Type Inference with Deferral

Michał Moskal

Institute of Computer Science, University of Wrocław
michal.moskal@nemerle.org,
WWW home page: http://www.ii.uni.wroc.pl/~mjm/

Abstract. Type inference in polymorphic, nominal type systems with subtyping, additionally equipped with *ad-hoc* overloading is not easy. However most mainstream languages like C#, Java and C++ have all those features, which makes extending them with type inference cumbersome. We present a practical, sound, but not complete, type inference algorithm for such type systems. It is based on the on–line constraint solving combined with deferral of certain typing actions. The algorithm is successfully employed in functional and object–oriented language for the .NET platform called Nemerle.

1 Introduction

Type inference is the task of statically reconstructing type annotations for a program without them. While it is used in the languages of the ML family since the very beginning, the more mainstream languages were rather lazy to adopt it. We believe this is partially due to the problems with combining type inference with nominal type systems with subtyping.

During the rest of this section we will try to depict this problem in more detail and present the key idea of our solution. The rest of the paper is dedicated to formal treatment of the solution.

1.1 Motivation

There are several problems with type inference and mainstream languages – proper error reporting, dealing with nominal types, ambiguous member access, subtyping and static overloading to name a few.

We faced them during our work on the Nemerle[1] programming language. It is a functional and object–oriented language for the .NET platform. It was thought as a transition language for people with C# (or similar) background, to convince them to functional programming. It provides easy access to the common attributes of functional languages (functional values, pattern matching, data types and so on), but still shares most of the syntax with C#. It was quite important to also share the semantics and object model, where possible.

[1] http://nemerle.org/

We therefore had to face a difficult problem of resolving member access in an environment with static overloading, nominal type system and subtyping. For example consider the following type judgment using overloaded member access operator (“`.`”)[2]:

$$(\lambda x.\ x.\texttt{foo}) : \forall \alpha, \beta.\, \alpha \to \beta \textbf{ where } \alpha <: \{\texttt{foo} : \beta\}$$

where $\{\texttt{foo} : \beta\}$ stands for type of objects with a field named `foo` of type β and $<:$ stands for subtyping relation. The typing is both correct and quite precise.

However the obvious problem here is that the type $\{\texttt{foo} : \beta\}$ is not nominal, for example there are no such types in .NET nor Java runtimes.

So what can we do to get the nominal type for this expression? First idea would be to look up classes with a field `foo`, and if there is only one such class, say τ, where the field has type σ, then we can give the above expression type $\tau \to \sigma$. This is what OCaml and O'Haskell do for record types – both require record fields to be unique.

The most important problem with this approach is the case where there are two or more classes with the field `foo`. Types get complicated, and moreover, in the generated code only one nominal type can be assigned to x due to runtime environment limits.

There is another idea – to leave the correct structural type, as the one above, until after inference is completed for a given compilation unit. That is to defer the nominal type selection until more information is collected. This is something along the lines of what SML does for records.

It is still problematic though – working with unconcreteised types is hard, when one wants features designed without inference in mind like static overloading or implicit conversions. The other problem is error messages – it is far better to complain about types the user knows something about, like `list[int]` and not about some structural monsters that can get a few pages long. Moreover we loose link between source code location and the error (constraint solver may fail in another constraint, resulting from some other source location).

Because of the problems above Nemerle did not have a proper inference rules for member access – it was always required to annotate parameters that were going to be used for member access like this: $\lambda x : \texttt{Bar}.\ x.\texttt{foo}$. This was problematic in the practical usage of the language, so while the algorithm we present is still not complete, it solves this problem which seems to be a step in the right direction.

We use a subtyping constraint solver combined with deferring parts of type inference to get precise nominal types, which is a novel approach to solve the ambiguous member access problem.

---

[2] This is supposed to mean a function extracting a field named `foo` from an object passed to it.

1.2 The Idea

This technique of deferring typing comes from a trivial observation that most functions are finally used somewhere. Because we infer parameter types only for local functions, the use sites can be easily inspected. As a result a proper type for the formal parameter can be often inferred. For example consider[3]:

$$\textbf{let } invoke = \lambda x.\, x.\text{some\_method\_of\_foo}() \textbf{ in}$$
$$\textbf{let } a\_list = [\text{make\_foo}(1), \text{make\_foo}(2), \text{make\_foo}(3)] \textbf{ in}$$
$$\text{List.Map}(a\_list, invoke)$$

The proper type of the local function *invoke* cannot be inferred until List.Map *(...)* is typed and the restriction is placed on the type of the *invoke* single argument. The List.Map function has the type $\forall\alpha, \beta.\, list[\alpha] \times (\alpha \to \beta) \to list[\beta]$, which binds the result type of make_foo, that is Foo, with the type of the single *invoke* argument (through the α type variable). After this is done, we can easily type member access on x inside the *invoke* function.

Note that various techniques of the local type inference [17] cannot type the above expression. They can (in most cases) handle a case when the lambda expression is used directly:

$$\text{List.Map}(a\_list, \lambda x.\, x.\text{some\_method\_of\_foo}())$$

We have however found this not to be good enough – it should be always possible to name and reuse local functions without additional hassle for software engineering reasons. In the example above the *invoke* function can be reused with some other list of objects of type *Foo* (though not some with other, unrelated objects with some_method_of_foo method). We have also found the locality of type variables not to work very well with polymorphic, mutable collections.

