Type inference

Type inference refers to the automatic deduction of the data type of an expression in a programming language. If some, but not all, type annotations are already present, it is termed **type reconstruction**.

It is a feature present in some strongly statically typed languages. It is often characteristic of functional programming languages in general. Some languages that include type inference include C++11, C# (starting with version 3.0), Clean, D, F#, FreeBASIC, Go, Haskell, ML, Nim, OCaml, Opa, RPython, Rust, Scala, Swift, and Visual Basic (starting with version 9.0). The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while still permitting type checking.

1 Nontechnical explanation

In most programming languages, all values have a data type explicitly declared at compile time, limiting the values a particular expression can take on at run-time. Increasingly, just-in-time compilation renders the distinction between run time and compile time moot. However, historically, if the type of a value is known only at run-time, these languages are dynamically typed. In other languages, the type of an expression is known only at compile time; these languages are statically typed. In statically typed languages, the input and output types of functions and local variables ordinarily must be explicitly provided by type annotations. For example, in C:

```
int addone(int x) { int result; /* declare integer result */ result = x + 1; return result; }
```

The signature of this function definition, int addone(int x), declares that addone is a function that takes one argument, an integer, and returns an integer. int result; declares that the local variable result is an integer. In a hypothetical language supporting type inference, the code might be written like this instead:

```
addone(x) { var result; /* inferred-type variable result */ var result2; /* inferred-type variable result #2 */ result = x + 1; result2 = x + 1.0; /* this line won't work (in the proposed language) */ return result; }
```

This is identical to how code is written in the language Dart, except that it is subject to some added constraints as described below. It would be possible to *infer* the types of all the variables at compile time. In the example above, the compiler would infer that result and x have type integer since the constant 1 is type integer, and hence that addone is a function int -> int. The variable result2 isn't used in a legal manner, so it wouldn't have a type.

In the imaginary language in which the last example is written, the compiler would assume that, in the absence of information to the contrary, + takes two integers and returns one integer. (This is how it works in, for example, OCaml). From this, the type inferencer can infer that the type of x + 1 is an integer, which means result is an integer and thus the return value of addone is an integer. Similarly, since + requires both of its arguments be of the same type, x must be an integer, and thus, addone accepts one integer as an argument.

However, in the subsequent line, *result2* is calculated by adding a decimal 1.0 with floating-point arithmetic, causing a conflict in the use of x for both integer and floating-point expressions. The correct type-inference algorithm for such a situation has been known since 1958 and has been known to be correct since 1982. It revisits the prior inferences and uses the most general type from the outset: in this case floating-point. This can however have detrimental implications, for instance using a floating-point from the outset can introduce precision issues that would have not been there with an integer type.

Frequently, however, degenerate type-inference algorithms are used that cannot backtrack and instead generate an error message in such a situation. This behavior may be preferable as type inference may not always be neutral algorithmically, as illustrated by the prior floating-point precision issue.

An algorithm of intermediate generality implicitly declares *result2* as a floating-point variable, and the addition implicitly converts x to a floating point. This can be correct if the calling contexts never supply a floating point argument. Such a situation shows the difference between *type inference*, which does not involve type conversion, and implicit type conversion, which forces data to a different data type, often without restrictions.

Finally, a significant downside of complex type-inference algorithm is that the resulting type inference resolution is not going to be obvious to humans (notably because of the backtracking), which can be detrimental as code is primarily intended to be comprehensible to humans.

The recent emergence of just-in-time compilation allows for hybrid approaches where the type of arguments supplied by the various calling context is known at compile time, and can generate a large number of compiled ver-

sions of the same function. Each compiled version can then be optimized for a different set of types. For instance, JIT compilation allows there to be at least two compiled versions of *addone*:

> A version that accepts an integer input and uses implicit type conversion.
>
> A version that accepts a floating-point number as input and uses floating point instructions throughout.

2 Technical description

Type inference is the ability to automatically deduce, either partially or fully, the type of an expression at compile time. The compiler is often able to infer the type of a variable or the type signature of a function, without explicit type annotations having been given. In many cases, it is possible to omit type annotations from a program completely if the type inference system is robust enough, or the program or language is simple enough.

To obtain the information required to infer the type of an expression, the compiler either gathers this information as an aggregate and subsequent reduction of the type annotations given for its subexpressions, or through an implicit understanding of the type of various atomic values (e.g. true : Bool; 42 : Integer; 3.14159 : Real; etc.). It is through recognition of the eventual reduction of expressions to implicitly typed atomic values that the compiler for a type inferring language is able to compile a program completely without type annotations.

In complex forms of higher-order programming and polymorphism, it is not always possible for the compiler to infer as much, and type annotations are occasionally necessary for disambiguation. For instance, type inference with polymorphic recursion is known to be undecidable. Furthermore, explicit type annotations can be used to optimize code by forcing the compiler to use a more specific (faster/smaller) type than it had inferred.[1]

Relative to program analysis, type inference is a special case of points-to analysis that uses a type abstraction on pointer targets.

