mathtt{Box\,Int} \leq \mathtt{Box}\,\alpha\}}\text{ (C)}}{[\mathtt{Int}/\alpha] \models_P \{\mathtt{Couple'} \leq \mathtt{Box}\,\alpha\}}\text{ (E)}}{[\mathtt{Int}/\alpha] \models_P \{\mathtt{Couple'} \leq \mathtt{Couple'}, \mathtt{Couple'} \leq \mathtt{Box}\,\alpha\}}\text{ (C)}}{[\mathtt{Int}/\alpha, \mathtt{Couple'}/\nu] \models_P \{\nu \leq \mathtt{Couple'}, \nu \leq \mathtt{Box}\,\alpha\}}\text{ (D)}$$

The next two examples illustrate the point of collecting constraints on the assumption environment, instead of solving them directly:

```
> f x = (x.fst, x.snd)
:: Couple a -> (a,a)
> g x = (x.fst, x.snd')
:: Couple' -> (Int,Bool)
```

Part of the derivation for the lambda-abstraction g is shown below, where (incidentally) C is identical to the constraint set solved in detail in the previous example.

$$\frac{C, \{\mathtt{x} : \nu\} \models_P (\mathtt{x.fst}, \mathtt{x.snd'}) : (\alpha, \mathtt{Bool}) \quad \Phi \models_P C}{\emptyset, \emptyset \models_P \lambda\mathtt{x}.(\mathtt{x.fst}, \mathtt{x.snd'}) : \mathtt{Couple'} \to (\mathtt{Int}, \mathtt{Bool})}\ (\text{ABS'})$$

Had the algorithm not accumulated any constraints on ν but instead attempted to solve these immediately, the order in which subexpressions are visited would make a difference, and the naive order would have failed at the subderivation

$$\_, \{x : \mathtt{Box}\,\alpha\} \models_P \mathtt{x.snd'} : \_$$

On the other hand, the following example would not have been accepted if the algorithm just naively collected *all* constraints and solved them on the top-level:

```
> h x = if x==0 then 3.14 else x
:: Int -> Real
```

If the type variable assigned to x is ν, our algorithm will analyze the right-hand side of h essentially as follows:[5]

$$\frac{\begin{array}{c}\{\nu \leq \mathtt{Int}\}, \Gamma \models_P \mathtt{x==0} : \mathtt{Bool} \\ \emptyset, \Gamma \models_P \mathtt{3.14} : \mathtt{Real} \quad \{\nu \leq \nu''\}, \Gamma \models_P \mathtt{x} : \nu'' \\ [\mathtt{Real}/\nu', \mathtt{Real}/\nu''] \models_P \{\mathtt{Real} \leq \nu', \nu'' \leq \nu'\}\end{array}}{\{\nu \leq \mathtt{Int}, \nu \leq \mathtt{Real}\}, \Gamma \models_P \mathtt{if\,x==0\,then\,3.14\,else\,x} : \mathtt{Real}}$$

Notice here that if the algorithm instead just had returned $\{\nu \leq \mathtt{Int}, \nu \leq \nu'', \mathtt{Real} \leq \nu', \nu'' \leq \nu'\}$, our simple constraint solver would be bound to fail when applied to this constraint set at a later stage, for reasons discussed in the beginning of section 4.

Recall that the function min from section 2 was assigned a rather limited type by our inference algorithm. Interestingly, min can be given an alternative coding, inspired by overloading in Haskell, that avoids losing information:

```
> min d x y = if d.less x y then x else y
:: Ord a -> a -> a -> a
> i = min idict 1 2
:: Int
> j = min rdict 1 2
:: Int
> k = min rdict 1 3.14
:: Real
```

The instantiation variable that corresponds to a above will have both upper and lower bounds in these examples. In example j the lower bound Int is chosen in favour of the upper bound Real, on the grounds that it makes the resulting type more precise (see section 4.1).

The effect of narrowing the context of an application can sometimes be puzzling:

```
> l = min rdict 1
:: Int -> Int
> k = min rdict
:: Real -> Real -> Real
```

Still, the algorithm just consequently uses every piece of local information it has – if there are no lower bounds on a variable it will use the upper bounds. In the next section we will see how the behaviour of the algorithm can be improved by letting it take some contextual information into account as well.