So the basic idea is to type the expression as usual, but in case where normally an ambiguity error would have been reported we just put this expression on the queue of expressions to be typed later. This requires certain open–mindness in treatment of type variables – we cannot generalize them too early, they have to be open to get new constraints, that can result from typing expressions from the "later–queue".

We use a subtyping constraint solver. We however require it to fulfill certain "on–line" requirements – we have to be able to add constraints on the fly, check if the solver is still consistent and query the upper and lower bounds on types that can be assigned to a given variable in the final solution. These bounds are used to type member access.

The rest of the paper is organized as follows: Sect. 2 describes the source language we are going to use, Sect. 3 describes the type inference algorithm, states two theorems about its soundness and gives a sketch of the constraint solver implementation, next Sect. 3.6 and 4 gives some insights about how to make the algorithm deterministic and how it performs in the Real World scenarios. Finally we present related and future work.

[3] Foo is a class type here, make_foo constructs new objects of this type and some_method_of_foo is clearly some method in the Foo class.

2 The Language

There are two aspects of the language, we are interested in. First we have a description of an environment in which a program is interpreted. This corresponds to all global definitions within a program. Second we have an expression to be typed. This corresponds to a single method's body.

The *environment* E is a set of classes. A *class* is a term of the following form:

$$\texttt{class}\ k(\alpha_1, \ldots, \alpha_n)\ :\ \tau_1, \ldots, \tau_l\ \{m_1 : \sigma_1, \ldots, m_q : \sigma_q\}$$

where $n, l, q \geq 0$, α_i range over infinite denumerable set $\mathcal{V}$ of *type variables*, m_i range over infinite denumerable set $\mathcal{M}$ of *member identifiers*, k range over infinite denumerable set $\mathcal{K}$ of *type constructors* and finally τ_i and σ_i range over the set $\mathcal{T}$ of *types* defined as follows:

$$\tau ::= \alpha \mid k(\tau_1, \ldots, \tau_n) \mid (\tau_1, \ldots, \tau_n) \to \tau_0$$

where $n \geq 0$, and additionally for a parametric type $k(\tau_1, \ldots, \tau_n)$ to be valid, there has to be a $\texttt{class}\ k(\alpha_1, \ldots, \alpha_n) \ldots \in E$.

An example class definition can look like this:

$$\begin{array}{l}\texttt{class}\ Set(\alpha)\ :\ IComparable(Set(\alpha)), IEnumerable(\alpha)\ \{\\ \quad add\ :\ (\alpha) \to Set(\alpha)\\ \quad add\ :\ (\alpha, \alpha) \to Set(\alpha)\\ \quad compare\_to\ :\ (Set(\alpha)) \to Int32()\\ \quad get\_enumerator\ :\ () \to IEnumerator(\alpha)\\ \}\end{array}$$

Please note that the *add* member is defined twice inside Set, with different types. We allow such overloading.

We define $FV : \mathcal{T} \to 2^{\mathcal{V}}$ to be a *free variable set* of a given type. It is simply set of all the type variables used in it.

A *substitution* is a function $\eta : \mathcal{V} \to \mathcal{T}$. We overload the symbol η for the above function and its homomorphic extension to $\eta : \mathcal{T} \to \mathcal{T}$. We use notion $[\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]$ (or sometimes $[\overline{\alpha} := \overline{\tau}]$) for a substitution replacing α_i with τ_i for $i = 1 \ldots n$, and being identity elsewhere. A substitution η is *ground* iff $\forall \alpha.\ FV(\eta(\alpha)) = \emptyset$.

2.1 Subtyping

To define what is subtyping we first need to place additional restrictions on E:

1. for any E and k there exists at most one class named k in E,
2. given a definition $\texttt{class}\ k(\alpha_1, \ldots, \alpha_n) \ldots$ each type τ occurring in it (either after the : or in the member declarations) has to fulfill $FV(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}$,
3. the relation $\vdash <:$ on types, defined according to the rules listed in Fig. 1, must have no cycles, i.e. $\vdash\ k(\overline{\tau_1}) <: \sigma$ and $\vdash\ \sigma <: k(\overline{\tau_2})$ implies $k(\overline{\tau_1}) = k(\overline{\tau_2}) = \sigma$.

$$\frac{}{\vdash \tau <: \tau} \qquad \frac{\vdash \tau_1 <: \tau_2 \quad \vdash \tau_2 <: \tau_3}{\vdash \tau_1 <: \tau_3} \qquad \frac{\vdash \tau_1 <: \tau_2}{\vdash \eta(\tau_1) <: \eta(\tau_2)}$$

$$\frac{\vdash \tau_0 <: \sigma_0 \quad \vdash \sigma_1 <: \tau_1 \quad \ldots \quad \vdash \sigma_n <: \tau_n \quad n \geq 0}{(\tau_1, \ldots, \tau_n) \to \tau_0 <: (\sigma_1, \ldots, \sigma_n) \to \sigma_0}$$

$$\frac{\texttt{class}\ k(\overline{\alpha}) : \ldots, \tau, \ldots\ \{\ldots\} \in E}{\vdash k(\overline{\alpha}) <: \tau}$$

Fig. 1. Closure rules for subtyping.

