Recursive Function Definition over Coinductive Types

John Matthews

Oregon Graduate Institute,
P.O. Box 91000, Portland OR 97291-1000, USA
johnm@cse.ogi.edu
http://www.cse.ogi.edu/~johnm

Abstract. Using the notions of *unique fixed point*, *converging equivalence relation*, and *contracting function*, we generalize the technique of well-founded recursion. We are able to define functions in the Isabelle theorem prover that recursively call themselves an infinite number of times. In particular, we can easily define recursive functions that operate over coinductively-defined types, such as infinite lists. Previously in Isabelle such functions could only be defined corecursively, or had to operate over types containing "extra" bottom-elements. We conclude the paper by showing that the functions for filtering and flattening infinite lists have simple recursive definitions.

1 Well-founded recursion

Rather than specify recursive functions by possibly inconsistent axioms, several higher order logic (HOL) theorem provers[3, 9, 12] provide well-founded recursive function definition packages, where new functions can be defined conservatively. Recursive functions are defined by giving a series of pattern matching reduction rules, and a well-founded relation.

For example, the *map* function applies a function f pointwise to each element of a finite list. This function can be defined using well-founded recursion:

$$map :: (\alpha \to \beta) \to \alpha\ list \to \beta\ list$$

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ (x\#xs) &= (f\ x)\ \#\ (map\ f\ xs) \end{aligned}$$

The first rule states that *map* applied to the empty list, denoted by [], is equal to the empty list. The second rule states that *map* applied to a list constructed out of the head element x and tail list xs, denoted by $x\#xs$, is equal to the list formed by applying f to x and *map* f to xs recursively.

To define a function using well-founded recursion, the user must also supply a *well-founded relation* on one of the function's arguments[1]. A well-founded

[1] Some well-founded recursion packages only allow single-argument functions to be defined. In this case one can gain the effect of multi-argument curried functions by tupling.

relation $(<)$ is a relation with the property that there exists no infinite sequence of elements $x_1, x_2, x_3, x_4, \ldots$ such that

$$\ldots < x_4 < x_3 < x_2 < x_1$$

For each reduction rule, the recursive definition package checks that every recursive call on the right-hand side of the rule is applied to a smaller argument than on the left-hand side, according to the user supplied well-founded relation.

In the case of *map*, we can supply the well-founded relation

$$xs < ys \equiv \mathit{length}\ xs < \mathit{length}\ ys$$

which is true when the number of elements in the relation's left-hand list argument is less than the number of elements in the relation's right-hand argument. The definition of *map* contains only one recursive rule, and it is easy to prove that the *xs* argument of the recursive call of *map* is smaller than the $(x\#xs)$ argument on the left-hand side of the rule, according to this relation. In general, well-founded relations ensure that there are no infinite chains of nested recursive calls.

2 Coinductive types and corecursive functions

Although well-founded recursion is a useful definition technique, there are many recursive definitions that fall outside its scope. For instance, there is a non-inductive type of *lazy lists* in the Isabelle[9] theorem prover, denoted by $\alpha\ \mathit{llist}$, that is the set of all finite and infinite lists of type α. The function *lmap* over this type is uniquely specified by the following recursive equations[2]:

$$\begin{aligned} &\mathit{lmap}\ f\ [] &&= [] \\ &\mathit{lmap}\ f\ (x\#xs) &&= (f\ x)\ \#\ (\mathit{lmap}\ f\ xs) \end{aligned}$$

One cannot define *lmap* using well-founded recursion since the length of an infinite list does not decrease when you take its tail. In fact, the expression $\mathit{lmap}\ f\ (x_1\ \#\ x_2\ \#\ x_3\ \#\ \ldots)$ can be unfolded using the above rules to an infinite chain of recursive calls:

$$\begin{aligned} &\quad \mathit{lmap}\ f\ (x_1\ \#\ x_2\ \#\ x_3\ \#\ ...) \\ &= \\ &\quad (f\ x_1)\ \#\ (\mathit{lmap}\ f\ (x_2\ \#\ x_3\ \#\ ...)) \\ &= \\ &\quad (f\ x_1)\ \#\ (f\ x_2)\ \#\ (\mathit{lmap}\ f\ (x_3\ \#\ ...)) \\ &= \\ &\quad (f\ x_1)\ \#\ (f\ x_2)\ \#\ (f\ x_3)\ \#\ (\mathit{lmap}\ f\ (...)) \\ &= \\ &\quad ... \end{aligned}$$

[2] Isabelle uses a different syntax for lazy lists than for finite lists. In this paper we use the same syntax for both types.

Defining functions corecursively

The α *llist* type is an example of a coinductive type. Although there is no general induction principle for coinductive types, one can use principles of coinduction to show that two coinductive values are equal, and one can build coinductive values using *corecursion*.

In Isabelle's theory of lazy lists[10], for instance, one builds potentially infinite lists through the *llist_corec* operator, which has type $\beta \to (\beta \to \mathit{unit} + (\alpha * \beta)) \to (\alpha\ \mathit{llist})$. The *llist_corec* operator uniquely satisfies the following recursion equation:

$$\mathit{llist\_corec}\ b\ g = \begin{cases} [], & \text{if } g\ b = \mathrm{Inl}\ () \\ (x \mathbin{\#} (\mathit{llist\_corec}\ b'\ g)), & \text{if } g\ b = \mathrm{Inr}\ (x, b') \end{cases}$$

The *llist_corec* operator takes as arguments an initial value b and a function g. When g is applied to b, it either returns $\mathrm{Inl}\ ()$, indicating that the result list should be empty, or the value $\mathrm{Inr}\ (x, b')$, where x represents the first element of the result list, and b' represents the new initial value to build the rest of the list from. Function g is called iteratively in this fashion, constructing a potentially infinite list.