3 Example

For example, let us consider the Haskell function map, which applies a function to each element of a list, and may be defined as:

```
map f [] = [] map f (first:rest) = f first : map f rest
```

Type inference on the map function proceeds (intuitively) as follows. map is a function of two arguments, so its type is constrained to be of the form a → b → c. In Haskell, the patterns [] and (first:rest) always match lists, so the second argument must be a list type: b = [d] for some type d. Its first argument f is applied to the argument first, which must have type d, corresponding with the type in the list argument, so f :: d → e (:: means "is of type") for some type e. The return value of map f, finally, is a list of whatever f produces, so [e].

Putting the parts together, we obtain map :: (d → e) → [d] → [e]. Nothing is special about the type variables, so we can simply relabel this as

map :: (a → b) → [a] → [b]

It turns out that this is also the most general type, since no further constraints apply. Note that the inferred type of map is parametrically polymorphic: The type of the arguments and results of f are not inferred, but left as type variables, and so map can be applied to functions and lists of various types, as long as the actual types match in each invocation.

4 Hindley–Milner type inference algorithm

Main article: Hindley–Milner type system

The algorithm first used to perform type inference is now informally termed the Hindley–Milner algorithm, although the algorithm should properly be attributed to Damas and Milner.[2]

The origin of this algorithm is the type inference algorithm for the simply typed lambda calculus that was devised by Haskell Curry and Robert Feys in 1958. In 1969 J. Roger Hindley extended this work and proved that their algorithm always inferred the most general type. In 1978 Robin Milner,[3] independently of Hindley's work, provided an equivalent algorithm, Algorithm W. In 1982 Luis Damas[2] finally proved that Milner's algorithm is complete and extended it to support systems with polymorphic references.

5 Side-effects of using the most general type

By design, type inference, especially correct (backtracking) type inference will introduce use of the most general type appropriate, however this can have implications as more general types may not always be algorithmically neutral, the typical cases being:

- floating-point being considered as a general type of integer, while floating-point will introduce precision issues

- variant/dynamic types being considered as a general type of other types, which will introduce casting rules and comparison that could be different, for instance such types use the '+' operator for both numeric additions and string concatenations, but what operation is performed is determined dynamically rather than statically

6 References

[1] Bryan O'Sullivan; Don Stewart; John Goerzen (2008). "Chapter 25. Profiling and optimization". *Real World Haskell*. O'Reilly.

[2] Damas, Luis; Milner, Robin (1982), "Principal type-schemes for functional programs", *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on principles of programming languages* (PDF), ACM, pp. 207–212

[3] Milner, Robin (1978), "A Theory of Type Polymorphism in Programming", *Jcss*, **17**: 348–375

7 External links

- Archived e-mail message by Roger Hindley, explains history of type inference
- Polymorphic Type Inference by Michael Schwartzbach, gives an overview of Polymorphic type inference.
- Basic Typechecking paper by Luca Cardelli, describes algorithm, includes implementation in Modula-2
- Implementation of Hindley-Milner type inference in Scala, by Andrew Forrest (retrieved July 30, 2009)
- Implementation of Hindley-Milner in Perl 5, by Nikita Borisov at the Wayback Machine (archived February 18, 2007)
- What is Hindley-Milner? (and why is it cool?) Explains Hindley-Milner, examples in Scala

8 Text and image sources, contributors, and licenses

8.1 Text

- **Type inference** *Source:* https://en.wikipedia.org/wiki/Type_inference?oldid=757359782 *Contributors:* Damian Yerrick, MarXidad, B4hand, Axlrosen, LittleDan, Dysprosia, Zoicon5, Jackson~enwiki, Ruakh, EvanED, Tea2min, Ancheta Wis, Connelly, Leonard G., Jabowery, Neilc, Ascánder, Spayrard, Euyyn, R. S. Shaw, Koper, Alansohn, Nighthawk4211, Ruud Koot, LinkTiger, Marudubshinki, Qwertyus, TheLaughingMan, Gfxmonk, Rjwilmsi, ErikHaugen, Debajit, Ground Zero, Bgwhite, YurikBot, Gaius Cornelius, Dogcow, SamuelRiv, Cedar101, That Guy, From That Show!, SmackBot, Aardvark92, Mgreenbe, Episteme-jp, Cybercobra, Almkglor, Jhammerb, Talandor, Jonathan S. Shapiro, Isaacdealey, Gregbard, Torc2, Thijs!bot, Oerjan, Igodard, Magioladitis, Gwern, Kyralessa, SparsityProblem, Semi Virgil, Daniel5Ko, Owengibbins, Jerryobject, AncientPC, Classicalecon, Adrianwn, Excirial, PixelBot, ChuckEsterbrook, Addbot, Ghettoblaster, Gasper.azman, Jarble, Peni, Yobot, Ptbotgourou, Ljaun, MrBlueSky, AnomieBOT, Rubinbot, Citation bot, FrescoBot, Денис Владимирович, Sae1962, Citation bot 1, RandomDSdevel, MastiBot, Francis Lima, ProjectSHiNKiROU, Gabaix, Gf uip, GoingBatty, ClueBot NG, HongxuChen, Clegoues, Helpful Pixie Bot, Cobalt pen, 786b6364, Mahendran Kathirvel, MatejLach, GreenC bot and Anonymous: 110

8.2 Images

8.3 Content license

- Creative Commons Attribution-Share Alike 3.0