This definition of $\vdash <:$ is a subset of runtime subtyping available in .NET 2.0 – in particular $\vdash \tau_1 <: \tau_2$ implies that the set of values of type τ_1 is contained within the set of values of type τ_2.

Additionally we define relation on type constructors $<: \subseteq \mathcal{V} \times \mathcal{V}$:

$$k_1 <: k_2 \;\Leftrightarrow\; \exists \overline{\tau_1}, \overline{\tau_2}, \vdash k_1(\overline{\tau_1}) <: k_2(\overline{\tau_2})$$

2.2 Expressions

The input to our type inference algorithm consists of an environment and an expression to be typed. We define the set $\mathcal{E}$ of *expressions* using the following abstract syntax:

$e ::=$	$\texttt{new}\ k$	(object construction)
$\mid$	x	(local value reference)
$\mid$	$e_0(e_1, \ldots, e_n)$	(function call)
$\mid$	$e.m$	(field access)
$\mid$	$\lambda(x_1, \ldots, x_n).\, e$	(function definition)
$\mid$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	(value binding)

where $n \geq 0$, k range over $\mathcal{K}$, m range over $\mathcal{M}$ and x (and x_i) range over infinite denumerable set $\mathcal{I}$ of *identifiers*.

The output is an error symbol or a *typed expression* (a member of the set $\mathcal{E}^T$) defined using the following abstract syntax:

$e^T ::=$	$\texttt{new}\ \tau$	(object construction)
$\mid$	x	(local value reference)
$\mid$	$e_0^T(e_1^T, \ldots, e_n^T)$	(function call)
$\mid$	$e^T.m^\tau$	(field access)
$\mid$	$\lambda(x_1 : \tau_1, \ldots, x_n : \tau_n).\, e^T$	(function definition)
$\mid$	$\texttt{let}\ x = e_1^T\ \texttt{in}\ e_2^T$	(value binding)

where $n \geq 0$, τ (and τ_i) range over $\mathcal{T}$ and other metavariables are defined as above.

The typed expressions are very similar to plain expressions, but the types of lambda variables are explicitly specified and member access is fully resolved. While the set $\mathcal{E}$ is considered to be equivalent to source Nemerle programs, the $\mathcal{E}^T$ set is considered a model for the CIL[4].

We require identifiers bound in the `let`-bindings and λ–abstractions to be unique and used only within their scope. We omit the trivial algorithm enforcing that for brevity.

3 Typing

For the rest of this section we fix an environment E.

3.1 The Constraint Solver

We use a device called the *constraint solver* to discover most facts about the code we are going to type. This section presents just the interface of the solver, implementations notes can be found in Sect. 3.5.

A constraint solver algorithm decides if a given set of subtyping inequations, has a solution.

Definition 1. *The* solution *of a constraint solver* $C = \{\sigma_i \lhd \tau_i\}_{i=1}^n$ *is a ground substitution* η *such that:*

$$\forall i \in \{1, \ldots, n\}. \;\; \vdash \eta(\sigma_i) <: \eta(\tau_i)$$

We say $C = \bot$ *iff* C *does not have a solution.*

Our constraint solver is constructive, that is it η as well as some other useful information can be extracted from it.

The solver itself is denoted by the letter C. We use notation $C + \{\sigma \lhd \tau\}$ for constraint solver C enriched with the inequation $\sigma \lhd \tau$. We further define:

$$C + \{\sigma_1 \lhd \tau_1, \ldots, \sigma_n \lhd \tau_n\} \equiv C + \{\sigma_1 \lhd \tau_1\} + \ldots + \{\sigma_n \lhd \tau_n\}$$
$$C + \{\sigma \lhd\rhd \tau\} \equiv C + \{\sigma \lhd \tau, \tau \lhd \sigma\}$$

Please note that the + operation is not a set theory sum operation, it is a notion for constraint solver algorithm invocation. Similarly the test $C = \bot$ is also a solver algorithm invocation.

3.2 The Algorithm

We use e as the entire expression we are going to type.

Let $Pos(e)$ be the set of all *positions* of subexpressions within e. The string $e\lceil_p$ where $p \in Pos(e)$ denotes a subexpression of e at the position p, while $e[e']_p$ denotes e where its subexpression at the position p has been replaced with e'.

[4] Common Intermediate Language, a stack based, high–level, typed assembly language (or bytecode) for the .NET platform. All the .NET compilers target it [11].

We use notation like $e\restriction_p = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ and later p_i for $i = 1, 2$ to denote position of e_i within e[5].

We use one type variable for each expression, and denote it α_p where $p \in Pos(e)$. In addition we have γ_x where $x \in \mathcal{I}$ for type variables assigned to identifiers. We also occasionally use additional type variables of the form β_k^p where $p \in Pos(e)$ and $k \in \mathbb{N}$.

The algorithm proceeds by evolving the solver C, the expression e and the set of positions P. It starts with an empty constraint solver, $P = Pos(e)$ and e as the expression to be typed. It then proceeds according to the inspection rules listed in Fig. 2.

We shall now define a few helper functions that we will use in the inspection rules.