Using *llist_corec*, we can define *lmap* corecursively as follows:

$$\begin{array}{l} \mathit{lmap}\ \ f\ \mathit{xs} \equiv \mathit{llist\_corec}\ \mathit{xs}\ (\mathit{map\_head}\ f) \\ \text{where} \\ \qquad \begin{array}{lll} \mathit{map\_head} & :: & (\alpha \to \beta) \to \alpha\ \mathit{llist}\ \to (\mathit{unit} + (\beta * \alpha\ \mathit{llist})) \\ \mathit{map\_head}\ f\ \mathit{xs} & \equiv & \text{case}\ \mathit{xs}\ \text{of} \\ & & \quad\ \ [] \qquad\quad \Rightarrow \mathrm{Inl}\ () \\ & & \mid\ (x\#\mathit{xs}') \ \Rightarrow \mathrm{Inr}\ (f\ x, \mathit{xs}') \end{array} \end{array}$$

One can then prove by coinduction that this definition satisfies *lmap*'s recursive equations. Needless to say, this is not the most intuitive specification of *lmap*, and most people would prefer to specify such functions using recursion, if possible. In the remainder of the paper we will present a framework for defining functions such as *lmap* recursively.

3 Solving recursive equations

The basic steps required in this framework to show that a set of recursive equations is well defined are as follows:

- Express the recursive equations as a fixed point of a functional F.
- Show that for any two different potential solutions supplied to F, F maps them to two potential solutions that are closer together, in a suitable sense.
- Invoke the main result (Sect. 4.3) to show that the above property of F is sufficient to guarantee that there is a unique solution to the original set of recursive equations.

In this section we deal with the first step.

3.1 Unique fixed points

We convert a system of pattern matching recursive equations into a functional form by employing a standard technique from domain theory[4, 15]. We start by recasting the equations as a single recursive equation using argument destructors or nested case-expressions. For example, the recursive equations defining the *lmap* function are equivalent to the following single recursive equation:

$$\begin{aligned} &\mathit{lmap}\ f\ l = \text{case } l \text{ of} \\ &\qquad\qquad [] \qquad\quad \Rightarrow [] \\ &\qquad\quad\ \mid (x\#xs) \quad \Rightarrow (f\ x)\ \#\ (\mathit{lmap}\ f\ xs) \end{aligned}$$

Given f, we can reify this pattern of recursion into a non-recursive functional F of type $(\alpha\ \mathit{llist} \to \beta\ \mathit{llist}) \to (\alpha\ \mathit{llist} \to \beta\ \mathit{llist})$ that takes a function parameter *lmap_f*:

$$\begin{aligned} &F\ \mathit{lmap\_f} = \lambda l\,.\ \text{case } l \text{ of} \\ &\qquad\qquad\qquad [] \qquad\quad \Rightarrow [] \\ &\qquad\qquad\quad\ \mid (x\#xs) \quad \Rightarrow (f\ x)\ \#\ (\mathit{lmap\_f}\ xs). \end{aligned}$$

Using the recursive equations for *lmap*, it is easy to show that $\mathit{lmap}\ f = F\ (\mathit{lmap}\ f)$. The value $\mathit{lmap}\ f$ is called a *fixed point* of F. In general, an element x of type α is a fixed point of a function g of type $\alpha \to \alpha$ if $x = g\,x$. A function may have many fixed points, or none at all. Considering g as a functional representation of a system of recursive equations, each fixed point of g represents a valid solution to the system. If the function g has exactly one fixed point x, then we can think of g as *defining* the value x. We use Hilbert's description operator (ε) to formalize this notion in HOL:

$$\begin{aligned} &\mathrm{fix} :: (\alpha \to \alpha) \to \alpha \\ &\mathrm{fix}\, g \equiv \varepsilon x\,.\ x = g\,x \wedge (\forall\, y\, z\,.\, y = g\,y \wedge z = g\,z \longrightarrow y = z) \end{aligned}$$

The expression $\mathrm{fix}\, g$ represents the unique fixed point of g, when one exists. If g does not have a *unique* fixed point, then $\mathrm{fix}\, g$ denotes an arbitrary value.

3.2 Properties of unique fixed points

As an aside, several nice properties hold when one can establish that a system of recursive equations has a unique solution. For example, unique fixed points can sometimes "absorb" functions applied to other fixed points.

Lemma 1 *Given functions $F : \alpha \to \alpha$, $G : \beta \to \beta$, $f : \alpha \to \beta$, and value $x : \alpha$, such that x is a (not necessarily unique) fixed point of F, G has a unique fixed point, and $f \circ F = G \circ f$, then $f\,x = \mathrm{fix}\, G$.*

Unique fixed points can also be "rotated", in the following sense:

Lemma 2 *If the composition of two functions $g : \beta \to \alpha$ and $h : \alpha \to \beta$ has a unique fixed point $\mathrm{fix}\,(g \circ h)$, then $h \circ g$ also has a unique fixed point, and $\mathrm{fix}\,(g \circ h) = g\,(\mathrm{fix}\,(h \circ g))$.*

Although we will not use Lemma 1 or Lemma 2 in the remainder of the paper, lemmas such as these are useful for manipulating systems of recursive equations as objects in their own right.

