Modest annotations with intersection types

Aleksy Schubert
alx@mimuw.edu.pl
Faculty of Mathematics, Informatics and Mechanics,
The University of Warsaw,
ul. Banacha 2,
02–097 Warsaw,
Poland

June 21, 2024

Abstract

We propose an intersection type system which is located between traditional Curry-style presentation of intersection types and a Church style. In this system one can omit type annotations of variables provided that they are simple types. Types with intersections are obligatory.

1 Introduction

In many contemporary programming languages (e.g. Python, Ruby, PHP) static typing is absent as its presence impairs flexibility and requires additional work [Tra09, NBD$^+$05]. Empirical studies show that indeed strong typing discipline can make standard programming tasks in one person projects take longer than when programmed in untyped languages [Han10]. Still, in many empirical settings strong typing shows its advantages [RF21, GBB17, HKR$^+$14, KRS$^+$12]. An important insight can be drawn from one study [SF14] on Groovy, a language where types are optional. It shows that types are popular when high-level structuring constructs are employed, i.e. modules, while are less used in scripts, tests and code that undergoes frequent changes.

This means that although types are not welcome in small-scale programming, they are essential in bigger, complex projects that involve many people. This suggests that a possible fruitful design choice for a programming language is not to avoid types altogether (as e.g. in [Jim00]), but allow simple types to be inferred automatically, while any more complex types, as relevant to more complicated developments, to be present obligatorily. This design choice is consistent with the common observation that type systems with more complicated type forms have the type reconstruction problem either undecidable [Pot80] or of high computational complexity [KW99].

$$\frac{}{\Gamma, x : \sigma \vdash x^\sigma : \sigma}(Var) \qquad \frac{\sigma \in T_\to}{\Gamma, x : \sigma \vdash x : \sigma}(VarS)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau}(\to I) \quad \frac{\Gamma, x : \sigma \vdash M : \tau \quad \sigma \in T_\to}{\Gamma \vdash \lambda x.M : \sigma \to \tau}(\to IS)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}(\to E)$$

$$\frac{\Gamma \vdash M : \sigma \wedge \tau}{\Gamma \vdash M : \sigma}(\wedge E1) \qquad \frac{\Gamma \vdash M : \sigma \wedge \tau}{\Gamma \vdash M : \tau}(\wedge E2)$$

Figure 1: Type assignment rules

2 Preliminaries

First, we present basic notions. As a departure point, we take the system by Liquori and Ronchi Della Rocca [LR07]. Types are generated from the grammar

$$\sigma, \tau \quad ::= \quad \alpha \mid \sigma \wedge \tau \mid \sigma \to \tau$$

where α ranges over a denumerable set of type atoms. We omit in the grammar parentheses and consider it to be a definition of a specific inductive type. *Simple types* are types without the $\wedge$ connective. The set of simple types is written $T_\to$. The terms are generated from the grammar

$$M, N \quad ::= \quad x^\sigma \mid x \mid \lambda x : \sigma.M \mid \lambda x.M \mid MN$$

where x^σ ranges over a denumerable set of term variables of type σ and x ranges over term variables for which the intent is to assign simple types to them. Terms are understood up to α-equivalence, i.e. the renaming of variables bound by λ.

A context Γ is a finite set of pairs $x : \sigma$ such that if $x : \sigma \in \Gamma$ and $x : \tau \in \Gamma$ then $\sigma = \tau$. The type assignment rules of our system are presented in Figure 1. In the following, we write $\Gamma \vdash M : \sigma$ to state that the judgement is derivable. We write $|M|$ for the term M with erased type type information. We write $\Gamma \vdash_\wedge M : \sigma$ when the judgement is derivable in the standard Curry style system.

Notably, the system does not include $(\wedge\ I)$ rule. This is a non-trivial design choice as demonstrated by Kurata and Takahashi [KT95]. We also do not use the traditional presentation of the system with help of a type ordering. This is also a non-trivial design choice that reduces the expressive power of the system. However, this makes the presentation of the type inference procedure simpler.

Traditionally, we define β-reduction for the system as

$$(\lambda x : \sigma.M)N \to_\beta M[x := N], \qquad (\lambda x.M)N \to_\beta M[x := N]$$

where $M[x := N]$ denotes the usual capture avoiding substitution.

The type system enjoys the following basic properties.