Definition 2. *The* span *of type variable in a given solver is the set of type constructors it can be unified with:*

$$span_C(\alpha) = \{k \mid \exists \tau_1, \ldots, \tau_n.\ C + \{\alpha \lhd\rhd k(\tau_1, \ldots, \tau_n)\} \neq \bot\}$$

Definition 3.

$$hint_C(\alpha) = \begin{array}{l}\textbf{if } \exists k \in span_c(\alpha).\ \forall k' \in span_c(\alpha).\ k <: k' \textbf{ then } k \\ \textbf{else if } \exists k \in span_c(\alpha).\ \forall k' \in span_c(\alpha).\ k' <: k \textbf{ then } k \\ \textbf{else } \bot\end{array}$$

$hint(\alpha)$ is our current best guess of what will be assigned to α in the final solution. As we will see in Sect. 3.5 the hint is easily computable from our constraint solver representation[6].

Definition 4.

$$fieldsof_C^\alpha(k, \sigma, m) = \left\{ \langle k'(\overline{\beta}), \eta(\tau), \tau \rangle \;\middle|\; \begin{array}{l} k <: k' \ \wedge\ \eta = [\overline{\alpha} := \overline{\beta}] \wedge \\ C + \{\sigma \lhd k'(\overline{\beta}), \eta(\tau) \lhd \alpha\} \neq \bot \wedge \\ \texttt{class}\ k'(\overline{\alpha}) \ldots \{\ldots m\ :\ \tau \ldots\} \in E \end{array} \right\}$$

where $\overline{\beta}$ are fresh

This function looks for members of k with type compatible with α when accessed from object of type σ.

Definition 5.

$$\begin{array}{l} more\_specific_C^\tau(\sigma, \langle \sigma_1, \tau_1, \rho_1 \rangle, \langle \sigma_2, \tau_2, \rho_2 \rangle) \ \Leftrightarrow \\ \qquad C' + \{\tau_1 \lhd \tau_2\} = \bot \ \wedge\ C' + \{\tau_2 \lhd \tau_1\} \neq \bot \\ \quad \textbf{where}\ \ C' = C + \{\sigma \lhd \sigma_1, \tau_1 \lhd \tau, \sigma \lhd \sigma_2, \tau_2 \lhd \tau\} \end{array}$$

$$most\_specific_C^\tau(\sigma, A) = \{p \mid p \in A\ \wedge\ \forall p' \in A.\ \neg more\_specific_C^\tau(\sigma, p', p)\}$$

[5] Note that it is not the position within $e\restriction_p$.

[6] The reader may find it odd that the best choice is happily chosen rather randomly between the highest and lowest type possible. This function is however used to lookup members at which point we consider any clue about the type to be good.

$$(C, P \cup \{p\}, e) \rightsquigarrow (C + \{\gamma_x \lhd\rhd \alpha_p\}, P, e) \quad \text{when: } e\restriction_p = x$$

$$(C, P \cup \{p\}, e) \rightsquigarrow (C + \{\alpha_p \lhd\rhd \tau, \alpha_{p_1} \lhd \sigma\}, P, e[e_1.m^\rho]_p) \quad \text{when: } e\restriction_p = e_1.m,\ member_C^{\alpha_p}(\alpha_{p_1}, m) = \langle \sigma, \tau, \rho \rangle,\ O(p) = \bot$$

$$(C, P \cup \{p\}, e) \rightsquigarrow (C + \{\begin{matrix}\alpha_{p_0} \lhd\rhd (\beta_1^p, \ldots, \beta_n^p) \to \alpha_p, \\ \alpha_{p_1} \lhd \beta_1^p, \ldots, \alpha_{p_n} \lhd \beta_n^p\end{matrix}\}, P, e) \quad \text{when: } e\restriction_p = e_0(e_1, \ldots, e_n)$$

$$(C, P \cup \{p\}, e) \rightsquigarrow (C + \{\alpha_p \lhd\rhd (\gamma_{x_1}, \ldots, \gamma_{x_n}) \to \alpha_{p_1}\}, P, e[\lambda(x_1 : \gamma_{x_1}, \ldots, x_n : \gamma_{x_n}).\, e_1]_p) \quad \text{when: } e\restriction_p = \lambda(x_1, \ldots, x_n).\ e_1$$

$$(C, P \cup \{p\}, e) \rightsquigarrow (C + \{\gamma_x \lhd\rhd \alpha_{p_1}, \alpha_p \lhd\rhd \alpha_{p_2}\}, P, e) \quad \text{when: } e\restriction_p = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

$$(C, P \cup \{p\}, e) \rightsquigarrow (C + \{\alpha_p \lhd\rhd k(\beta_1^p, \ldots, \beta_n^p)\}, P, e[\texttt{new}\ \alpha_p]_p) \quad \text{when: } e\restriction_p = \texttt{new}\ k,\ \texttt{class}\ k(\alpha_1, \ldots, \alpha_n) \ldots \in E$$

Fig. 2. Inspection rules

The functions above are used for finding the most appropriate overload. They choose the less general type as "better" because the $\to$ type constructor is contravariant on the left hand side – so the actual function types chosen are the ones with most specific argument types.

If desired the rules above can be restricted to compare only function types and possibly only at the argument positions.

Definition 6.

$$member_C^\tau(\sigma, m) = most\_specific_C^\tau(\sigma, fieldsof_C^\tau(hint_C(\sigma), \sigma, m))$$

Finally the *member* function returns a set of appropriate member access resolutions when accessing m on σ. We furthermore require the field to have type τ in the solver C.

If by applying inspection rules we reach the state $(C, \emptyset, e)$ for some e and $C \neq \bot$, then we say that the type inference algorithm has *succeeded*, and the result is e, with the solution η of C applied to all the types contained in e.

