Refinement ordering, approximation, and infinite data structures

COMP 150 — Dataflow

February 2, 2010

Refinement

The construction of infinite data structures is based on the refinement ordering $a \sqsubseteq b$, which can be pronounced:

- “$a$ approximates $b$”
- “$b$ is at least as evaluated as $a$”
- “$b$ is at least as well defined as $a$”

The refinement ordering is a partial order with least element $\bot$, pronounced “bottom.” We’re going to be sloppy about the meaning of the bottom element, seeing it variously as

- Undefined
- Not evaluated yet
- Divergent (infinite loop)

Infinite data structures work only in a complete partial order, i.e., every infinite ascending chain $v_1, v_2, v_3, \ldots$ has a least upper bound, which is written $\bigsqcup \{v_i\}$. Under this assumption, the values are

$$ v \Rightarrow \bot $$

$$ \mid C_n v_1 \cdots v_n \quad C_n \text{ is a value constructor of arity } n; \text{ this application is fully saturated} $$

$$ \mid \bigsqcup \{v_i\} \quad \text{where } \{v_i\} \text{ is an infinite ascending chain} $$

$$ \mid \lambda x.e \quad \text{functions are values} $$

The value constructors cover a multitude of sins:

- An integer literal can be treated as a nullary value constructor.
- Value constructors are always fully saturated, never partially applied. A partially applied value constructor can be turned into a value by eta-expanding it with $\lambda$-terms.

If we ignore functions, we can develop a purely syntactic theory of refinement:

<table>
<thead>
<tr>
<th>BOTTOM</th>
<th>REFLEXIVITY</th>
<th>MORPHISM</th>
<th>LIMITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot \sqsubseteq v$</td>
<td>$v \sqsubseteq v$</td>
<td>$v_i \sqsubseteq v'_i, 1 \leq i \leq n \Rightarrow C_n v_1 \cdots v_n \sqsubseteq C_n v'_1 \cdots v'_n$</td>
<td>$v \sqsubseteq \bigsqcup {v_i}$</td>
</tr>
</tbody>
</table>

To show that $\sqsubseteq$ is a partial order, we should show that it is transitive and antisymmetric. This could be done metatheoretically by induction on the height of the proofs. The key step is that a constructor applied to equal values produces equal values.
Functions

I don’t know how to develop a syntactic theory of refinement for functions. The refinement relation on functions is defined pointwise: \( f \sqsubseteq g \) if \( g \) is defined everywhere \( f \) is defined (that is \( g(x) = \bot \Rightarrow f(x) = \bot \)) and if \( g \) refines \( f \) pointwise on their common domain: \( \forall x (f(x) \sqsubseteq g(x)) \).

Well-behaved functions

Terms in the lambda calculus (and in Haskell) stand for functions that are in some technical sense well behaved. The technicalities are that the function must be monotonic and continuous:

- **monotonic**: \( \forall x, y, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \)
- **continuous**: \( f(\sqcup \{ a_i \}) = \sqcup \{ f(a_i) \} \)

We define \( A \rightarrow B \) as the set of total, monotonic, continuous functions from \( A \) to \( B \). (Totality may be achieved by mapping to \( \bot \).) This set is also a complete partial order.

Definition of values by recursion equations

You have seen in class that any recursive equation can be converted to a function that maps from one approximation to the next. If this function is monotonic and continuous, it has a fixed point, and in the value domain we take the least fixed point as the desired solution to the recursion equation.

We prove the existence of a fixed point constructively:

- \( \bot \sqsubseteq f(\bot) \) because \( \bot \sqsubseteq x \) for all \( x \)
- By previous, monotonicity, \( f(\bot) \sqsubseteq f(f(\bot)) \) (Apply \( f \) to both sides)
- \( \bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq f(f(f(\bot))) \sqsubseteq \cdots \) is an ascending chain.
- Then \( F = \sqcup \{ f(i)(\bot) \} \) is the least fixed point of \( f \) (exploit continuity):
  \[
  f(F) = f(\sqcup \{ f(i)(\bot) \}) = \sqcup \{ f(f(i)(\bot)) \} = \sqcup \{ f(i)(\bot) \} = F
  \]

Exercises

1. **Currying.** Show that \( A \times B \rightarrow C \simeq A \rightarrow (B \rightarrow C) \). Give an isomorphism explicitly in Haskell and get the compiler to type-check it.

2. Given the recursion equation \( o = 1 : o \),
   (a) Write the function such that a fixed point of the function is a solution to the recursion equation.
   (b) Prove that the function is monotonic.
   (c) Tell us what you think the least fixed point is.

3. Given the recursion equation \( u = u \),
   (a) Write the function such that a fixed point of the function is a solution to the recursion equation.
   (b) Prove that the function is monotonic.
   (c) Tell us what you think the least fixed point is.

4. See if you can think of a function that is not monotonic. **Hint:** You can’t write such a function in Haskell; you have to think about the world of mathematics.