Mini Projects MP1
Type Equality, Bidirectional Typing, or Exceptions

15-814: Types and Programming Languages
Frank Pfenning

Due Thursday, November 5, 2020
150 points

Please pick one among the following mini-projects and hand in your report in PDF form on Canvas. You may do your chosen mini-project solo or team up with a classmate.

As with homework assignments, you are allowed to use online resources such as papers and technical reports, but you must give credit to the sources you consult.

1 When Are Two Types Equal?

In programming language theory we often distinguish between *negative types* whose values we *can not observe*, and *positive types* whose values we *can observe*. A *purely positive type* has no negative types embedded in it.

$$\text{Purely positive type} \quad \tau^+ \quad ::= \quad \tau_1^+ \times \tau_2^+ \mid 1 \mid \tau_1^+ + \tau_2^+ \mid 0 \mid \rho\alpha^+.\, \tau^+ \mid \alpha^+$$

We say that two purely positive types τ^+ and σ^+ are *equivalent* if for all v *value* we have $\cdot \vdash v : \tau^+$ if and only if $\cdot \vdash v : \sigma^+$. We assume here the values do not contain any type information (as we have been doing throughout the semester). Note that this is a much stricter notion than type isomorphism.

1. Give examples of types that are equivalent but not syntactically equal (modulo variable renaming, as always).

2. Define a judgment $\tau^+ \equiv \sigma^+$ via rules of inference so that $\tau^+ \equiv \sigma^+$ can be derived if and only if the types τ^+ and σ^+ are equivalent. If this is difficult to achieve, your judgment should at least be *sound* (see part 3). You may need to define some auxiliary judgments along the way.

3. Sketch the proof of soundness of your judgment, which shows that if $\tau^+ \equiv \sigma^+$ then τ^+ and σ^+ are equivalent in the sense of containing the same values. Show two different cases of the proof.

4. Describe an algorithm for deciding type equivalence for purely positive types. This could, for example, consist of a set of rules (maybe even the set of rules from part 2), where the algorithm is a systematic procedure for constructing derivations of a proposed equivalence.

5. For negative types (omitting parametric polymorphism for simplicity) we postulate that function types are only equivalent to function types with equivalent domains and codomains, and lazy pairs types are only equivalent to lazy pair types with equivalent component types. Because our language of expressions has recursion, function types and lazy pairs are never empty.

   Extend your rules and algorithm from parts 2 and 4 so it applies to all types, not just purely positive ones.

2 Bidirectional Typing

In lecture we briefly talked about how *type inference* exploits the syntax-directed nature of the typing rules and then reads off and solves a collection of constraints on the types. For our full language, however, this quite difficult due to (variadic) sums. Instead, the implementation of LAMBDA uses a so-called *bidirectional algorithm*. This is based on two separate judgments

$\Gamma \vdash e \Rightarrow \tau$ which holds if e *synthesizes* the type τ (which must be unique), and

$\Gamma \vdash e \Leftarrow \tau$ which holds if e *checks against* type τ.

For the simply-typed case, these are defined by the following rules:

$$\frac{\Gamma, x : \tau \vdash e \Leftarrow \sigma}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau \to \sigma}\ \textsf{chk/lam} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \quad \tau' \equiv \tau}{\Gamma \vdash e \Leftarrow \tau}\ \textsf{chk/syn}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}\ \textsf{syn/var} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau_1}\ \textsf{syn/app}$$

In the simply-typed case you may take type equivalence $\tau' \equiv \tau$ as syntactic equality $\tau = \tau'$. For the general case, see mini project 1, which you may assume as an oracle in this project if needed.

1. Bidirectional typing is *sound* with respect to typing: Prove that if $\Gamma \vdash e \Leftarrow \tau$ or $\Gamma \vdash e \Rightarrow \tau$ then $\Gamma \vdash e : \tau$.

2. Bidirectional typing is *not complete* with respect to typing: provide a concrete counterexample e and τ such that $\cdot \vdash e : \tau$ but neither $\cdot \vdash e \Rightarrow \tau$ nor $\cdot \vdash e \Leftarrow \tau$.

3. Characterize the λ-terms that can be checked against a type and those that synthesize a type.

4. Extend the syntax of expressions by allowing an explicit type annotations $(e : \tau)$ and extend the bidirectional rules to cover them.

5. We define the erasure of a term to simply remove all type annotations. Prove that if $\Gamma \vdash e : \tau$ then there is exists an e' such that $\Gamma \vdash e' \Leftarrow \tau$ and e is the erasure of e'.

6. Extend the system of bidirectional typing to cover the expressions for types $\tau_1 \times \tau_2$, 1, $\tau_1 + \tau_2$, 0, $\rho\alpha.\, \tau$, and $\forall \alpha.\, \tau$. They should have the properties explored in parts 1–5, but you do not need prove them.

3 First-Class Exceptions

We would like to generalize exceptions further so they can be returned by functions, passed as arguments, and dynamically created. For this purpose we create a new type exn. We treat the type exn as a disjoint sum

$$\mathsf{exn} = (\mathbf{Match} : 1) + (\mathbf{DivByZero} : 1) + \cdots$$

except that the programmer can add new alternatives to the sum with a (top-level) declaration

$$\mathsf{exn} = \mathsf{exn} + (i : \tau)$$

For example, in the absence of strings in the language, we could number different error exceptions instead of relying on the general Match exception by including

$$\mathsf{exn} = \mathsf{exn} + (\mathbf{Error} : \mathit{nat})$$

and then let the programmer raise Error k to indicate error number k.

A key property to keep in mind that $e : \mathit{exn}$ is an ordinary expression (and its value can be passed around) and raise e requires e to be an exception that can be raised, changing the control flow. Generalized pattern matching should now work to match against exceptions, as they are ordinary values.

1. Give the syntax for a language with first-class exceptions, general nested pattern matching, and a try construct to handle exceptions. You do not need to repeat the types and terms already present in our language.
2. Write out the typing rules for any new constructs, and those that are different from the ones in Lectures 12 and 13. You should assume that the type exn is fixed throughout the typing derivation because it can only be extended at the top level.
3. For extending the K machine there are two options: (a) a raised exception will unwind the stack frame-by-frame until a suitable handler is found, and (b) the machine maintains two stacks, one for normal control flow and one for exception handlers. Explore both options. You may extend the set of possible machine states beyond $k \triangleright e$ and $k \triangleleft v$, and you may also modify the information present in the state, at your discretion.
4. State the preservation, progress, and canonical form theorems for your preferred K machine with first-class exceptions. You do not need to prove them, but you should convince yourself that they hold.
5. Explain, using with suitable examples or counterexamplesm why it is important that the exn type can only be extended with fresh alternatives and not otherwise be modified.