Using Typings as Types

Casper Bach Poulsen, Peter D. Mosses, and Neil Sculthorpe

Department of Computer Science, Swansea University, UK
casperbp@gmail.com, p.d.mosses@swansea.ac.uk, n.a.sculthorpe@swansea.ac.uk

Abstract

In languages with dynamic scope, the free variables of an abstraction correspond to implicit parameters. The values of these parameters are determined by the bindings current where the abstraction is applied, but their allowed types can be checked statically. We give a novel type system for dynamic scope using types that involve type environments and intersections. We also explore how to give typing rules with a high degree of modularity using typings themselves as types.

1 Introduction and Background

Dynamic scope arises when the values of the free variables of a function body are those current when the function is applied, rather than when it was formed. Table 1 shows a conventional type system for a simply-typed λ-calculus with static scope (using notation following [4]). These rules are clearly unsound when the semantics requires dynamic scope.

In Sect. 2 we present a novel type system for a λ-calculus with dynamic scope. Here, abstractions with dynamic scope are first-class values: we let them be passed as arguments and returned by other abstractions, in contrast to previous type systems [1, 2, 6]. The required types include *typings* $A \vdash \tau$ and intersections $\sigma \wedge \tau$.

The benefits of higher-order abstractions with dynamic scope in programming are debatable [6]. Our motivation for considering them comes from component-based semantics (CBS) [3, 7]. In this framework, the semantics of a programming language is defined by translating it to so-called fundamental programming constructs ('funcons'). Functions with static scope are translated to funcon compositions of the form $\mathsf{close}(\mathsf{thunk}(e))$, where close takes an abstraction and returns its closure. The abstraction funcon $\mathsf{thunk}(e)$ is a value incorporating an unevaluated expression e; the free variables of e inherently have dynamic scope unless close is used.

CBS uses a modular variant of SOS for defining the dynamic semantics of each funcon independently [3]. In Sect. 3, we explore how to obtain modularity and scalability in type systems using richer forms of typings as types. As noted by Wells [8], typings in arbitrary type systems contain all the information from typing judgements other than the term being typed.

$$M, N ::= n \mid x \mid (M + N) \mid (\lambda x.M) \mid (M\ N)$$

$$\sigma, \tau ::= \mathsf{int} \mid t \mid (\sigma \to \tau)$$

$$\emptyset \vdash n : \mathsf{int} \tag{1}$$

$$\{x : \tau\} \vdash x : \tau \tag{2}$$

$$\frac{A \vdash M : \mathsf{int} \quad A \vdash N : \mathsf{int}}{A \vdash (M + N) : \mathsf{int}} \tag{3}$$

$$\frac{A_x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x.M) : (\sigma \to \tau)} \tag{4}$$

$$\frac{A \vdash M : (\sigma \to \tau) \quad A \vdash N : \sigma}{A \vdash (M\ N) : \tau} \tag{5}$$

$$\frac{A \vdash M : \tau}{A' \vdash M : \tau}\ {\scriptstyle A \subseteq A'} \tag{6}$$

Table 1: A conventional type system for static scope

2 A Type System for Dynamic Scope

When an abstraction $\lambda x.M$ with dynamic scope is applied to an argument, the context should provide a bound value for each of its free variables. The types of these bound values, along with the type of the argument to be associated with x, should ensure that M is well-typed. We obtain types for abstractions with dynamic scope by enriching the types used in Table 1 with typings $A \vdash \tau$, indicating the requirement of a context providing variables according to the type environment A:

$$\sigma, \tau ::= \mathsf{int} \mid t \mid (\sigma \to \tau) \mid (A \vdash \tau) \mid (\sigma \wedge \tau)$$

The type of an abstraction with dynamic scope is of the form $A \vdash (\sigma \to \tau)$, where σ and τ may also involve type environments.