If the algorithm reaches the state (C, P, e) where $C \neq \bot$, $P \neq \emptyset$ and no rule can be applied, then the algorithm is said to have *failed because of ambiguity*.

If the algorithm reaches the state $(\bot, P, e)$, then the algorithm is said to have *failed because of a type clash*.

3.3 Examples

In the example below we assume the following environment:

$$\begin{array}{l}\texttt{class}\ Animal()\ \{\\ \quad legs\ :\ ()\to Int()\\ \}\\ \texttt{class}\ Dog()\ :\ Animal()\ \{\}\\ \texttt{class}\ Cat()\ :\ Animal()\ \{\}\end{array}\qquad\begin{array}{l}\texttt{class}\ Int()\ \{\}\\ \texttt{class}\ Set(\alpha)\ \{\\ \quad add\ :\ (\alpha)\to Set(\alpha)\\ \}\end{array}$$

Let us now consider following piece of source program:

$$\begin{array}{l}\textbf{let}\ s_1\ =\ \texttt{new}\ Set\ \textbf{in}\\ \textbf{let}\ s_2\ =\ s_1.\texttt{add}(\texttt{new}\ Dog)\ \textbf{in}\\ \textbf{let}\ s_3\ =\ s_2.\texttt{add}(\texttt{new}\ Cat)\ \textbf{in}\\ s_3\end{array}$$

To see how the typing proceeds we show state of the constraint solver after typing each line of the input. We only show parts describing types of local variables as the are probably the most interesting. The first line results in:

$$\{\gamma_{s_1} \lhd\rhd Set(\beta_1)\}$$

then $s_1.add$ member access adds: $\{\gamma_{s_1} \lhd Set(\beta_2), \alpha_{s_1.add} \lhd\rhd \beta_2 \to Set(\beta_2)\}$ and calling this member adds: $\{Dog() \lhd \beta_1, \gamma_{s_2} \lhd\rhd Set(\beta_2)\}$ so the current state is:

$$\{\gamma_{s_1} \lhd\rhd Set(\beta_1), \gamma_{s_2} \lhd\rhd Set(\beta_1), Dog() \lhd \beta_1\}$$

next the third line adds: $\{\gamma_{s_2} \lhd Set(\beta_3), Cat() \lhd \beta_3, \gamma_{s_3} \lhd\rhd Set(\beta_3)\}$ so the current state is:

$$\{\gamma_{s_1} \lhd\rhd Set(\beta_1), \gamma_{s_2} \lhd\rhd Set(\beta_1), \gamma_{s_3} \lhd\rhd Set(\beta_1), Dog() \lhd \beta_1, Cat() \lhd \beta_1\}$$

which finally results in:

$$\{\gamma_{s_1} \lhd\rhd Set(\beta_1), \gamma_{s_2} \lhd\rhd Set(\beta_1), \gamma_{s_3} \lhd\rhd Set(\beta_1), Animal() \lhd \beta_1\}$$

As we can see the type of s_1 got generalized to $Set(Animal())$, even though it is not a subtype of $Set(Dog())$. We can now consider a more "delayed" example:

$$\begin{array}{l}\textbf{let}\ f\ =\ \lambda(x).\, x.legs\ \textbf{in}\\ f(\texttt{new}\ Cat)\end{array}$$

In this example typing fail to cope with the lambda expression (actually some constraints can be produced, but they are not very reasonable). We therefore first proceed with the second line obtaining:

$$\{\alpha_{\texttt{new}\ Cat} \lhd\rhd Cat(), \gamma_f \lhd\rhd (\beta_4) \to \alpha_{f(\texttt{new}\ Cat)}, Cat() \lhd \beta_4\}$$

so we can go back to the lambda expression and add $\beta_4 \lhd\rhd \gamma_x$ which makes *hint* return Cat on γ_x. This allows typing member access on x, which adds: $\{\alpha_{x.legs} \lhd \rhd Int(), \gamma_x \lhd Animal()\}$. So the type assigned to f is between $Animal() \to Int()$ and $Cat() \to Int()$. If we would use f again on $Dog()$, the type would get strictly $Animal() \to Int()$.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \texttt{new}\ \tau : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau \vdash e_2 : \sigma}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma} \qquad \frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_1 : \tau}{\Gamma \vdash \lambda(x_1 : \tau_1, \ldots, x_n : \tau_n).\, e_1 : (\tau_1, \ldots, \tau_n) \to \tau}$$

$$\frac{\Gamma \vdash e_0 : (\tau_1, \ldots, \tau_n) \to \sigma \qquad \Gamma \vdash e_1 : \tau_1' \quad \vdash \tau_1' <: \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n' \quad \vdash \tau_n' <: \tau_n}{\Gamma \vdash e_0(e_1, \ldots, e_n) : \sigma}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \vdash \sigma <: k(\overline{\rho}) \quad \texttt{class}\ k(\overline{\alpha}) \ldots \{\ldots m : \tau \ldots\} \in E \quad \theta = [\overline{\alpha} := \overline{\rho}]}{\Gamma \vdash e_1.m^{\tau} : \theta(\tau)}$$

Fig. 3. Typing rules for E_T.