4 Converging equivalence relations and contracting functions

While unique fixed points are a useful definition mechanism, it can be difficult to show that they exist for a given function. A direct proof usually involves constructing an explicit fixed point witness using other definition techniques, such as corecursion or well-founded recursion. Little effort seems to be saved.

We propose an alternative proof technique, based on concepts from domain theory[4, 15] and topology[1, 11] where one builds a collection of ever-closer approximations to the desired fixed point, and shows that the limit of these approximations exists, is a fixed point of the function under consideration, and is unique. The approximation process can be parameterized to some extent, and reused across multiple definitions that are "similar" enough. Furthermore these parameterized approximations can be composed hierarchically, yielding more powerful approximation techniques.

4.1 Converging equivalence relations

To make the notion of approximation precise, we need a way of stating how "close" two potential approximations are to each other. One approach would be to define a suitable metric space[1] and use the corresponding distance function, which returns either a rational or real number, given any two elements in the domain of the metric space. However, proving that a series of approximations converges to a limit point often requires one to reason about exponentiation and division over a theory of rationals or reals. An alternative way to measure "closeness", which we call a *converging equivalence relation* (CER), instead only involves reasoning about well-founded sets, such as the set of natural numbers, or the set of finite lists. In many cases we can prove a unique fixed point exists by performing a simple induction over the natural numbers, something which all of the current HOL theorem provers support well.

A converging equivalence relation consists of:

- A type ρ, called the *resolution space*
- A type τ, called the *target space*
- A well-founded, transitive relation $(<)$ over type ρ, called a *resolution ordering*
- A three-argument predicate $(\approx)$ of type $(\rho \to \tau \to \tau \to \mathit{bool})$, called an *indexed equivalence relation.* Given an element i of type ρ, and two elements x and y of type τ, we denote the application of $(\approx)$ to i, x and y as $(x \overset{i}{\approx} y)$, and if this value is true, then we say that x and y are *equivalent at resolution i.*

The resolution ordering ($<$) and indexed equivalence relation ($\approx$) must satisfy the properties in Fig. 1, for arbitrary $i, i' : \rho$; $x, y, z : \tau$; and $f : \rho \to \tau$. Axioms (1), (2), and (3) state that ($\approx$) must be an equivalence relation at each resolution i. Axiom (4) states that if a resolution i has no lower resolutions, then ($\approx$) treats all target elements as equivalent at that resolution. Such resolutions are called *minimal.* There is always at least one minimal resolution (and perhaps more than one), since ($<$) is well-founded. Axiom (5) states that if two elements are equivalent at a particular resolution, then they are equivalent at all lower resolutions. Thus higher resolutions impose finer-grained, but compatible, partitions of the target space than lower resolutions do. Although no particular resolution may distinguish all elements, (6) states that if two elements are equivalent at all resolutions, then they are in fact equal.

$$x \overset{i}{\approx} x \tag{1}$$

$$x \overset{i}{\approx} y \longrightarrow y \overset{i}{\approx} x \tag{2}$$

$$x \overset{i}{\approx} y \wedge y \overset{i}{\approx} z \longrightarrow x \overset{i}{\approx} z \tag{3}$$

$$(\forall j \,.\, \neg(j < i)) \longrightarrow x \overset{i}{\approx} y \tag{4}$$

$$x \overset{i'}{\approx} y \wedge i < i' \longrightarrow x \overset{i}{\approx} y \tag{5}$$

$$(\forall j \,.\, x \overset{j}{\approx} y) \longrightarrow x = y \tag{6}$$

$$(\forall j, j' \,.\, j < j' < i \longrightarrow (f\,j) \overset{j}{\approx} (f\,j')) \longrightarrow (\exists z \,.\, \forall j < i \,.\, z \overset{j}{\approx} (f\,j)) \tag{7}$$

$$(\forall j, j' \,.\, j < j' \longrightarrow (f\,j) \overset{j}{\approx} (f\,j')) \longrightarrow (\exists z \,.\, \forall j \,.\, z \overset{j}{\approx} (f\,j)) \tag{8}$$

Fig. 1. The CER axioms. Each of these axioms must hold for arbitrary i, x, y, and f.

Axioms (7) and (8) deal with "limits" of approximations. First some terminology: a function $f : \rho \to \tau$ from the space of resolutions to the target space of elements is called an *approximation map.* An approximation map f is *convergent up to resolution* i if for all resolutions j and j' such that $j < j' < i$, then $(f\,j)$ is equivalent at resolution j to $(f\,j')$. Note that it is possible for $(f\,i)$ itself not to be equivalent to any of the lower-resolution $(f\,j)$'s. An approximation map f is *globally convergent* if for all resolutions j and j' such that $j < j'$, then $(f\,j) \overset{j}{\approx} (f\,j')$.

Axiom (7) states that if f is locally convergent up to resolution i, then there exists a limit-like element z that is equivalent at each resolution $j < i$ to the corresponding $(f\,j)$ approximation (there may be multiple such elements). Axiom (8) states that if f is globally convergent, then there exists a limit element z that is equivalent to each approximation $(f\,j)$ at resolution j.