7 Type checking

Since we are working with an incomplete inference algorithm, the standard method of checking type annotations *after* a principal type has been inferred will not work. As a remedy, we develop an alternative approach to type annotations in this section, that makes type checking an integral part of the inference algorithm.

First we extend the expression syntax to include terms annotated with a signature

$$e \quad ::= \quad \ldots \mid e :: \sigma$$

and add a corresponding rule to the type system

$$\frac{C, \Gamma \vdash_P e : \sigma}{C, \Gamma \vdash_P (e :: \sigma) : \sigma}\ (\text{TYP})$$

Since a signature is a type scheme, it can contain subtype constraints as well as explicit quantifiers, even though it is arguably more convenient to let the latter be implied in the concrete syntax, as is done for example in Haskell. We will actually require here that all variables appearing in a signature be bound in the same annotation, although extending the system to deal with type variables of nested scope should present no specific problems.

The main change we introduce compared to the previous sections is that the type of an expression can now also be determined by the type expected by the context in which the expression appears. In principle such contextual information has its origin in an explicit type annotation, but we also want to make sure that expected types are propagated down to subexpressions as far as possible. For an application $e\,e'$ this requirement means that if the result type is expected to be τ, e should have the type $\tau' \to \tau$, where τ' is the type *inferred* for e'.[6]

A consequence of propagating information this way is that an expected type will generally contain a mixture of universally quantified variables (which must be protected from instantiation and in effect be treated as unique constants), and free instantiation variables originating from inferred types. This complicates the type checking algorithm slightly, but once it is handled properly, two other benefits come almost for free.

Firstly, the type checker and the inference algorithm can now be integrated, considering type inference as a special

[5] The rule used here can easily be derived if the syntax sugar if e_1 then e_2 else e_3 is expanded into $\{\mathtt{True} \to \lambda() \to e_2, \mathtt{False} \to \lambda() \to e_3\}\ e_1$.

[6] Alternatively, we could have let the contextual information flow in the other direction and extract the expected type of e' from the type inferred for e, as is done in [PT98]. But this scheme would not make any good use of the parameter τ, and since we most often do not need to help the inference algorithm finding types for the anonymous lambda abstractions that might appear in e' (which is the motivation behind the choice in [PT98]), it makes more sense to collect as much information as possible before instantiating the generally polymorphic type of e.

case of checking where the expected type is a unique instantiation variable. Secondly, it becomes possible to let the programmer exploit the use of *partial* type signatures, i. e. signatures where the type component may contain "holes" in the form of wildcards (_). Such signatures may for example come in handy in situations where the programmer needs to specify that the type of an expression should be an application of a specific constant, but where the inference algorithm may be trusted with inferring the type arguments. We will not develop this idea any further here; we only want to emphasize that partial signatures is a natural generalization that our algorithm directly supports, provided that all wildcard types are replaced with unique instantiation variables prior to type checking.

The extended inference algorithm is shown in the form of an inference system in figure 7. Judgements in this system are of the form $C, D, \Gamma \models_P e : \Phi(\tau)$ and should be read "given an assumption environment Γ, a subtyping theory P, an expression e, and an expected type τ, return constraint sets C and D, and substitution Φ". The intuition behind this judgement form is captured by the soundness theorem for the extended algorithm, which states that under certain reasonable restrictions on the variables free in τ, if $C, D, \Gamma \models_P e : \Phi(\tau)$ then $C \cup D, \Gamma \vdash_P e : \Phi\tau$.

The rationale for returning two constraint sets is that we want to separate those constraints that restrict variables free in Γ (the Γ-constraint C) from those that directly or indirectly originate from type signatures supplied by the programmer (the set D). The latter set is a natural part of the type schemes generated in rule (LET"). Note that by letting D contain instantiated constraints from rule (VAR"), as well as constraints that appear directly in a signature (rule (TYP")), we obtain an algorithm with the intuitive property that both x and y receive the same type scheme in $\texttt{let}\, x = e :: \sigma \,\texttt{in}\,\texttt{let}\, y = x \,\texttt{in}\, e'$.

