Science of Computer Programming 95 (2014) 254–271

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Deriving a complete type inference for Hindley–Milner and vector sizes using expansion ☆

Axel Simon

Technische Universität München, Lehrstuhl für Informatik 2, Garching b. München, Germany

HIGHLIGHTS

- A derivation of a complete type inference algorithm from the denotational semantics.
- An inference for vector types of polymorphic size.
- Practical implementation and application of polymorphic recursion.

ARTICLE INFO

Article history:
Received 22 May 2013
Received in revised form 8 January 2014
Accepted 18 March 2014
Available online 25 March 2014

Keywords:
Type inference
Completeness
Abstract interpretation
Polymorphic recursion
Vector size inference

ABSTRACT

Type inference and program analysis both infer static properties about a program. Yet, they are constructed using very different techniques. We reconcile both approaches by deriving a type inference from a denotational semantics using abstract interpretation. We observe that completeness results in the abstract interpretation literature can be used to derive type inferences that are backward complete, implying that type annotations cannot improve the result of type inference, thus making type annotations optional. The resulting algorithm is similar to that of Milner–Mycroft, that is, it infers Hindley–Milner types while allowing for polymorphic recursion. Although undecidable, we present a practical check that reliably distinguishes typeable from untypeable programs. Instead of type schemes, we use *expansion* to instantiate types. Since our expansion operator is agnostic to the abstract domain, we are able to apply it not only to types. We illustrate this by inferring the size of vector types using systems of linear equalities and present practical uses of polymorphic recursion using vector types.

1. Introduction

New computer languages are continuously designed, often in the form of domain-specific languages (DSLs). One design aspect of a DSL is if the language should be statically typed and, if so, whether type inference should be used. Type inference is likely to be ruled out since addressing the special language features of a DSL in the type inference may turn into an open-ended research problem. We therefore propose to apply the abstract interpretation framework [2] to constructively derive a type inference algorithm. The motivation is that a variety of abstract domains have been developed over the years that can potentially be re-used for type inference, thereby going beyond Herbrand abstractions (type expressions containing type variables) commonly used in type inference. However, even if a set of domains are chosen and some type inference rules are put forth, different front-ends for the DSL might implement the type inference slightly differently, thus infer

☆ Supported by DFG Emmy Noether program SI 1579/1 and Microsoft SEIF 2013 grant.
E-mail address: Axel.Simon@in.tum.de.

http://dx.doi.org/10.1016/j.scico.2014.03.005
0167-6423/

different types for the same input and thereby accept a different set of programs. In order to address this issue, we propose to require the type inference to be backward complete, meaning that it infers best (most general) types for every function and expression in the program. The benefit is that it is easy to decide if an implementation of the type inference is incorrect or imprecise (its inferred types are unsound or not the best) and that type annotations are not required since there is no need to refine a type that is best (although the DSL may allow them for documentation purposes). These qualities also apply to inferences that deliver *principal typings* [9], however, this notion additionally requires that the inference never approximates a result which is not possible in the Hindley–Milner type system [21]. We thus say that our analysis infers the best typing. We give a constructive way to create type inferences that infer best typings by building on a result from the abstract interpretation literature [15] that shows that, once certain properties hold, a backward complete type inference can be derived by simply abstracting the semantics of the language. However, the presence of any branching construct (an **`if`** or a **`case`** statement) makes the derivation of a backward complete type inference impossible, as illustrated by the following example:

Example 1. Consider the semantics of the Haskell expression **`if`** `(f x)` **`then`** `42` **`else`** `[]` which is $\bot$ if `(f x)` does not terminate, 42 if it returns **`True`** or `[]` (an empty list) if it returns **`False`**. The corresponding type of these results are a (a type variable), `Int` or [a]. In case the evaluation of `(f x)` never terminates, then a is indeed the best type of the expression. Since determining whether `(f x)` may terminate is undecidable, the common approach in type inference is to ignore the outcome of the condition by assuming that both branches of the **`if`**-statement are taken. This is, in fact, an abstraction of the denotational semantics that our approach makes explicit.