4.2 Examples of converging equivalence relations

Discrete CER The simplest useful CER has as a resolution space a two-element type containing the values $\bot$ and $\top$, with $(\bot < \top)$, and a target space τ with $(\approx)$ defined such that $(x \overset{\bot}{\approx} y) \equiv \mathit{True}$, and $(x \overset{\top}{\approx} y) \equiv (x = y)$. Axioms (1) through (6) are easy to verify. Axiom (7) holds for any element. The limit element satisfying (8) is $f\,\top$.

Lazy list CER We can construct a converging equivalence equation for comparing coinductive lists by comparing the first i elements of two lazy lists l_1 and l_2 at a given resolution i. To perform the comparison, we make use of the *ltake* function, with type $\mathit{nat} \to \alpha\,\mathit{llist} \to \alpha\,\mathit{list}$. The expression $(\mathit{ltake}\,n\,\mathit{xs})$ returns a finite list consisting of the first n elements of xs. If xs has fewer than n elements, then *ltake* returns the whole of xs. The *ltake* function can be defined by well-founded recursion on its numeric argument with the following recursive equations:

$$\begin{array}{llll} \mathit{ltake} & 0 & \mathit{xs} & = [] \\ \mathit{ltake} & (n+1) & [] & = [] \\ \mathit{ltake} & (n+1) & (x \# \mathit{xs}) & = x \# (\mathit{ltake}\,n\,\mathit{xs}) \end{array}$$

We then define the lazy list CER with the natural numbers as the resolution space, $(\alpha\,\mathit{llist})$ as the target space, the usual ordering on the natural numbers for $(<)$, and $(\approx)$ defined as follows:

$$\mathit{xs} \overset{i}{\approx} \mathit{ys} \equiv (\mathit{ltake}\,i\,\mathit{xs} = \mathit{ltake}\,i\,\mathit{ys}).$$

Axioms (1) through (3) hold trivially. The only minimal resolution in this CER is 0, and since $(\mathit{ltake}\,0\,\mathit{xs}) = []$, then (4) holds. If two lazy lists are equal up to the first i positions, then they are equal up to any $i' < i$ position, so (5) holds. Axiom (6) reduces to the Take Lemma[10], which can be proved by coinduction.

Axioms (7) and (8) require us to construct appropriate limit elements, given an approximation map. Both limit elements can be constructed by a single function, which we call *llist_diag*. For a given approximation map f, the limit elements may be of infinite length, so we define *llist_diag* by corecursion, using *llist_corec*:

$$\mathit{llist\_diag}\ f \equiv \mathit{llist\_corec}\,0\,(\mathit{nthElem}\,f)$$
where
$$\mathit{nthElem}\,f\,n \equiv \begin{cases} \mathrm{Inl}\,(), & \text{if } \mathit{ldrop}\,n\,(f(n+1)) = [] \\ \mathrm{Inr}\,(x, n+1), & \text{if } \mathit{ldrop}\,n\,(f(n+1)) = (x \# \mathit{xs}) \end{cases}$$

The helper function *nthElem* uses the *ldrop* function on lazy lists. The *ldrop* function has type $\mathit{nat} \to (\alpha\,\mathit{llist}) \to (\alpha\,\mathit{llist})$, and $(\mathit{ldrop}\,i\,\mathit{xs})$ removes the first i elements from xs, returning the remainder. Like *ltake*, it is defined by well-founded recursion on its numeric argument:

$$\begin{array}{llll} \mathit{ldrop} & 0 & \mathit{xs} & = \mathit{xs} \\ \mathit{ldrop} & (n+1) & [] & = [] \\ \mathit{ldrop} & (n+1) & (x \# \mathit{xs}) & = \mathit{ldrop}\,n\,\mathit{xs} \end{array}$$

The overall action of *llist_diag* is to construct a so-called *diagonal list* from the approximation map f, where the n^{th} element of the result list is drawn from the n^{th} element of approximation $f\,(n+1)$, if the n^{th} element exists. If the n^{th} element does not exist (i.e., the length of $f\,(n+1)$ is less than n), then the result list is terminated at that point.

It turns out that for any CER whose $(<)$ relation is the less-than ordering on the natural numbers, the following property implies both (7) and (8):

$$\forall f\,.\,(\forall i\,.\,(f\,i) \overset{i}{\approx} (f\,(i+1))) \longrightarrow (\exists x\,.\,\forall i\,.\,x \overset{i}{\approx} (f\,i)).$$

With some work, one can show that this property holds for the lazy list CER by supplying *llist_diag* f as the existential witness element for x.

4.3 Contracting functions

In the theory of metric spaces, a *contracting function* is a function F such that for any two points x and y, $F\,x$ is closer to $F\,y$ than x is to y, given a suitable distance function. Banach's theorem states that all contracting functions over suitable metric spaces have unique fixed points. We can define an analogous notion over a CER:

Definition 1 *A function F is* contracting *over a CER given by $(<)$ and $(\approx)$ if for all resolutions i and target elements x and y,*

$$(\forall i' < i\,.\,x \overset{i'}{\approx} y) \longrightarrow (F\,x) \overset{i}{\approx} (F\,y).$$

Intuitively a function is contracting if, given two elements x and y that are close enough together at all lower resolutions $i' < i$ to satisfy the CER, but are potentially too far away at resolution i, then F maps them to two elements that are now close enough at resolution i.