3.4 Soundness

We will now define what do we mean by well–typed expressions. We are not going to prove subject reduction, as we are not interested directly in the dynamic safety of the code. We are however interested in the static property of CIL being verifiable. The .NET standard [11] specifies the subset of CIL that can be statically type checked, and is thus safe to execute. It is called *verifable CIL*. Our typing rules for E_T are an *ad–hoc* translation of these rules to our model of CIL. The rules are however pretty straightforward. They are listed in Fig. 3

First we can observe, that if e' is a result of a successful run of the type inference algorithm on e, then e is e' with additional type information erased. So the algorithm returns the same program, but typed. Now we can state validity of the output:

Theorem 1. *(soundness): If e' is a result of a successful run of the type inference algorithm on e, then e' is a verifiable typed expression.*

Proof. Let $(C, \emptyset, e')$ be the final configuration in the algorithm run. Let η be the solution of C. We will construct a proof of $\Gamma \vdash e' : \eta(\alpha_\epsilon)$, where ϵ is a root position ($e\restriction_\epsilon = e$), and $\Gamma = \{x : \eta(\gamma_x) \mid x \in \mathcal{I} \text{ occurs in } e'\}$. Moreover the environments in all the subtrees of the proof will be the same. This proof can be easily converted to the form where $\emptyset \vdash e' : \eta(\alpha_\epsilon)$ is at the root, because identifiers are unique and occur only in their respective scopes.

Lemma 1. *For any $p \in Pos(e')$ a proof can be constructed with $\Gamma \vdash e'\restriction_p : \eta(\alpha_p)$ in the root.*

The proof of the lemma is by induction on the structure of e'. All the cases are straightforward. We show the reasoning for member access as it is probably syntactically the most complicated. We have $e'\restriction_p = e_1.m^{\tau}$. The *member* function returns one of the tuples returned by the $fieldsof$ function, let's call it $\langle k'(\overline{\beta}), \eta'(\tau), \tau\rangle$. By $fieldsof$ definition we have $\texttt{class}\ k'(\overline{\alpha}) \ldots \{\ldots m : \tau \ldots\} \in E$ and $\eta' = [\overline{\alpha} := \overline{\beta}]$. By solver validity $\vdash \eta(\alpha_{p_1}) <: \eta(k'(\overline{\beta}))$, and by induction

hypothesis $\Gamma \vdash e_1 : \eta(\alpha_{p_1})$, so the θ from the typing rule is really $\theta = [\overline{\alpha} := \eta(\overline{\beta})]$. The final type assigned in the rule is then $\theta(\eta(\tau))$ which equals $\eta(\eta'(\tau))$, which by solver validity equals $\eta(\alpha_p)$. □

3.5 The Constraint Solver Implementation

We have already specified the solver interface in Sect. 3.1. This section sketches its implementation.

State Description The state of the constraint solver consists of a relation R and two functions: $\cdot^{\uparrow}$ and $\cdot_{\downarrow}$.

The relation $R \subseteq \mathcal{V} \times \mathcal{V}$ represents subtyping relations to be enforced on type variables ($\mathcal{V}$ is the set of type variables defined in Sect. 2).

Upper and lower limits on types assigned to given type descriptor in the final solutions are represented by the functions $\cdot^{\uparrow} : \mathcal{V} \to \mathcal{T}$ and $\cdot_{\downarrow} : \mathcal{V} \to \mathcal{T}$. The idea is that the limits tighten the space of possible solutions until they eventually meet. These functions are also used for computing the *hint* function.

The state of constraints solver is a triple $Q = (R, \cdot^{\uparrow}, \cdot_{\downarrow})$.

One can think about the solver as a directed graph (V, R) where additionally each vertex has upper and lower constraints. Each incoming constraint is decomposed into relations on type variables and possibly additional restrictions on upper and lower bounds. This decomposition makes sure that any substitution η fulfilling the following conditions in Q:

$$(1)\ \forall \alpha \in \mathcal{V}.\ \ \vdash \eta(\alpha^{\uparrow}) <: \eta(\alpha) \wedge \vdash \eta(\alpha) <: \eta(\alpha_{\downarrow})$$
$$(2)\ \forall (\alpha, \beta) \in R.\ \ \vdash \eta(\alpha) <: \eta(\beta)$$

is a solution of Q.

If the constraint being added is $\alpha \lhd k(\overline{\tau})$ ($k(\overline{\tau}) \lhd \alpha$), then we replace the previous lower (upper) limit with its intersection (sum) with $k(\overline{\tau})$. For constraints of the form $k_1(\overline{\tau}) \lhd k_2(\overline{\sigma})$ we lookup appropriate axiom and add constraints on $\overline{\tau}$ and $\overline{\sigma}$.

We use conservative approximations for sum and intersection types – it is respectively a common superclass and smaller of the two types (if such smaller type does not exists, we treat this as an error). This is one of reasons our constraint solver is not complete. We have however not found this to be a problem in practice.

Normalized Solver We say a solver $Q = (R, \cdot^{\uparrow}, \cdot_{\downarrow})$ is in *normalized* form if it fulfills the following conditions:

1. the directed graph $(\mathcal{V}, R)$ is closed with respect to transitivity and reflexivity, that is R is a partial preorder on $\mathcal{V}$,
2. $\forall \alpha.\ \ Q + \{\alpha^{\uparrow} \lhd \alpha_{\downarrow}\} = Q$,
3. $\forall (\alpha, \beta) \in R.\ \ Q + \{\alpha^{\uparrow} \lhd \beta^{\uparrow}, \alpha_{\downarrow} \lhd \beta_{\downarrow}\} = Q$.