Proposition 2.1 `(basic properties)`

- *(Substitution correctness)*
  If $\Gamma, x : \sigma \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$ *then* $\Gamma \vdash M[x := N] : \tau$.
- *(Subject reduction)*
  If $\Gamma, x : \sigma \vdash M : \tau$ *and* $M \to_\beta M'$ *then* $\Gamma, x : \sigma \vdash M' : \tau$.
- *(Type erasure) If* $\Gamma \vdash M : \tau$ *then* $\Gamma \vdash_\wedge |M| : \tau$.
- *(Type erasure and reduction)*
  If $\Gamma \vdash M : \tau$ *and* $M \to_\beta N$ *then* $|M| \to_\beta |N|$.
- *(Strong normalisation)*
  If $\Gamma \vdash M : \tau$ *then there is no infinite* $\to_\beta$*-reduction sequence that starts from* M.

A non-standard development is required to prove decidability of the system. The approach requires introduction of unification-like constraints. We need here two kinds of unknowns. The first of them are first-order variables, $X, Y, etc.$. A solution assigns to such variables expressions that may consist both of type constructors and variables. Certain variables in our setting can assume only expressions that are build of type atoms and arrows. In such case we annotate variable with arrow like in $X^\to, Y^\to$, etc.

The second kind of variables, F, G, etc., represent second-order operations and are always applied to an expression (e.g. $\mathsf{F}(\alpha \wedge \beta)$). However, these do not represent the traditional second-order variables where a substitution S assigns to such a variable F some function such as $\lambda x.\sigma$ and the result of application of the solution $S(\mathsf{F}(\tau))$ is the expression $\sigma[x := \tau]$. These are also not traditional expansion variables where a substitution S assigns to such a variable F some expression $\Box \wedge \cdots \wedge \Box$ and the result of application of the solution $S(\mathsf{F}(\tau))$ is the expression $\tau^1 \wedge \cdots \wedge \tau^n$ where τ^i are versions of the expression τ with appropriately renamed variables. While this approach seems to be the right one in case the system contains introductions of the $\wedge$ connective, it seems not to be adequate in case when these are absent.

In our setting a substitution S assigns to a second-order variable F an expression generated from the grammar

$$A, B := \Box \mid \blacksquare \mid A \wedge B$$

with exactly one occurrence of $\Box$. Assume that $S(\mathsf{F}) = A$. The result of application of the substitution to an expression $\mathsf{F}\sigma$ is $A(\sigma)$ where

- $\Box(\sigma) = \sigma$, • $\blacksquare(\sigma)$ is undefined,
- $A_0 \wedge A_1(\sigma_0 \wedge \sigma_1) = A_i(\sigma_i)$ in case $A_{1-i}(\sigma_{1-1})$ is undefined for $i \in \{0, 1\}$,
- $A_0 \wedge A_1(\sigma_0 \wedge \sigma_1) = A_0(\sigma_0) \wedge A_1(\sigma_1)$ in case $A_i(\sigma_i)$ is defined for all $i \in \{0, 1\}$,

Note that the operation $A(\cdot)$ is not defined for expressions of the form $\sigma \to \tau$. This is another situation where the substitution S does not have a well defined result. To differentiate our second-order variables from other flavours of these, we call them *projection variables*.

The unification constraints are generated by a procedure Constr that is invoked as $\mathsf{Constr}(\Gamma; M)$ where Γ is a context under which we look for type of the term M. The procedure operates on projection variables of the form F_N and first-order variables of the form X_N for each subterm N of M. As a little abuse of the notation we tacitly distinguish in these variable annotations different occurrences of the same subterm (precise formulation would require additional marking of different occurrences with address of the subterm in M, but this would obscure the notation).

The ultimate goal of the procedure is to generate a set of equation constraints E such that each solution S of E assigns a first-order expression to X_M so that $\Gamma \vdash M : S(X_M)$. A substitution S is a solution of an equation $A \doteq B$ when both $S(A)$ and $S(B)$ are well defined and $S(A) = S(B)$. The substitution is a solution of E when it is a solution of each element of E.

Definition 1 ($\mathsf{Constr}(\Gamma; M)$)
Given Γ, M we define the function $\mathsf{Constr}(\Gamma; M)$ by induction on M as follows.