For example, the function $\mathit{consZero}\ \mathit{xs} \equiv (0\#\mathit{xs})$ is contracting over the lazy list CER, since given any i and two lazy lists xs and ys,

$$(\forall i' < i\,.\,\mathit{ltake}\,i'\,\mathit{xs} = \mathit{ltake}\,i'\,\mathit{ys}) \longrightarrow \mathit{ltake}\,i\,(\mathit{consZero}\ \mathit{xs}) = \mathit{ltake}\,i\,(\mathit{consZero}\ \mathit{ys}).$$

The main result of this paper is as follows:

Theorem *A contracting function F over a CER has a unique fixed point.*

The proof is discussed in Sect. 7. For now, we would like to apply this theorem to define some simple recursive functions over lazy lists.

4.4 Recursive definitions over coinductive lists

To begin with, we can simplify the definition of a contracting function F over a CER when the $(<)$ relation of that CER is the less-than relation over the natural numbers. In this case, Definition 1 reduces to

$$\forall\,i\,x\,y\,.\,x \overset{i}{\approx} y \longrightarrow (F\,x) \overset{i+1}{\approx} (F\,y). \tag{9}$$

Specializing this formula for the lazy list CER, we have that F is contracting on lazy lists if

$$\forall\, i\, x\, y\, .\, \mathit{ltake}\, i\, x = \mathit{ltake}\, i\, y \longrightarrow \mathit{ltake}\,(i+1)\,(F\, x) = \mathit{ltake}\,(i+1)\,(F\, y). \tag{10}$$

Defining iterates Let us establish that the following recursive equation, defined over x and f, has a unique solution, and is thus a definition:

$$\mathit{iterates} = (x \,\#\, (\mathit{lmap}\; f\; \mathit{iterates})) \tag{11}$$

This equation builds the infinite list $[x, f\,x, f\,(f\,x), \ldots]$. We first define the non-recursive functional F that characterizes this equation:

$$F\; \mathit{iterates}' \equiv (x \,\#\, (\mathit{lmap}\; f\; \mathit{iterates}')).$$

and then show that it is a contracting function. To do this we rely on (10), and assume we have two arbitrary lazy lists xs and ys such that $\mathit{ltake}\, i\, \mathit{xs} = \mathit{ltake}\, i\, \mathit{ys}$. We now need to show that $\mathit{ltake}\,(i+1)\,(F\,\mathit{xs}) = \mathit{ltake}\,(i+1)\,(F\,\mathit{ys})$. Using a process of equational simplification we are able to reduce the goal to the assumption, as follows:

$$\begin{aligned}
&\mathit{ltake}\,(i+1)\,(F\,\mathit{xs}) = \mathit{ltake}\,(i+1)\,(F\,\mathit{ys}) \\
\Leftrightarrow\; &\mathit{ltake}\,(i+1)\,(x \,\#\, (\mathit{lmap}\; f\; \mathit{xs})) = \mathit{ltake}\,(i+1)\,(x \,\#\, (\mathit{lmap}\; f\; \mathit{ys})) \\
\Leftrightarrow\; &\mathit{ltake}\, i\,(\mathit{lmap}\; f\; \mathit{xs}) = \mathit{ltake}\, i\,(\mathit{lmap}\; f\; \mathit{ys}) \\
\Leftarrow\; &\mathit{ltake}\, i\, \mathit{xs} = \mathit{ltake}\, i\, \mathit{ys}
\end{aligned}$$

The simplification relies on the following facts, each proved by induction on i:

$$\begin{gathered}
(\mathit{ltake}\,(i+1)\,(z \,\#\, \mathit{xs}) = \mathit{ltake}\,(i+1)\,(z \,\#\, \mathit{ys})) \Leftrightarrow (\mathit{ltake}\, i\, \mathit{xs}) = \mathit{ltake}\, i\, \mathit{ys}) \\
(\mathit{ltake}\, i\,(\mathit{lmap}\; f\; \mathit{xs}) = \mathit{ltake}\, i\,(\mathit{lmap}\; f\; \mathit{ys}) \Leftarrow (\mathit{ltake}\, i\, \mathit{xs} = \mathit{ltake}\, i\, \mathit{ys})
\end{gathered}$$

These facts illustrate a nice property of this proof: We did not have to expand the definitions of (#) or *lmap* during the simplification process, relying instead on an abstract characterization of their behavior with respect to *ltake*. This turns out to be the case for many functions, even recursive ones defined by contracting functions. In general we can often incrementally define recursive functions and prove properties about how they behave with respect to $(\approx)$, without having to expand the definitions of functions making up the body of the recursive definition.

5 Composing converging equivalence relations

The lazy list CER allows us to give recursive definitions of individual lazy lists, but we are often more interested in recursively defining functions that transform lazy lists. Fortunately, there are several *CER combinators* that allow us to build CERs over complex types, if we have CERs that operate on the corresponding atomic types.

