Proving Type Safety

Kim B. Bruce
Pomona College College

October 16, 2008

1 Introduction

The purpose of this brief write-up is to show how one can formally prove (at least partially) the correctness of a type system for a language with respect to its formal semantics. The basic idea is that what the type checker tells you statically should correspond to the value you obtain by actually evaluating the term. In particular, if the type checker tells you that the type of a term is `T`, then the value obtained by evaluating that term should also be of type `T`. This is usually called a type preservation theorem.

There is a second theorem, usually termed "Progress", that is generally needed to prove type safety. This theorem states that well typed terms never get "stuck". That is, if `e` is a term with type `T`, then either it computes to a final legitimate value or it runs forever. More precisely, the computation never gets to a point where no rule is applicable.

It turns out that "progress" theorems are difficult to prove using the kind of natural semantics that we have used here. As a result, we will not attempt to prove that second theorem. Nevertheless we hope it is intuitively clear that the rules that we have written down are sufficient to avoid getting "stuck".

2 Terms of Typed PCF

We begin by defining a typed version of PCF. The syntax differs from that of untyped PCF by adding a type annotation to bound variables in function and recursive definitions: `fn (x:T) => body` and `rec (r:T) => body`. The grammar for types is as follows:

```
T :: = Int | Bool | T -> T
```

Expressions are defined similarly to those in PCF:

```
e ::= x | n | true | false | succ | pred | iszero |
      if e then e else e | (fn (x:T) => e) | (e e) | rec (x:T) => e
```

In this grammar, `n` stands for any non-negative integer, `x` is an identifier, and `T` is a type.

3 Type-checking rules

We can specify type-checking rules for this language in a way similar to that for arithmetic expressions. However, type checking will be done with respect to a "type environment," `E`, which is an association of strings to types (similar to the regular environment in the environment-based interpreter that was a mapping from strings to values). Thus if `E(x) = Int`, then identifier `x` has type `Int` in `E`. (See slides 6 and 7 of Lecture 13 for similar rules.)

```
E |- n: Int,  if n is an integer

E |- true: Bool,  E |- false: Bool

E |- succ: Int → Int,  E |- pred: Int → Int

E |- iszero: Int → Bool

E |- x: T,  if E(x) = T

E |- e: Bool      E |- e1: T      E |- e2: T
-----------------------------------------
         E |- if e then e1 else e2 : T

E |- f: T → U      E |- x: T
-----------------------------
         E |- (f x): U


        E[x:T] |- body: U
-------------------------------
E |- (fn (x:T) => body): T → U
```

In the last two rules, `E[x:T]` stands for the type environment obtained by adding to `E` the binding of identifier `x` to have type `T`. In other words, in order to determine the type of `fn (x:T) => body` in the presence of `E`, determine the type, `U`, of body in the presence of the updated environment, `E[x:T]`. The type of the function is then `T -> U`.

Aside from the `if` and `fn` clauses, the other type-checking rules are straightforward. When type-checking `AST_IF`, the type of the result should be the same as for each clause (we must be sure they are the same!).

It is easy to see (formally, by induction on the proof of typing) that if `E |- e: T` then all free variables of `e` are contained in the domain of `E`.

4 Natural semantics

The semantic rules for typed PCF are exactly the same as for untyped PCF as the type annotations are simply ignored. Recall from our discussion of the environment-based semantics of PCF, the only non-error *values* obtained by evaluating PCF terms are integers, the boolean values *true* and *false*, the interpretations of primitive functions *succ*, *pred*, and *iszero*, and closures, which we write in the form $<$ `fn x => e3`, ***env*** $>$.

The function *env* is an assignment of values to identifiers. This makes it quite different from the `E` above which assigns types to identifiers. Anything not matching a rule returns *error*.

The following axioms and rules define the semantics of type PCF. For (`e`, *env*) $\Rightarrow v$ to be defined, we require that domain(*env*) contain all free variables of `e`.

id $\quad$ (`id`, ***env***) $\Rightarrow$ ***env***(`id`)

int $\quad$ (`n`, ***env***) $\Rightarrow$ ***n, for n an integer***

true $\quad$ (`true`, ***env***) $\Rightarrow$ ***true***

$false$ $\quad (\texttt{false}, \mathit{env}) \Rightarrow \mathit{false}$

$succ$ $\quad (\texttt{succ}, \mathit{env}) \Rightarrow \mathit{succ}$

$pred$ $\quad (\texttt{pred}, \mathit{env}) \Rightarrow \mathit{pred}$

$iszero$ $\quad (\texttt{iszero}, \mathit{env}) \Rightarrow \mathit{iszero}$

$cond1$ $\quad \dfrac{(\texttt{b}, \mathit{env}) \Rightarrow \mathit{true} \qquad (\texttt{e1}, \mathit{env}) \Rightarrow v}{(\texttt{if b then e1 else e2, } \mathit{env}) \Rightarrow v}$