Thus, we derive a backward complete type inference with respect to a semantics that performs a non-deterministic choice between the two branches of a conditional. With respect to this slightly abstracted semantics, we constructively derive a backward complete type inference, meaning that it infers the best type within the universe of types. Deriving this backward complete inference requires an abstraction function α from program values to types that encodes the difference between **let** and λ-bound variables that the Hindley–Milner type system prescribes. Hence, the first contribution of this paper is this derivation including the abstraction map which, to our knowledge, is novel. Previous approaches [1] only used a concretization function γ that is insufficient to show backward completeness [15].

Our second contribution is to derive type inference rules for the special language features occurring in a DSL. In particular, we designed a language to specify instruction decoders [16] which are programs that turn a byte stream into processor instructions. These programs make heavy use of bit-vectors whose type is |a| where a is a type variable denoting the size of the vector. Consider the decoding of an Atmel AVR microcontroller instruction `ORI` (read "or with immediate") which has the bit pattern `0110kkkkddddkkkk`. Here, each `k` denotes one bit of a constant and `d` denotes a register. Note that the constant bits are not contiguous. Our DSL allows an instruction to be specified exactly as this bit pattern. For this to work, each `k` is in fact a function that reads one bit and appends it to a bit vector `ks` that is initially empty. Analogous for `d` and `ds`. These bit vectors are later used to construct the appropriate arguments of the instruction. Assuming that the bits read by the function `d` are `d0`, `d1`, `d2`, and `d3`, the following calculation is performed internally (here `''` denotes the empty bit-vector and `^` denotes the concatenation of bit vectors):

```
let ds0 = ''; ds1 = ds0 ^ d1; ds2 = ds1 ^ d2; ds3 = ds2 ^ d3; ds4 = ds3 ^ d4 in
case ds4 of '0000' -> R0; -- turn the bit vector into a datatype representing registers
            '0001' -> R1;
            ...
```

Since `ds4` is matched against several 4-bit vectors, it is clear that `ds4` : |4|. Furthermore, `ds0` : |0|. However, inferring the sizes of the other bit vectors requires the instantiation of the concatenation function `^` whose type can be given as $|a| \to |b| \to |c|$ with the additional arithmetic constraint that $a + b = c$. Intuitively, it seems clear that calculating an instance of this type, say $|d| \to |e| \to |f|$, also needs to duplicate the size information to $d + e = f$. In general, this task might be more complicated if several equalities over a, b, c exist, so the question arises how calculating an instance can be performed in a principled manner. To this end, we observe that a relational *expand* operation [17] can be used for instantiating both, the type and the size information. Expansion replaces the syntactic operation of instantiating a type scheme with a semantic operation on the abstract domain of Herbrand abstractions (which is used to track the types of variables). Due to its semantic nature, we can derive how to apply expansion as part of deriving the type inference. While an expand operator has been proposed for instantiating types in System F [12], it was specific to types and its correctness was shown with respect to a universe of types containing type schemes. Our derived type inference is correct by construction [2], thereby making a proof with respect to inference rules using type schemes unnecessary. Analogously, the use of the expand operator on the domain of affine equations [10] can also be derived from the semantics, thereby suggesting that the expand operation can be useful when enhancing type systems with other abstract domains. Demonstrating the utility of expansion is our third contribution.

The derived type inference algorithm corresponds to the algorithm proposed by Mycroft [14], extended by the inference of vectors sizes. Mycroft's (and our) algorithm allows for polymorphic recursion, that is, given a definition **let** $f\ x = e_1$ **in** e_2, the expression e_1 may call f with different types. While polymorphic recursion is hardly required in everyday programs, we show that it has interesting applications in a type system with vector sizes. Henglein showed that polymorphic recursion