Rules (7–9) in Table 2 replace (4–6) in Table 1. The type environment A_x is as A except that any type constraint for x is ignored. Type environment union $A_1 \cup A_2$ assumes A_1 and A_2 have disjoint domains, whereas intersection $A_1 \wedge A_2$ combines the type constraints of A_1 and A_2 with intersection of the types of any common variables. The intersection $A \wedge A'$ is needed in (8) because x may be required to have one type when evaluating M to an abstraction, and a different type when evaluating the body of the abstraction; similarly in (10).

$$\frac{A \vdash M : \tau}{\emptyset \vdash \lambda x.M : (A_x \vdash (\sigma \to \tau))}\ {\scriptstyle (A(x)=\sigma)} \qquad (7)$$

$$\frac{A \vdash M : (A' \vdash (\sigma \to \tau)) \quad A \vdash N : \sigma}{(A \wedge A') \vdash (M\ N) : \tau} \qquad (8)$$

$$\frac{A \vdash M : \sigma}{A' \vdash M : \tau}\ {\scriptstyle (A \vdash \sigma) <: (A' \vdash \tau)} \qquad (9)$$

$$M ::= \cdots \mid \mathsf{close}(M)$$

$$\frac{A \vdash M : (A' \vdash (\sigma \to \tau))}{(A \wedge A') \vdash \mathsf{close}(M) : (\emptyset \vdash (\sigma \to \tau))} \qquad (10)$$

Table 2: A type system for dynamic scope, and an extension with explicit closures

A typing rule corresponding to (4) can be derived for the term $\mathsf{close}(\lambda x.M)$ from (7) and (10), showing that such abstractions are supposed to have static scope (cp. [3, §3.4]).

The subtype relation on types (implicit in the use of $\sigma \wedge \tau$) is also used on type environments and typings. Some of its properties are shown in Table 3.

$$\frac{\sigma' <: \sigma \qquad \tau <: \tau'}{(\sigma \to \tau) <: (\sigma' \to \tau')} \qquad (11)$$

$$\frac{A' <: A \qquad \sigma <: \tau}{(A \vdash \sigma) <: (A' \vdash \tau)} \qquad (12)$$

$$(A_x \cup \{x : \tau\}) <: A_x \qquad (13)$$

$$\frac{A_x <: A'_x \qquad \tau <: \tau'}{(A_x \cup \{x : \tau\}) <: (A'_x \cup \{x : \tau'\})} \qquad (14)$$

Table 3: Some subtyping rules

As an illustrative example, consider the dynamically scoped term

$$M = (\lambda f.\ (\lambda y.\ f\ 1)\ 2)\ (\lambda x.\ x + y)$$

The abstraction $\lambda x.\ x + y$ has a free variable y that has to be bound to a value of type int when the abstraction gets applied, and we can derive $\emptyset \vdash (\lambda x.\ x + y) : (\{y : \mathsf{int}\} \vdash (\mathsf{int} \to \mathsf{int}))$. The application $(\lambda y.\ f\ 1)\ 2$ satisfies this constraint, and we can derive $\emptyset \vdash M : \mathsf{int}$.

3 Modularity

For specifying a transition from M to M' in dynamic semantics, CBS provides the notation $R \vdash \langle M, S \rangle \xrightarrow{W} \langle M', S' \rangle$ where R consists of read-only entities (e.g. environments ρ), S and S' consist of mutable entities (e.g. stores σ), and W consists of write-only entities (e.g. outputs α). Entities can be omitted, and are then implicitly propagated in transition rules. For instance,

$$\frac{\rho \vdash \langle M, \sigma \rangle \xrightarrow{\alpha} \langle M', \sigma' \rangle}{\rho \vdash \langle (M;N), \sigma \rangle \xrightarrow{\alpha} \langle (M';N), \sigma' \rangle} \quad \text{can be abbreviated to} \quad \frac{M \to M'}{(M;N) \to (M';N)}$$

and $\rho \vdash \langle ((\,);N), \sigma \rangle \overset{\cdot}{\to} \langle N, \sigma \rangle$ abbreviated to $((\,);N) \to N$. Modular foundations for such abbreviations are provided by a rule-by-rule translation to MSOS (see [3]) where all auxiliary entities are incorporated in labels.

To obtain a similar degree of modularity for typing rules, we propose to let the types of terms be general typings that contain the corresponding types of all auxiliary entities. For example, consider $R \vdash \langle \tau, S \rangle \overset{W}{\Longrightarrow} \langle \tau', S' \rangle$ where R may include the type environment (A), S and S' may include store types (Σ), and W may include interactive behaviour types (α). The type τ is for explicit arguments of abstractions, and τ' for computed values. Using implicit propagation for omitted entities, we could abbreviate