1. $\mathsf{Constr}(\Gamma; x^{\sigma}) = \{\Gamma(x) \doteq \sigma, \sigma \doteq X_x\}$,
2. $\mathsf{Constr}(\Gamma; x) = \{\Gamma(x) \doteq X_x^{\to}\}$,
3. $\mathsf{Constr}(\Gamma; MN; \vec{X}) = \texttt{let } E_M = \mathsf{Constr}(\Gamma; M) \texttt{ and } E_N = \mathsf{Constr}(\Gamma; N)$
   $\texttt{in } \{\mathsf{F}_M(X_M) \doteq \mathsf{F}_N(X_N) \to X_{MN}\} \cup E_M \cup E_N$,
4. $\mathsf{Constr}(\Gamma; \lambda x{:}\sigma.M) = \texttt{let } E_M = \mathsf{Constr}(\Gamma, x{:}\sigma; M)$
   $\texttt{in } \{X_{\lambda x:A.M} \doteq \sigma \to X_M\} \cup E_M$,
5. $\mathsf{Constr}(\Gamma; \lambda x.M) = \texttt{let } E_M = \mathsf{Constr}(\Gamma, x : X_x^{\to}; M)$
   $\texttt{in } \{X_{\lambda x:A.M} \doteq X_x^{\to} \to X_M\} \cup E_M$.

We can now prove the relevant soundness and completeness property.

Proposition 2.2 (`Soundness and completeness of Constr`)
Assume that $\mathsf{Constr}(\Gamma; M) = E$.

1. *If S is a solution of E then $\Gamma \vdash M : S(X_M)$.*
2. *If $\Gamma \vdash M : A$, for some A then there is a solution S of E such that $A = S(X_M)$.*

Proof:
We omit the proof due to the space constraints. □

It remains to show that the constraints can be solved. We do this by means of a terminating reduction $\rightsquigarrow_s$ defined as follows. In the following $E \uplus E'$ means the disjoint sum of sets E and E'. We use here also Dec_0 and Dec_1 that are defined further below.

- $E \uplus \{A \doteq A\} \leadsto_s E$,
- $E \uplus \{A_1 \to A_2 \doteq B_1 \to B_2\} \leadsto_s E \uplus \{A_1 \doteq B_1, A_2 \doteq B_2\}$,
- $E \uplus \{A_1 \wedge A_2 \doteq B_1 \wedge B_2\} \leadsto_s E \uplus \{A_1 \doteq B_1, A_2 \doteq B_2\}$,
- $E \uplus \{Y \doteq B\} \leadsto_s E[Y := B]$ provided that B does not contain Y,
- $E \uplus \{X^{\to} \doteq B\} \leadsto_s E[X^{\to} := \mathrm{Dec}_0(B)] \cup \mathrm{Dec}_1(B)$ provided that $\mathrm{Dec}_0(B) \neq B_1 \wedge B_2$ for some B_1, B_2 and that B does not contain $X^{\to}$,
- $E \uplus \{\mathsf{F}(X^{\to}) \doteq B\} \leadsto_s E[\mathsf{F} := \square] \uplus \{X^{\to} \doteq B[\mathsf{F} := \square]\}$,
- $E \uplus \{\mathsf{F}(A_1 \to A_2) \doteq B\} \leadsto_s E[\mathsf{F} := \square] \uplus \{(A_1 \to A_2 \doteq B)[\mathsf{F} := \square]\}$,
- $E \uplus \{\mathsf{F}(X) \doteq B\} \leadsto_s E[X := X^{\to}] \uplus \{\mathsf{F}(X^{\to}) \doteq B[X := X^{\to}]\}$,
- $E \uplus \{\mathsf{F}(X) \doteq B\} \leadsto_s E[X := X_1 \wedge X_2] \uplus \{\mathsf{F}(X_1 \wedge X_2) \doteq B[X := X_1 \wedge X_2]\}$,
- $E \uplus \{\mathsf{F}(A_1 \wedge A_2) \doteq B\} \leadsto_s E[\mathsf{F} := \square] \uplus \{(A_1 \wedge A_2 \doteq B)[\mathsf{F} := \square]\}$,
- $E \uplus \{\mathsf{F}(A_1 \wedge A_2) \doteq B\} \leadsto_s E[\mathsf{F} := \blacksquare \wedge \mathsf{F}] \uplus \{(\mathsf{F}(A_2) \doteq B)[\mathsf{F} := \blacksquare \wedge \mathsf{F}]\}$,
- $E \uplus \{\mathsf{F}(A_1 \wedge A_2) \doteq B\} \leadsto_s E[\mathsf{F} := \mathsf{F} \wedge \blacksquare] \uplus \{(\mathsf{F}(A_1) \doteq B)[\mathsf{F} := \mathsf{F} \wedge \blacksquare]\}$.