In rule (LET") it is assumed that there exists a constraint simplification algorithm *simp* that is applied to D before a type scheme is generated. This algorithm can of course be arbitrarily sophisticated; however, since the constraints in its domain are either explicitly given in the program text, or instantiations of such constraints, it seems like the only really necessary requirements on *simp* are that it removes tautological constraints like $\texttt{Int} \leq \texttt{Real}$, and flags an error if a constraint is obviously inconsistent.

Variables which are universally quantified in an enclosing derivation are assumed to be *skolemized*, i. e. replaced with unique type constants of arity 0, whenever they appear free in a premise (see the first two premises in rule (TYP")). For this purpose we assume that the set of type constants contains a set of *skolem* constants that is sufficiently large for the program in question, and that $fs(C)$ returns the set of skolem constants that occur in C. Furthermore, we apply the same notational technique as we do for indicating unique type variables, and assume that $\overline{\mu}$ stands for a vector of zero or more skolem constants that are equal only to the constants denoted by other occurrences of $\overline{\mu}$ in the same rule.

The annotated expression e in rule (TYP") is checked according to an extended subtyping theory $P \oplus [\overline{\mu}/\overline{\alpha}]D'$, where $\oplus$ denotes the operator implicitly referenced in section 5 that adds axioms to a subtyping theory and validates the generated closure. Note that all axioms in $[\overline{\mu}/\overline{\alpha}]D'$ must be monomorphic, since by assumption $fv(D') \subseteq \overline{\alpha}$. The extended subtyping theory is also used to solve all direct and indirect signature constraints encountered while checking e, in order to establish that they are all implied by D'. When it has been verified that no universally quantified variable escapes its scope, a fresh instance of the signature is generated, and a final check is made to ensure that the annotated type fits below the type expected by the context.

The parts of the type checker that deal with records and datatypes are straightforward and will not be shown. Type checking does nonetheless appear to be especially useful for these constructs in practice, since type signatures for all selectors and constructors are already given in their respective type declarations.

Finally it should be noted that this integrated type-checking/type-inference algorithm is still incomplete, in the sense that adding a top-level type annotation is not necessarily sufficient to make a typeable program accepted by the algorithm. Seeing this is easy: the type checker relies on type inference for subexpressions in several places, and we know that the inference algorithm is incomplete.

An (almost) complete type checker could probably be developed for our system by following the ideas in [TS96, MW97], and would be valuable for the same reason that a complete inference algorithm is. However, since the former kind of algorithm by necessity must be based on the latter, problems regarding unreadable diagnostic messages would reappear, and the programmer would experience significantly different responses from the system depending on whether a term has a type annotation or not.

Despite its less ambitious goal, our approach to type checking fulfills a vital need in conjunction with our inference strategy, in that it enables the programmer to guide the inference algorithm at points where it would otherwise have made a bad choice. Indeed, identifying these points on basis of error messages is likely to be facilitated by the very same properties that contribute to making the algorithm incomplete: contextual information in form of expected types is explicitly propagated top-down, and readable types are assigned not only to top-level terms, but to every subexpression as well.

8 Related work

Most of the work on subtype inference cited in the introduction take a *structural* view of the subtype relation, that is, types are subtype-related if and only if they have the same structure (a function, a pair, a list, etc) and their respective constituent parts are subtype-related [Mit84, FM90, FM89, Mit91, Kae92, Smi94, AW93, MW97]. This leads to a subtyping theory where the only primitive subtype relationships are between nullary base types. The system described in [EST95] follows [CW85] and allows two records to be subtype related if their set of selectors are in a *superset* relationship. Still, it is the structure of a record that defines its position in the subtype hierarchy, not any intentions explicitly declared by the programmer. Henglein's type system [Hen96] is unique in that it is *generic* in the choice of subtyping theory, something which we have exploited in our system based on *name* inequivalence.