Local and global limits When constructing a new CER C' out of an existing CER C, we usually have to show (7) and (8) hold for C' by invoking (7) and (8) for C, to create the necessary limit witness elements. To make this process explicit, we use Hilbert's description operator (ε) to create functions that return these witness elements[3], given an appropriate approximation mapping f:

$$\begin{aligned} \mathit{local\_limit} &:: (\rho \to \tau) \to \rho \to \tau \\ \mathit{local\_limit}\, f\, i &\equiv (\varepsilon z\,.\, \forall j < i\,.\, z \overset{j}{\approx} (f\, j)) \end{aligned} \tag{12}$$

$$\begin{aligned} \mathit{global\_limit} &:: (\rho \to \tau) \to \tau \\ \mathit{global\_limit}\, f &\equiv (\varepsilon z\,.\, \forall j\,.\, z \overset{j}{\approx} (f\, j)) \end{aligned} \tag{13}$$

We can use (7) and (8) to prove the basic properties we want *local_limit* and *global_limit* to have for any CER given by $(<)$ and $(\approx)$:

$$(\forall j, j'\,.\, j < j' < i \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \longrightarrow (\forall j < i\,.\, (\mathit{local\_limit}\, f\, i) \overset{j}{\approx} (f\, j))$$
$$(\forall j, j'\,.\, j < j' \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \longrightarrow (\forall j\,.\, (\mathit{global\_limit}\, f) \overset{j}{\approx} (f\, j))$$

Function-space CER The functions *local_limit* and *global_limit* allow us to concisely specify the limit elements of CER combinators. For example, given a CER C from resolution space ρ to target space τ given by $(<)$ and $(\approx)$, we can construct a new *function-space over* C CER with the same resolution ordering $(<)$, and a new indexed equivalence relation $(\approx')$ with type
$\rho \to (\sigma \to \tau) \to (\sigma \to \tau) \to \mathit{bool}$, defined as

$$g \overset{i}{\approx'} h \equiv \forall x\,.\, (g\, x) \overset{i}{\approx} (h\, x).$$

The limit elements satisfying (7) and (8) can be given as

$$\begin{aligned} \mathit{local\_limit\_fun}\, f\, i &\equiv (\lambda x\,.\, \mathit{local\_limit}\, (\lambda i\,.\, f\, i\, x)\, i) \\ \mathit{global\_limit\_fun}\, f &\equiv (\lambda x\,.\, \mathit{global\_limit}\, (\lambda i\,.\, f\, i\, x)) \end{aligned}$$

Given these limit-producing functions, is relatively easy to show that the function-space over C CER satisfies the CER axioms.

5.1 Defining recursive functions with the function-space CER

Defining lmap We can apply the function-space CER to define *lmap* recursively. The recursion equations for *lmap* are:

$$\begin{aligned} \mathit{lmap}\ f\ \ []\ &= [] \\ \mathit{lmap}\ f\ (x\#\mathit{xs}) &= (f\, x)\, \#\, (\mathit{lmap}\ f\ \mathit{xs}) \end{aligned}$$

[3] This is merely a convenience. The CER properties can be shown with a little more work in Isabelle using (7) and (8) directly.

We translate the equations into a non-recursive form (parameterized over f)

$$\begin{array}{l} F\ lmap' \equiv (\lambda xs\,.\ \text{case}\ xs\ \text{of} \\ \qquad\qquad [] \qquad\quad \Rightarrow [] \\ \qquad\quad \mid (y \,\#\, ys) \Rightarrow (f\,y) \,\#\, (lmap'\ ys)). \end{array}$$

We then need to show that $\text{fix}\,F$ is the unique fixed point of F by proving that F is a contracting function on the function-space over lazy lists CER. By (9) we must show for arbitrary resolution i and functions g and h, that $(g \overset{i}{\approx'} h \longrightarrow (F\,g) \overset{(i+1)}{\approx'} (F\,h))$. Expanding definitions, we obtain

$$\begin{array}{l} g \overset{i}{\approx'} h \longrightarrow (F\,g) \overset{(i+1)}{\approx'} (F\,h) \\ \Leftrightarrow (\forall\, xs\,.\, g\,xs \overset{i}{\approx} h\,xs) \longrightarrow (\forall\, xs\,.\,(F\,g\,xs) \overset{(i+1)}{\approx} (F\,h\,xs)) \\ \Leftrightarrow (\forall\, xs\,.\, ltake\,i\,(g\,xs) = ltake\,i\,(h\,xs)) \longrightarrow \\ \qquad (\forall\, xs\,.\, ltake\,(i+1)\,(F\,g\,xs) = ltake\,(i+1)\,(F\,h\,xs)). \end{array}$$

So, to prove F is contracting we take an arbitrary resolution i and two arbitrarily chosen functions g and h such that $(\forall\, xs\,.\, ltake\,i\,(g\,xs) = ltake\,i\,(h\,xs))$, and show for an arbitrary xs that $ltake\,(i+1)\,(F\,g\,xs) = ltake\,(i+1)\,(F\,h\,xs)$. There are two cases to consider:

case $xs = []$:

$$\begin{array}{ll} & ltake\,(i+1)\,(F\,g\,[]) = ltake\,(i+1)\,(F\,h\,[]) \\ \Leftrightarrow & ltake\,(i+1)\,[] = ltake\,(i+1)\,[] \\ \Leftrightarrow & \text{True}. \end{array}$$

case $xs = (y\#ys)$:

$$\begin{array}{ll} & ltake\,(i+1)\,(F\,g\,(y\#ys)) = ltake\,(i+1)\,(F\,h\,(y\#ys)) \\ \Leftrightarrow & ltake\,(i+1)\,((f\,y) \,\#\, (g\,ys)) = ltake\,(i+1)\,((f\,y) \,\#\, (h\,ys)) \\ \Leftrightarrow & ltake\,i\,(g\,ys) = ltake\,i\,(h\,ys) \\ \Leftrightarrow & \text{True}\ \{\text{by assumption}\}. \end{array}$$

Given the definition of F and basic lemmas about *ltake*, Isabelle's high-level simplification tactics allow the above proof to be carried out in two steps. The proof completes in about a second on a 266MHz Pentium II.