$cond2$ $\quad \dfrac{(\texttt{b}, \mathit{env}) \Rightarrow \mathit{false} \qquad (e2, \mathit{env}) \Rightarrow v}{(\texttt{if b then e1 else e2}, \mathit{env}) \Rightarrow v}$

$succAp$ $\quad \dfrac{(\texttt{e1}, \mathit{env}) \Rightarrow \mathit{succ} \qquad (\texttt{e2}, \mathit{env}) \Rightarrow n}{(\texttt{(e1 e2)}, \mathit{env}) \Rightarrow \mathit{(n+1)}}$

$predAp1$ $\quad \dfrac{(\texttt{e1}, \mathit{env}) \Rightarrow \mathit{pred} \qquad (\texttt{e2}, \mathit{env}) \Rightarrow \mathit{0}}{(\texttt{(e1 e2)}, \mathit{env}) \Rightarrow \mathit{0}}$

$predAp2$ $\quad \dfrac{(\texttt{e1}, \mathit{env}) \Rightarrow \mathit{pred} \qquad (\texttt{e2}, \mathit{env}) \Rightarrow n}{(\texttt{(e1 e2)}, \mathit{env}) \Rightarrow \mathit{(n-1)}}$

$iszero1$ $\quad \dfrac{(\texttt{e1}, \mathit{env}) \Rightarrow \mathit{iszero} \qquad \texttt{e2}, \mathit{env}) \Rightarrow \mathit{0}}{(\texttt{(e1 e2)}, \mathit{env}) \Rightarrow \mathit{true}}$

$iszero2$ $\quad \dfrac{(\texttt{e1}, \mathit{env}) \Rightarrow \mathit{iszero} \qquad (\texttt{e2}, \mathit{env}) \Rightarrow \mathit{n+1}}{(\texttt{(e1 e2)}, \mathit{env}) \Rightarrow \mathit{(n-1)}}$

$userFunc$ $\quad ((\texttt{fn (x:T) => e}), \mathit{env}) \Rightarrow < \texttt{fn (x:T) => e}, \mathit{env} >$

$userAp$ $\quad \dfrac{(\texttt{e1}, \mathit{env}) \Rightarrow < \texttt{fn (x:T) => e3}, \mathit{env}' > \qquad (\texttt{e2}, \mathit{env}) \Rightarrow \mathit{v2} \qquad (\texttt{e3}, \mathit{env}'[\mathit{v2}/\texttt{x}]) \Rightarrow v}{(\texttt{(e1 e2)}, \mathit{env}) \Rightarrow v}$

By our restrictions on pairs (e, *env*) to be involved in reductions, the closure, $<$ `fn x => e`$, \mathit{env} >$ resulting from the applicatin of $userFunc$ has the property that all free variables of `fn x => e` are contained in the domain of *env*.

5 Typing Rules for Values

Because we wish to show that types are preserved under computation, we will need to calculate the types of the values obtained from the semantics rules. It is important to note that these values have no free variables. That is obviously correct for numbers and the values true and false. Closures, which include the syntax for the function definitions are the only ones that might be able to have free variables. However, the environment paired with the closure is required to have interpretations of all of the free variables in the function definition, so all free variables in the function body are bound in the environment.

The first rules are trivial:

$$\mid\!-_v n : \texttt{Int}, \text{for n an integer}$$

$$\mid\!-_v \mathit{true} : \texttt{Bool}$$

$$\mid\!-_v \mathit{false} : \texttt{Bool}$$

$$\mid\!-_v \mathit{succ} : \texttt{Int} \rightarrow \texttt{Int}$$

$$\mid\!-_v \mathit{pred} : \texttt{Int} \rightarrow \texttt{Int}$$

$$\mid\!-_v \mathit{iszero} : \texttt{Int} \rightarrow \texttt{Bool}$$

In order to state the rule for closures we require the definition of when a regular environment is compatible with a type environment.

Definition: We say that a typing environment `E` and a regular environment *env* are *compatible* iff domain(`E`) = domain(*env*), and for all `x` in domain(*env*), $\mid\!-_v$ *env*(`x`): `E(x)`.

That is, *env* is compatible with `E` iff they have the same domain, and the value for an identifier obtained from *env* has the same type given that identifier by `E`.

$$\frac{\texttt{E'} \mid\!- \texttt{fn (x:T) => e: T} \rightarrow \texttt{U}, \text{ for the } \texttt{E'} \text{ such that } \texttt{E'} \text{ and } \mathit{env} \text{ are compatible}}{\mid\!-_v < \texttt{fn (x:T) => e}, \mathit{env} >: \texttt{T} \rightarrow \texttt{U}}$$

While this may look a bit confusing, the intuition should be clear. To type a closure, just create a type environment that is compatible with the environment *env* in the closure and type check with respect to that type environment to get the type of the closure.

