Comp 411
Principles of Programming Languages
Lecture 21
Extending the (Simply) Typed λ-Calculus to λ-Languages with Functional Constructs

Corky Cartwright

March 22, 2023

Accomodating Standard Ground Types

Any realistic statically typed function language includes ground types like `bool` and `int` (and many more such as `char` and `float`).

Hence, the definition of types (type expressions) `τ` looks like:

```
  τ ::= int | bool | ... | τ → τ
```

and the base environment (which is empty in the simply typed lambda calculus) contains types for all of the primitive functions and operators. (We interpret operators as abbreviated syntax for conventional function applications.)

To support multi-ary functions, we also need to introduce Cartesian products of types `τ × τ` as an extra clause after `τ → τ` above. (The association convention (left or right) does not matter (either choice works), so we will ignore it. We will use this extension without further comment when we need it.

How Do We Type Functional Constructs?

Examples:

- Conditional expressions
- New algebraic (inductively defined) data types
- Recursive let

General answer:

(i) for each new syntactic construct like conditionals, we introduce a new rule;

(ii) for each new form of data (which only requires adding new constants [including functions] to the languge), we simply augment the set of type constructors by the new type constructor (*e.g.* `int-list`) and augment the contents of the base type environment by the new constants (including primitive functions) and their types.

Let's look at each example in our list above.

Typing Conditional Expressions

Note: if conditional expressions are simply written as applications of a ternary `if` operator, then all we need to do is add `if` to the set of constant symbols and add the type of `if` to the base type environment (assuming our typing framework is polymorphic which we will explain later). In fact, our new rule for the `if` construct simply codifies the same typing constraints (but does not introduce a new constant symbol or rely on polymorphism).

Our conditional expressions presume the existence of a bool type

$$\frac{\Gamma \vdash \mathtt{B:bool};\ \Gamma \vdash \mathtt{M}:\tau;\ \Gamma \vdash \mathtt{N}:\tau}{\Gamma \vdash \mathtt{if\ B\ then\ M\ else\ N} : \tau}$$

This rule is parametric in τ; it can be any type. If we treated `if` as a ternary function constant, we need a polymorphic type for τ. Note the abstraction and application inference rules are parametric in the embedded types as well.

Typing New Forms of Data

Assume we define some new form of data in a program. In structurally typed programming languages, all data values have a unique type. Hence, when a new union type is introduced, all values of that type must be disjoint from all existing types. This invariant is maintained by forcing all of the components of a new union type to be tagged. This data construction is called a discriminated union.

Example: binary trees

```
BT :: = leaf(int) | branch(BT, BT)
```

In ML-like languages, new forms of data are introduced in `datatype` declarations which have an abstract syntax similar to our example. In ML, the names of accessors (selectors) are implicit because pattern-matching notation is used to extract the fields of constructed data objects.

Note the that the “equation” introducing a new data type looks almost exactly like a production in a context-free-grammar. The only difference is that the constructions are trees rather than strings.

Typing New Forms of Data cont.

Given

```
 BT :: = leaf(int) | branch(BT, BT)
```

we augment the set of type expressions by the primitive type **`BT`** and the base typing environment by the declarations:

```
  make-leaf: int → BT
  make-branch: BT x BT → BT
  leaf-1: BT → int
  branch-1: BT → BT
  branch-2: BT → BT
```

assuming that we use the name **`leaf-1`** for the accessor for **`make-leaf`** and the names **`branch-1`** and **`branch-2`** for the accessors for **`make-branch`**. In languages with pattern matching like ML, the names of the accessors (can be implicit because pattern matching notation makes them unnecessary.

In ML-like languages, data type definitions are typically lexically scoped so **`datatype`** statements have an abstract syntax like **`let`** with an explicit body which is the scope of the definition. We will ignore scoping for data definitions and fix the data types for any program that we consider. (In Java, data definitions do not have scope. Visibility is an administrative notion not a semantic one.) Values of a given type can escape from their scope which creates some interesting semantic problems.