Defining lappend We can apply the function-space CER combinator repeatedly, to prove that multi-argument curried functions have unique fixed points. As a concrete example, the curried function *lappend* has type $\alpha\ llist \rightarrow \alpha\ llist \rightarrow \alpha\ llist$. It takes two lazy list arguments xs and ys and returns a new list consisting of the elements of xs followed by the elements of ys. The recursive equations for *lappend* are

$$\begin{array}{lll} lappend & [] & ys = ys \\ lappend & (x\#xs) & ys = (x \,\#\, lappend\ xs\ ys) \end{array}$$

To prove that these equations have a unique solution, we apply the function-space CER combinator to the lazy list CER to obtain a new CER C'. We then apply the function-space CER combinator again to C', obtaining a new CER C'' with the usual less-than relation on *nat* for $(<)$ and the following indexed equivalence relation $(\approx'')$:

$$g \overset{i}{\approx''} h \equiv (\forall\, xs\ ys\, .\, ltake\, i\, (g\, xs\ ys) = ltake\, i\, (h\, xs\ ys)).$$

Next, we convert the recursive equations for *lappend* into a non-recursive function F:

$$\begin{array}{lll} F\ lappend' \equiv (\lambda xs\ ys\, .\ \ \text{case}\ xs\ \text{of} & & \\ \quad\quad [] & \Rightarrow ys & \\ \quad | \ (x \,\#\, xs') & \Rightarrow (x \,\#\, (lappend'\ xs'\ ys))). & \end{array}$$

By (9) we must show for arbitrary resolution i and functions g and h, that

$$\begin{array}{l}(\forall\, xs\ ys\, .\, ltake\, i\, (g\, xs\ ys) = ltake\, i\, (h\, xs\ ys)) \longrightarrow \\ \quad (\forall\, xs\ ys\, .\, ltake\, (i+1)\, (F\, g\, xs\ ys) = ltake\, (i+1)\, (F\, h\, xs\ ys)).\end{array}$$

So we take arbitrary i, xs, and ys, and prove

$$ltake\, (i+1)\, (F\, g\, xs\ ys) = ltake\, (i+1)\, (F\, h\, xs\ ys)$$

assuming we have $(\forall\, xs\ ys\, .\, ltake\, i\, (g\, xs\ ys) = ltake\, i\, (h\, xs\ ys))$. There are two cases to consider, depending on whether xs is empty or not:

case $xs = []$:

$$\begin{array}{ll} & ltake\, (i+1)\, (F\, g\, []\ ys) = ltake\, (i+1)\, (F\, h\, []\ ys) \\ \Leftrightarrow & ltake\, (i+1)\, ys = ltake\, (i+1)\, ys \\ \Leftrightarrow & \text{True}. \end{array}$$

case $xs = (x\#xs')$:

$$\begin{array}{ll} & ltake\, (i+1)\, (F\, g\, (x\#xs')\ ys) = ltake\, (i+1)\, (F\, h\, (x\#xs')\ ys) \\ \Leftrightarrow & ltake\, (i+1)\, (x \,\#\, (g\, xs'\ ys)) = ltake\, (i+1)\, (x \,\#\, (h\, xs'\ ys)) \\ \Leftrightarrow & ltake\, i\, (g\, xs'\ ys) = ltake\, i\, (h\, xs'\ ys) \\ \Leftrightarrow & \text{True}\ \{\text{by assumption}\}. \end{array}$$

Thus we can conclude that *lappend* has a unique fixed point definition. We were able to carry out this proof in Isabelle in three steps, again taking about a second of CPU time.

5.2 Other CER combinators

CER combinators can also be defined over product and sum types. The lazy list CER can be generalized to work over any coinductive type that has a notion of depth, such as coinductive trees. A more powerful function-space CER is discussed in Sect. 6.

5.3 Demonstrating equality between coinductive elements

Converging equivalence relations can also be useful in showing that two elements of a target space are equal. Axiom (6) (restated below) says that to show two target elements x and y are equal, one simply needs to show they are equivalent at all resolutions j

$$(\forall j \,.\, x \overset{j}{\approx} y) \longrightarrow x = y.$$

We can often demonstrate that x and y are equivalent at all resolutions by well-founded induction, since $(<)$ is a well-founded relation. For example, given two arbitrary lazy lists ys and zs, we can prove the following lemma about *lappend* by (simple) induction on i, followed by a case split on xs:

Lemma 3

$$\forall xs \,.\, \mathit{ltake}\, i\, (\mathit{lappend}\, (\mathit{lappend}\, xs\, ys)\, zs) = \mathit{ltake}\, i\, (\mathit{lappend}\, xs\, (\mathit{lappend}\, ys\, zs)).$$

The proof takes four steps in Isabelle. Given (6) instantiated to the lazy list CER, we can then easily show in one Isabelle step that $\mathit{lappend}\,(\mathit{lappend}\, xs\, ys)\, zs = \mathit{lappend}\, xs\,(\mathit{lappend}\, ys\, zs)$.