$$\frac{M : A \vdash \langle \tau, \Sigma_1 \rangle \overset{\alpha_1}{\Longrightarrow} \langle \mathsf{int}, \Sigma_2 \rangle \quad N : A \vdash \langle \tau, \Sigma_2 \rangle \overset{\alpha_2}{\Longrightarrow} \langle \mathsf{int}, \Sigma_3 \rangle}{(M + N) : A \vdash \langle \tau, \Sigma_1 \rangle \overset{\alpha_1 \cup \alpha_2}{\Longrightarrow} \langle \mathsf{int}, \Sigma_3 \rangle} \quad \text{to} \quad \frac{M : \Rightarrow \mathsf{int} \quad N : \Rightarrow \mathsf{int}}{(M + N) : \Rightarrow \mathsf{int}}$$

and abbreviate $n : \emptyset \vdash \langle \cdot, \Sigma \rangle \overset{\cdot}{\Rightarrow} \langle \mathsf{int}, \Sigma \rangle$ to $n : \Rightarrow \mathsf{int}$.

4 Conclusion and Future Work

We have given a novel type system for a λ-calculus with dynamic scope, illustrated the typing of terms where abstractions with free variables are passed as arguments, and suggested a technique to obtain modularity in type systems. We now aim to prove the soundness of the type system, extend it with universal quantification and recursive types, and obtain principal typings [4, 5, 8].

References

[1] R. Chugh. A fix for dynamic scope. In *ML '13*, 2013.

[2] R. Chugh. IsoLATE: A type system for self-recursion. In *ESOP '15*, volume 9032 of *LNCS*, pages 257–282. Springer, 2015.

[3] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In *TAOSD XII*, volume 8989 of *LNCS*, pages 132–179. Springer, 2015.

[4] T. Jim. What are principal typings and what are they good for? In *POPL '96*, pages 42–53. ACM, 1996.

[5] T. Jim. A polar type system. In *ITRS '00*, pages 323–338. Carleton Scientific, 2000.

[6] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL '00*, pages 108–118. ACM, 2000.

[7] PLanCompS: Programming language components and specifications. `http://www.plancomps.org`.

[8] J. Wells. The essence of principal typings. In *ICALP '02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

Appendix

This appendix shows how we derived the typing claimed in Sect. 2, and sketches a denotational semantics for dynamic scope. It is not intended for inclusion in the final version.

The claimed typing is:

$$\emptyset \vdash (\lambda f.\ (\lambda y.\ f\ 1)\ 2)\ (\lambda x.\ x + y) : \mathsf{int} \tag{15}$$

For any A and τ' we derive:

$$\{f : (A \vdash (\mathsf{int} \to \tau'))\} \wedge A \vdash f\ 1 : \tau' \tag{16}$$

$$\emptyset \vdash (\lambda y.\ f\ 1) : (\ (\{f : (A_y \cup \{y : \tau\} \vdash (\mathsf{int} \to \tau'))\} \wedge A_y) \vdash (\tau \to \tau')\ ) \tag{17}$$

$$\{f : (A_y \cup \{y : \mathsf{int}\} \vdash (\mathsf{int} \to \tau'))\} \wedge A_y \vdash (\lambda y.\ f\ 1)\ 2 : \tau' \tag{18}$$

$$\emptyset \vdash \lambda f.\ (\lambda y.\ f\ 1)\ 2 : (A_{yf} \vdash ((A_y \cup \{y : \mathsf{int}\} \vdash \mathsf{int} \to \tau') \to \tau')) \tag{19}$$

Taking $A = \{y : \mathsf{int}\}$ gives:

$$\emptyset \vdash \lambda f.\ (\lambda y.\ f\ 1)\ 2 : (\emptyset \vdash (\{y : \mathsf{int}\} \vdash \mathsf{int} \to \tau') \to \tau')) \tag{20}$$

We also have:

$$\emptyset \vdash (\lambda x.\ x + y) : (\{y : \mathsf{int}\} \vdash (\mathsf{int} \to \mathsf{int})) \tag{21}$$

Taking $\tau' = \mathsf{int}$, the claimed typing follows.

A denotational semantics for dynamic scope

$$\begin{aligned}
V &= Z + F \\
F &= Env \to V \to V \\
Env &= Var \to V \\
[[M]] &: Env \to V \\
[[n]]\rho &= n \\
[[x]]\rho &= \rho(x) \\
[[M + N]]\rho &= [[M]]\rho|_Z + [[N]]\rho|_Z \\
[[\lambda x.M]]\rho &= \lambda\rho'.\lambda v.[[M]](\rho'[x \mapsto v]) \\
[[M\ N]]\rho &= ([[M]]\rho|_F)(\rho)([[N]]\rho)
\end{aligned}$$