The operations Dec_0 and Dec_1 are defined by mutual recursion as follows

- $\mathrm{Dec}_0(X) = X^{\to}$, $\mathrm{Dec}_1(X) = \{X \doteq X^{\to}\}$,
- $\mathrm{Dec}_0(X^{\to}) = X^{\to}$, $\mathrm{Dec}_1(X^{\to}) = \emptyset$,
- $\mathrm{Dec}_0(\mathsf{F}(B)) = X^{\to}_{\mathsf{F}(B)}$, $\mathrm{Dec}_1(\mathsf{F}(B)) = \mathrm{Dec}_1(B) \cup \{\mathsf{F}(\mathrm{Dec}_0(B)) \doteq X^{\to}_{\mathsf{F}(B)}\}$,
- $\mathrm{Dec}_0(A_1 \wedge A_2) = \mathrm{Dec}_0(A_1) \wedge \mathrm{Dec}_0(A_2)$, $\mathrm{Dec}_1(A_1 \wedge A_2) = \mathrm{Dec}_1(A_1) \cup \mathrm{Dec}_1(A_2)$,
- $\mathrm{Dec}_0(A_1 \to A_2) = \mathrm{Dec}_0(A_1) \to \mathrm{Dec}_0(A_2)$, $\mathrm{Dec}_1(A_1 \to A_2) = \mathrm{Dec}_1(A_1) \cup \mathrm{Dec}_1(A_2)$, provided that $\mathrm{Dec}_0(A_i) \neq B_1 \wedge B_2$ for some B_1, B_2.

We can now prove the following statement.

Proposition 2.3 `(Soundness and completeness of` $\leadsto_s$`)`

- *If* $E \leadsto_s^* \emptyset$ *then there is a solution of* E.
- *If* E *has a solution then* $E \leadsto_s^* \emptyset$.

The relation $\leadsto_s$ as presented above is not terminating. However, we can prove that constraints generated by Constr have special form that allows us to provide a reduction strategy of $\leadsto_s$ that terminates.

References

[GBB17] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 758–769, 2017.

[Han10] Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, oct 2010.

[HKR$^+$14] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empir. Softw. Eng.*, 19(5):1335–1382, 2014.

[Jim00] Trevor Jim. A polar type system. In José D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and J. B. Wells, editors, *ICALP Workshops 2000, Proceedings of the Satelite Workshops of the 27th International Colloquium on Automata, Languages and Programming, Geneva, Switzerland, July 9-15, 2000*, pages 323–338. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[KRS$^+$12] Sebastian Kleinschmager, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Eric Tanter. Do static type systems improve the maintainability of software systems? an empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162, 2012.

[KT95] Toshihiko Kurata and Masako Takahashi. Decidable properties of intersection type systems. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 297–311, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[KW99] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 161–174. ACM, 1999.

[LR07] Luigi Liquori and Simona Ronchi Della Rocca. Intersection-types à la church. *Information and Computation*, 205(9):1371–1386, 2007.

[NBD$^+$05] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the revival of dynamic languages. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition - 4th International Workshop,* *SC@ETAPS 2005, Edinburgh, UK, April 9, 2005, Revised Selected Papers*, volume 3628 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2005.

[Pot80] Garrel Pottinger. *To H. B. Curry, Essays in Combinatory Logic, Lambda-Calculus and Formalism*, chapter A type assignment for the strongly normalizabile λ-terms. Academic press, 1980. Ed. J. R. Hindley and J. P. Seldin.

[RF21] Jaakko Rinta-Filppula. Is static type checking worth it? on the pros and cons of adding a static type checker to an existing codebase. Master's thesis, Faculty of Information Technology and Communication Sciences, Tampere University, 2021.

[SF14] Carlos Souza and Eduardo Figueiredo. How do programmers use optional typing? an empirical study. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, page 109–120, New York, NY, USA, 2014. Association for Computing Machinery.

[Tra09] Laurence Tratt. Dynamically typed languages. *Adv. Comput.*, 77:149–184, 2009.