Key intuition: to accommodate new forms of data, we only need to add new primitive types (corresponding to the new data) and entries (the types of new constants) to the base type environment Γ_0.

Typing Let and Recursive Let

The typing rule for pure **`let`** is simply an abbreviation for a combination of the abstraction and application rules:

$$\frac{\Gamma \vdash M:\sigma;\quad \Gamma \cup \{x:\sigma\} \vdash N:\tau}{\Gamma \vdash \texttt{let}\ x:\sigma := M\ \texttt{in}\ N : \tau} \quad \text{(pure let rule)}$$

The corresponding rule for recursive **`let`** is:

$$\frac{\Gamma \cup \{x:\sigma\} \vdash M:\sigma;\quad \Gamma \cup \{x:\sigma\} \vdash N:\tau}{\Gamma \vdash \texttt{let}\ x:\sigma := M\ \texttt{in}\ N : \tau} \quad \text{(letrec rule)}$$

which differs only in one small (but very important!) detail: the type environment for proving $M:\sigma$.

Type Soundness

If a type system is properly designed, we can prove that it guarantees some (fairly weak) semantic properties hold. In particular, we can prove that the type system is *sound*, i.e. that well-typed programs (those that "type check") "never go wrong". More precisely, we can prove (using a simple syntactic, substitution based [often called *small-step*] semantics) that every program expression `M` of type τ either:

(i) Evaluates to a value of type τ.

(ii) Generates a run-time error (and aborts) where type errors are *not* classified as run-time errors.

(iii) Diverges.

Note that expression `M` cannot evaluate to a value that does not belong to type τ, which is extremely important.

The standard approach to proving type soundness breaks down into two steps:

(i) Type preservation: every reduction rule preserves the type of the reduced expression.

(ii) The syntactic evaluation process never get "stuck" where the program text cannot be reduced but does not denote a *value* (an answer).

Type preservation proofs typically proceed by induction on the length of evaluations (which look exactly like the hand evaluations we have learned to do for Jam). These proofs are typically straightforward but tedious because there are many cases (reduction rules applied in the last evaluation step) to consider. The "stuck-free" proof proceeds by structural induction on the form of closed ASTs; we simply show that every typable closed AST that is not a value is reducible.

A Glimpse at the Technical Details

To prove type-soundness in languages with run-time errors involving operations that are not *strict* (do not necessarily evaluate all of their arguments) like conditional expressions (**`if`** in Jam) and short-circuit **`&`** and **`|`**, we need to formalize aborting errors in our syntactic semantics by including the constant **`abort`** in our language and reductions to **`abort`** in our table specifying the behavior of primitive operations (function constants) applied to type correct arguments. The constant **`abort`** *is not a value*; if it ever appears as the leftmost unreduced expression in a computation, the computation immediately aborts with a run-time error. We have already been doing the same computation steps informally to handle run-time errors in hand evaluations.

This approach is mathematically precise and correct, but it means the statement "well-typed programs never go wrong" is a bit of an exaggeration. More accurately, *well-typed programs never generate type errors* in the course of syntactic evaluation. These type errors are modeled as "stuck" states, but run-time errors (like **`1/0`**) other than type errors still exist.

Type soundness proofs were originally done using a denotational semantics including an error element for type errors (modeled by stuck states in our syntactic semantics) and a separate error element for run-time errors that are not type errors (modeled by **`abort`** in our syntactic semantics). Note that both of these error elements are distinct from divergence ($\perp$).

Type Reconstruction

- Given an untyped expression `M` in a λ-language with constants, can we determine if it has a typing in the corresponding simply typed λ-language?
- Yes! Moreover the *reconstructed type* **τ** (sometimes ambiguously called the *inferred type*) may contain free type variables that can be instantiated (replaced) by arbitrary type expressions (perhaps containing type variables. These instantiations correspond to valid types for **τ**. The reconstructed type **τ** with the free type variables is called the *principal* (most general) type of `M`. (More advanced PL courses often prove this result.)
- Languages in the ML family (Haskell, Ocaml, …) perform type reconstruction.