If the above constraints are fulfilled then taking $\cdot^{\uparrow}$ or $\cdot_{\downarrow}$ as η will result in a proper solution.

The conditions above are maintained as invariants by simply adding constraints enforced by them in a fix point fashion.

3.6 Syntax Tree Inspection Order

The reader can note that the result of our algorithm depends on particular ordering of syntax tree inspection. But choosing the proper ordering is NP–hard (see proof in the appendix).

So while some heuristics may be invented here – they are not going to be simple. And because it is good for the programmer to understand what static type is a given expression going to have – any heuristic should be understandable by the programmer.

The easiest option here was to inspect the tree in a linear source code order. This is what we did – we first apply typing rules storing yet unresolved member access nodes in a simple FIFO queue. Next we try to type each member access in the queue, returning member accesses to it, if we still do not have enough information. We repeat this process until no more member accesses can be typed. If the queue is empty, the algorithm has succeeded, otherwise user needs to annotate member access to resolve ambiguity.

The linear inspection ordering also gives overload resolutions quite close to what the C# compiler would choose, when type annotations are specified. This is important, because programmers are likely to specify types explicitly if they relay on particular overload being chosen.

4 Complexity and the Implementation

It is easy to observe that the types can become exponentially big during type inference. It is possible, this can be cured using some kind of DAG's to represent types. Our implementation does not employ this method though. We have found that programmers do not write program with exponential types, which is much like the situation in ML [20]. Therefore the types stay at a reasonable size.

Practical experimentation shows the presented type inference scheme to be no more than two times slower compared to Cardelli's greedy inference [3]. This seems acceptable as the the inference process accounts for only about 30% of the compiler running time.

Of course the performance depends very much on the implementation. The initial one was several times slower, mainly due to the excessive amount of temporary type variables generated. Employing special cases for type variables that are only going to be unified with something else (as opposed to having proper upper and lower bounds) helped here enormously. Another problem with the implementation was related to the fact that we require the constraint solver to have a cheap copy operation (we sometimes use shallow backtracking, so the state needs to restored after failure). This was first achieved using a mainly

functional implementation. Switching to an imperative one with sophisticated copy–on–write tricks also helped with the performance.

5 Contributions

The main contribution of this paper is a practical application of the idea of inferring types in non–source–code order. Instead of the usual approach of analyzing the code in a linear order and possibly deferring some checks for later, we defer entire typing process of some syntax tree nodes. We have found this approach to work quite well in practice.

A very important advantage of this deferral technique compared to collecting constraints on types and resolving them later is extensibility. The Nemerle language can be extended by the user through pluggable compiler modules (macros), run during the typing process.

For example the `foreach` loop (used to iterate over a collection) is added this way. It can emit several special–case implementations for particular data structures (like list or arrays) and a default implementation for classes supporting the `IEnumerable` interface. So when the `foreach` macro is executed, and at this point of the typing process the type of the collection is not yet known, we would need to just fall back to the default (less efficient) implementation. However instead, the macro can defer its execution for later, when the type will be known.

The macro can also handle error reporting itself. When we discover that no new expression was resolved during a deferred typing queue run, we ask the first deferred typing (which also happens to be the first one in textual program order) to display an error message. This way it is simple and readable to the user, for example "cannot determine the type of collection, please specify it", instead of something like "constraint x of α is not fulfilled".

Nemerle supports several different kinds of members (fields, properties, indexers, methods), which all have some kind of special access and overloading resolution rules. In addition we have named parameters and implicit conversions, which also complicate things. In such a rich environment we found the flexibility given by the deferral technique quite appealing.

There are of course also disadvantages of this approach. We immediately loose principal types property (the type of function clearly depends on use site, therefore it cannot have any principal type itself). There are problems with generalizing function types, to make them polymorphic – we never know if some deferred computation can place some additional constraint on type variables that were just generalized. This is however an area of future research.

It is possible to extend the technique presented to use type schemes instead of types for member types, it is also possible to include possibly recursive lower bounds on type variables (which results in an implementation of F-bounded polymorphism [2]). However both features only obscure the notation and don't introduce anything new to the algorithm. They are however both present in the real implementation.

6 Related Work

Various type inference mechanisms are used in a wide variety of programming languages. Most them are research languages, but some have a wide audience, particularly languages of the ML family, like Standard ML [14], OCaml [13] and Haskell [12], but also by some implementations of dynamic languages for statically typed runtime environments like IronPython [9] and Boo [5]. The best known algorithm of type inference is the $\mathcal{W}$ algorithm [4]. It deals with parametric polymorphism and functional values, and with certain extensions with imperative features.

Probably the simplest mutation of the $\mathcal{W}$ algorithm to deal with subtyping is Cardelli's greedy algorithm [3]. It simply unifies variables as soon as possible without any possibility of using subtyping there. It as used as a basis of previous Nemerle inference engine.

Type inference is in general very well explored topic – for an up to date review of extensions here we refer the reader to [18]. However the area of nominal type systems with subtyping and parametric polymorphism (like the one in Generic Java [1] and C# 2.0 [10] [11]) occupies only a small fraction of research efforts. Moreover practical implementations are in short supply here. Works worth mentioning in this context include using global constraint solving for adding generic type annotations in Java bytecode [6] and techniques of local type inference [17] with extensions [16] [7] used in the Scala [15] programming language.