Despite the fact that type systems with name-based (in)equivalence dominate among both functional and object-oriented languages currently in widespread use, polymorphic subtype inference for such systems have not gained much attention. Reppy and Riecke develop a scheme where object types are related by declaration in their object-oriented ML variant OML [RR96], while Jategonkar and Mitchell outline a similar (future) system for abstract datatypes in [JM93]. Both these systems consider only parameterless type constants. Depth subtyping and variances appear in Freeman's

$$\frac{\overline{\alpha_i} = fv(\sigma) \quad \tau'|D = inst(\sigma) \quad \Phi \models_P \{[\overline{\nu_i}/\overline{\alpha_i}]\tau' \leq \tau\}}{\{\Phi\nu_i \leq \alpha_i\}^{\alpha_i \in \sigma^-} \cup \{\alpha_i \leq \Phi\nu_i\}^{\alpha_i \in \sigma^+}, \Phi[\overline{\nu_i}/\overline{\alpha_i}]D, \Gamma \cup \{x : \sigma\} \models_P x : \Phi(\tau)} \text{ (VAR”)}$$

$$\frac{C_i, D_i, \Gamma \models_P e_i : \Phi_i(\nu_i) \quad C, D, \Gamma \models_P e : \Phi(\overline{\Phi_i\nu_i} \to \tau)}{\bigcup_i \Phi C_i \cup C, \bigcup_i \Phi D_i \cup D, \Gamma \models_P e\,\overline{e_i} : \Phi(\tau)} \text{ (APP”)}$$

$$\frac{\Phi \models_P \{\overline{\nu_i} \to \nu \leq \tau\} \quad C, D, \Gamma \cup \{x_i : \Phi\nu_i\}^i \models_P e : \Phi'(\Phi\nu) \quad \Phi'' \models_P C \backslash C_\Gamma}{\Phi'' C_\Gamma, \Phi'' D, \Gamma \models_P \lambda\overline{x_i}.e : \Phi''\Phi'\Phi(\tau)} \text{ (ABS”)}$$

$$\frac{C, D, \Gamma \models_P e : \Phi(\nu) \quad \sigma = gen(C, \Phi\nu | simp(D)) \quad C', D', \Gamma \cup \{x : \sigma\} \models_P e' : \Phi'(\tau) \quad \Phi'' \models_P C' \backslash C'_\Gamma}{\Phi''(C \cup C'_\Gamma), \Phi'' D', \Gamma \models_P \mathtt{let}\, x = e \,\mathtt{in}\, e' : \Phi''\Phi'(\tau)} \text{ (LET”)}$$

$$\frac{C, D, \Gamma \models_{P \oplus [\overline{\mu}/\overline{\alpha}]D'} e : \Phi([\overline{\mu}/\overline{\alpha}]\tau') \quad \Phi' \models_{P \oplus [\overline{\mu}/\overline{\alpha}]D'} D \quad \overline{\mu} \notin fs(\Phi' C) \quad \Phi'' \models_P \{\Phi'\Phi[\overline{\nu}/\overline{\alpha}]\tau' \leq \tau\}}{\Phi''\Phi' C, \Phi''[\overline{\nu}/\overline{\alpha}]D', \Gamma \models_P (e :: \forall\overline{\alpha}.\tau'|D') : \Phi''(\tau)} \text{ (TYP”)}$$

Figure 7: The integrated type-checking/type-inference algorithm

work on refinement types [Fre94], although in a quite different setting where the subtyping system is used to provide additional information on terms that already have an ML type. We are not aware of any system besides ours that provides a combination of subtype declarations resembling our polymorphic subtype axioms, and variance-based depth subtyping.

The choice to settle for a deliberately incomplete inference algorithm sets our work apart from most of the systems above. Aiken and Wimmers' constraint solver makes some non-conservative simplifications in the interest of efficiency, although for a very rich type language including union and intersection types [AW93]. Reppy and Riecke have implemented type inference for OML such that only constraint-free type schemes are inferred [RR96], but they do not describe their algorithm, nor do they provide any technical results.

Most related to our approach is perhaps Pierce and Turner's work on local type inference [PT98]. They start with a variant of System F with subtyping, and develop an inference technique that is able to fill in missing type arguments, as well as missing annotations, in many cases. Their method for inferring types for anonymous abstractions is similar to our implementation of integrated type-checking, although their algorithm switches between strict phases of inference and checking, which ours does not. Pierce and Turner's algorithm is not complete w. r. t. the Hindley/Milner system, but they have the advantage of a declarative specification of the programs that the inference algorithm accepts.

Cardelli's implementation of $F_\leq$ [Car93] also contains a partial type inference algorithm that, like ours, uses unification to solve constraints involving unbound type variables. However, this “greedy” algorithm solves *all* constraints, including those on variables in the assumption environment, at the earliest stage possible, thus its behaviour can sometimes appear quite unpredictable.