6 Type safety

We expect that the type checking rules and semantic rules should have some connection. In particular, if we can prove that a term has a particular type, then we expect that if we evaluate the term then its value should have the same type predicted by the type checker. After all, if that fails to be the case then what is the purpose of the type system!

Theorem 1 (Subject Reduction) *Let* `E` *be a type environment assigning types to identifiers and let env be a run-time environment assigning values to identifiers such that* `E` *and env are compatible.*

Let e be a term of typed PCF. If `E |- e: T` *and* `(e,env)` $\Rightarrow$ `v`*, then* $\mid\!-_v$ *v*: `T`.

Proof The proof is by induction on the derivation of the typing of `e`. I.e. we are allowed to assume the theorem holds for all e' such that the proof of `E |- e: T` involves an earlier typing of `e'`.
Identifiers: Let `e` be an identifier, `id`, and suppose `E |- id: T` because `E(x) = T`. By assumption, $\emptyset$ `|- env(x): T`, and we are done.
Numbers: Let `e` be a number, `n`. Thus `E |- n: Int`. By the definition of semantics, (`n`,*env*) $\Rightarrow$ *n* and $|-_v$ *n*: `Int`, as desired.
true, false, succ, pred, iszero: The proof is similar to that for numbers as they simply evaluate to themselves and contain no identifiers.
Conditional: Suppose `E |- if b then e1 else e2: T` because `E |- b: boolean`, `E|-e1: T`, and `E|-e2: T`. There are two cases depending on the evaluation of `b`.

- Suppose that (`b`, *env*) $\Rightarrow$ *true*. Then if (`e1`,*env*) $\Rightarrow$ *v*, it follows that (`if b then e1 else e2`,*env*) $\Rightarrow$ `v`.

  By induction and `E|-e1: T`, it follows that $|-_v$ *v*: `T`, which is all we need.

- The case for (`b`, *env*) $\Rightarrow$ *false* is similar.

Applications: Suppose that e is `(e1 e2)` and `E |- (e1 e2): U` because `E |- e1: T` $\rightarrow$ `U` and `E |- e2: T`. Suppose also that (`e1`, *env*) $\Rightarrow$ *f* and (`e2`, *env*) $\Rightarrow$ *v2*. By induction, $|-_v$ *f*: `T` $\rightarrow$ `U` and $|-_v$ *v2*: `T`.
The rest of the proof of that the final value has type `U` depends on the actual value of f.

- *f* is *succ*. Then `T = U = Int`. Because $|-_v$ *v*: `Int`, *v* must be an integer. By the semantics rule *succAp*, the result is *v+1*, another integer, which has type `Int`, as desired as `E |- (e1 e2): Int`.

- *f* is *pred*. Like the case for *succ*.

- *f* is *iszero*. Like the case for *succ*.

- *f* is a closure, < `fn (x:T) => e3`,*env'* >. By induction, $|-_v$ < `fn (x:T) => e3`,*env'* >: `T` $\rightarrow$ `U` (note that the `T` in the closure must match the `T` in the typing by the typing rule for closures). Because there is only one way to type a closure, there must be an `E'` such that `E'` and *env'* are compatible and `E' |- fn (x:T) => e3: T` $\rightarrow$ `U`.

  To complete the evaluation of `(e1 e2)`, via rule *userAp*, suppose (`e3`,*env'*[*v2*/`x`]) $\Rightarrow$ *v*. We must show that $|-_v$ *v*: `U`.

  Because `E' |- fn (x:T) => e3: T` $\rightarrow$ `U`, it must have been the case that `E'[x:T] |- e3: U`. However, because `E'` and `env'` are compatible, so are `E'[x:T]` and *env'*[*v2*/`x`] (recall that $|-_v$ *v2*: `T`). Therefore by the induction hypothesis and (`e3`,*env'*[*v2*/`x`]) $\Rightarrow$ *v*, it follows that $|-_v$ *v*: `U`, as desired.

■

Hopefully this gives you a sense as to how these type-safety results work. Again, we normally also need to prove a progress theorem that shows one never gets to a state where no rule applies. To do this with a natural semantics we must fill in lots of error terms to make sure there is always something to do – and make sure that error terms are never in any of the types that we care about. The most popular alternative operational semantics, structured operational semantics, makes that easier. However, we tend to prefer this sort of "big-step semantics" to the SOS "small-step semantics".