7 Future Work

The algorithm we have presented does not infer polymorphic types (it simply fails with a type clash when a function intended to be polymorphic is used for the second time with different types of the arguments). It can however handle polymorphic types with explicit annotations. It is an open question if it can be extended to infer polymorphic types in some cases.

We have proven NP–hardness of overloading in our system. However it can be possible to develop some better, simple heuristics here.

There are some techniques for improving error messages, like removal of less–certain constraints from the solver in place of an error in spirit of [8].

Acknowledgments I would like to thank Macrin Młotkowski for his continuous remarks about this paper, and Kamil Skalski for help getting the implementation running. I would also like to thank Gerard Murphy for his remarks and corrections.

References

1. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language.

In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
2. Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM Press.
3. Luca Cardelli. An implementation of $F_{<:}$. Technical Report 97, 130 Lytton Avenue, Palo Alto, California 94301, 1993.
4. Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.
5. Rodrigo B. de Oliveira. The Boo language, http://boo.codehaus.org/. 2004.
6. Alan Donovan and Michael D. Ernst. Inference of generic types in Java. Technical Report MIT/LCS/TR-889, 2003.
7. Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In X.Leroy, editor, *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04)*, pages 281–292, Venice, Italy, January 2004. ACM Press. Extended version available as Technical Report CMU-CS-04-117, March 2004.
8. Bastiaan Heeren, Johan Jeuring, S. Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical Report UU-CS-2002-009, Institute of Information and Computing Science, University Utrecht, Netherlands, February 2002.
9. Jim Hugunin. Ironpython: A fast Python implementation for .NET and Mono. In *PyCON*, Washington, DC, 2004.
10. ISO. *C# Language Specification, ISO/IEC 23270:2003*. International Organization for Standardization, 2003.
11. ISO. *Common Language Infrastructure, ISO/IEC 23271:2003*. International Organization for Standardization, 2003.
12. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.
13. Xavier Leroy. *The Objective Caml system: Documentation and user's manual*. 2000.
14. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, August 1990.
15. Martin Odersky. Report on the programming language Scala. Ecole Polytechnique Federale de Lausanne, Switzerland, http://scala.epfl.ch/, 2002.
16. Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, 2001.
17. 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, San Diego, California*, pages 252–265, New York, NY, 1998.
18. Franois Pottier and Didier Remy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
19. Geoffrey Seward Smith. Polymorphic type inference for languages with overloading and subtyping. Technical Report 91-1230, PhD thesis, Department of Computer Science, Cornell University, 1991.
20. Tomasz Wierzbicki. Reflections on complexity of ML type reconstruction.

8 Appendix: NP–Hardness of Syntax Tree Ordering

Theorem 2. *Choosing a syntax tree inspection order for which the typing will not fail is an NP–hard problem.*

Proof. We will encode a 3SAT problem in a program. The program will have a proper typing in a type system we have shown iff the corresponding 3SAT instance has a solution.

We need the following environment:

$$
\begin{array}{l}
\texttt{class}\ F()\ \{\} \\
\texttt{class}\ T()\ :\ F()\ \{\} \\
\texttt{class}\ Ops()\ \{ \\
\quad neg : (T()) \to F() \qquad\qquad neg : (F()) \to T() \\
\quad or : (T(), F(), F()) \to T() \quad or : (T(), T(), F()) \to T() \\
\quad or : (F(), T(), F()) \to T() \quad or : (T(), F(), T()) \to T() \\
\quad or : (F(), F(), T()) \to T() \quad or : (F(), T(), T()) \to T() \\
\quad or : (T(), T(), T()) \to T() \\
\}
\end{array}
$$

We will now construct an expression e for a 3SAT formulae Ψ. We start by adding:

$$\texttt{let}\ ops = \texttt{new}\ Ops\ \texttt{in}$$

to e. Now for each variable $v \in FV(\Psi)$ we add the following piece of program to e:

$$
\begin{array}{l}
\texttt{let}\ p_v = ops.neg(\texttt{new}\ T)\ \texttt{in} \\
\texttt{let}\ n_v = ops.neg(p_v)\ \texttt{in}
\end{array}
$$

Then we define a function $\widehat{\cdot}$ from literals in Ψ to V, such that:

$$\widehat{q} = p_q \qquad \widehat{\neg q} = n_q$$

Now for each clause $c_i = (l_1 \vee l_2 \vee l_3)$ where l_i are literals we add the following line to e:

$$\texttt{let}\ dummy_i = ops.or(\widehat{l_1}, \widehat{l_2}, \widehat{l_3})\ \texttt{in}$$

Finally, after last `in` we add *ops*.

Now the expression e posses a proper typing iff the corresponding 3SAT instance has a solution.

The crucial NP–hard choice is if we first type $ops.neg(\texttt{new}\ T)$ or $ops.neg(p_v)$. In the first case p_v will get type $F()$, as the most specific overload of *neg* will be chosen. Otherwise it will get type $T()$, because the most specific overload will be chosen in the second *neg* invocation and the restriction on return type of the first one will be placed. The proof is a straightforward encoding argument[7]. □

[7] The overloading resolution has been even proven undecidable in a less restricted environment [19].