Objective ML [RV97] conservatively extends ML with subtype inference and a powerful type abbreviation mechanism that achieves much of the succinctness we obtain by using name inequivalence. The type system of Objective ML is based on extensible records, though, and does not support subsumption.

Many of the systems mentioned allow subtype constraints to be recursive [Kae92, EST95, AW93, RV97, MW97], thus facilitating certain object-oriented programming idioms not otherwise expressible [BCC+96]. Our system does not currently support recursively constrained types, although a generalization that would allow such constraints to be *checked* (just as ordinary constraints can be checked but not automatically inferred by our system) would be a worthwhile future extension.

9 Conclusion and future work

We have described a subtyping extension to the Hindley/Milner type system, and an accompanying partial type inference algorithm which (1) finds principal types for all expressions that do not use subtyping, (2) finds succinct types for most expressions that use subtyping but do not need constraints, and (3) fails for all expressions that must be typed using constraints. We have also shown how a subtype relation based on name inequivalence can be used to implement a system of incrementally definable datatypes and record types, that is close in spirit to both datatypes of languages like ML and Haskell, and interface hierarchies in popular object-oriented languages like Java.

The inference algorithm has been implemented as a prototype interpreter. Initial experiments suggest that the algorithm works very well in practice, and that explicit type annotations are rarely needed. Currently, a modification of Hugs is under development, that would allow practical experimentation on a larger scale.

Future directions for this work include finding better characterizations of which expressions the algorithm accepts. One possible approach to this could be to identify a set of restrictions on terms and typing derivations that would yield minimal types in the presence of subtyping, and show that under these restrictions, the algorithm is complete. There are also some interesting extensions that we would like to explore, including type-checking recursive constraints, implementing higher-order polymorphism as in [Jon93], and investigating ways to encode and eventually integrate the overloading system of Haskell.

Acknowledgement The author would like to thank Thomas Hallgren, Magnus Carlsson, Lars Pareto, Björn von Sydow, Fritz Henglein, and the ICFP referees for comments, encouragement, and instructive feedback.

References

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[Car93] Luca Cardelli. An implementation of $F_{\leq}$. Technical Report Research report 97, DEC Systems Research Center, February 1993.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), 1985.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In *OOPSLA '95*. ACM, 1995.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Theory and Practice of Software Development*, Barcelona, Spain, March 1989. Springer Verlag.

[FM90] Y. Fuh and P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73, 1990.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, March 1994.

[Gas97] Benedict R. Gaster. Polymorphic Extensible Records for Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[Hen96] Fritz Henglein. Syntactic properties of polymorphic subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, May 1996.

[HM95] M. Hoang and J. Mitchell. Lower Bounds on Type Inference With Subtypes. In *ACM Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press.

[JM93] L. Jategaonkar and J.C. Mitchell. Type inference with extended pattern matching and subtypes. *Fund. Informaticae*, 19:127–166, 1993.

[Jon93] M.P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[Jon96] M.P. Jones. Using Parameterized Signatures to Express Modular Structure. In *ACM Principles of Programming Languages*, St Petersburg, FL, January 1996. ACM Press.

[Jon97] M.P. Jones. First-Class Polymorphism with Type Inference. In *ACM Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[Kae92] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *ACM Lisp and Functional Programming*, pages 193–204, San Francisco, CA, June 1992.

[Läu92] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, July 1992.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mit84] J. Mitchell. Coercion and type inference. In *ACM Principles of Programming Languages*, 1984.

[Mit91] J. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), 1982.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP*, pages 136–149, Amsterdam,Holland, June 1997.

[NC97] Johan Nordlander and Magnus Carlsson. Reactive Objects in a Functional Language – An escape from the evil "I". In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[Reh97] J. Rehof. Minimal typings in atomic subtyping. In *ACM Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[RR96] J. Reppy and J. Riecke. Simple objects for Standard ML. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *ACM Principles of Programming Languages*, Paris, France, January 1997.

[Smi94] G. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, (23):197–226, 1994.

[TS96] Valery Trifonov and Scott Smith. Subtyping Constrained Types. In *Third International Static Analysis Symposum*, LNCS 1145, Aachen, Germany, September 1996.