6 Defining functions with unbounded look-ahead

The functions we have defined so far examine their arguments by performing at most one pattern match on a lazy list before producing an element of a result list. However, there is a class of functions that can examine a potentially infinite amount of their argument lists before deciding the next element to output. An example is the *lazy filter* function of type $(\alpha \to \mathit{bool}) \to \alpha\, \mathit{llist} \to \alpha\, \mathit{llist}$, which takes a predicate P and a lazy list xs, and returns a lazy list of the same type consisting only of those elements of xs satisfying P. A candidate set of recursion equations for this function might be

$$\begin{array}{lll}
\mathit{lfilter}\, P\, [] & = [] & \\
\mathit{lfilter}\, P\, (x \# xs) & = \mathit{lfilter}\, P\, xs, & \text{if } \neg(P\, x) \\
\mathit{lfilter}\, P\, (x \# xs) & = x \# (\mathit{lfilter}\, P\, xs), & \text{if } P\, x
\end{array}$$

Sadly, this intuitively appealing set of equations does not completely define *lfilter*. If *lfilter* is given an infinite list xs, none of whose elements satisfy P, then the above equations do not specify what the result list should be. The *lfilter* function is free to return any value at all in this case. In other words, the equations do not have a unique solution.

Happily we can remedy the situation as follows: We define by induction over *nat* a predicate *firstPelemAt* of type $(\alpha \to \mathit{bool}) \to \alpha\, \mathit{llist} \to \mathit{nat} \to \mathit{bool}$. The expression $(\mathit{firstPelemAt}\, P\, xs\, i)$ is true if xs has at least $(i + 1)$ elements and i

is the position of the first element of xs satisfying P. We can then define the predicate *never* of type $(\alpha \to \mathit{bool}) \to \alpha\, \mathit{llist} \to \mathit{bool}$ as

$$\mathit{never}\, P\, xs \equiv \forall\, i\, .\, \neg(\mathit{firstPelemAt}\, P\, xs\, i)$$

which is true when there are no elements in xs satisfying P. If we modify the initial recursive equations as follows:

$$\begin{array}{lll} \mathit{lfilter}\, P\, xs & = [], & \text{if } \mathit{never}\, P\, xs \\ \mathit{lfilter}\, P\, (x\#xs) & = \mathit{lfilter}\, P\, xs, & \text{if } \neg(\mathit{never}\, P\, xs) \wedge \neg(P\, x) \\ \mathit{lfilter}\, P\, (x\#xs) & = x \,\#\, (\mathit{lfilter}\, P\, xs), & \text{if } P\, x \end{array}$$

then the set of equations does indeed have a unique solution. This function is not computable, since the predicate *never* can scan an infinite number of elements, but it is nevertheless mathematically valid in HOL. The CERs described above are not powerful enough to prove this, but we can define a *well-founded function-space* CER combinator that is. Given a CER C with $(<)$ of type $\rho \to \rho \to \mathit{bool}$ and $(\approx)$ with type $\rho \to \tau \to \tau \to \mathit{bool}$, and another well-founded transitive relation $(\prec)$ of type $\sigma \to \sigma \to \mathit{bool}$, we define our new CER C' with $(<')$ and $(\approx')$ as follows:

$$\begin{array}{l} (<') :: (\rho * \sigma) \to (\rho * \sigma) \to \mathit{bool} \\ (\approx') :: (\rho * \sigma) \to (\sigma \to \tau) \to (\sigma \to \tau) \to \mathit{bool} \end{array}$$

$$\begin{array}{ll} (a', t') <' (a, t) & \equiv a' < a \vee (a' = a \wedge t' \prec t) \\ g \overset{(a,t)}{\approx'} h & \equiv \forall\, a'\, t'\, .\, (a', t') \leq' (a, t) \longrightarrow (g\, t') \overset{a'}{\approx} (h\, t') \end{array}$$

It is a fair amount of work to show that C' is in fact a CER, and space constraints force us to elide the details.

Intuitively, however, C' allows us to generalize well-founded recursion in the following way: A well-founded recursive function is forced to have its argument decrease in size on every recursive call. With C', the function being defined is allowed a choice; it can either decrease the size of its argument when making a recursive call, or not decrease its argument size but then make sure the element it is returning is "larger" than the element returned from its recursive call.

In the case of functions returning lazy lists, a "larger" lazy list is one that looks just like the lazy list returned by the recursive call, but with at least one extra element added to the front.

For us to use C' on *lfilter*, we need to specify a suitable well-founded transitive relation $(\prec)$. The relation we choose is one that holds when the first element satisfying P occurs sooner on the left-hand argument than on the right-hand argument:

$$xs \prec ys \equiv \mathit{firstPelem}\, P\, xs < \mathit{firstPelem}\, P\, ys$$

where

$$\begin{array}{lll} \mathit{firstPelem}\, P\, xs & = 0, & \text{if } \mathit{never}\, P\, xs \\ & = 1 + (\varepsilon i\, .\, \mathit{firstPelemAt}\, P\, xs\, i), & \text{otherwise} \end{array}$$

We arbitrarily decide that a list containing no P-elements is $\prec$-smaller than any list with at least one P-element.

When analyzing the revised recursive equations for *lfilter*, if xs has no P-elements then we return immediately, otherwise xs has to have at least one P-element. If that element is not at the head of the list, then the tail of the list is $\prec$-smaller than xs. If the first P-element is at the head of xs, then the tail of the list is not $\prec$-smaller than xs, but the output list has one more element than the list returned by the recursive call. Thus we informally conclude that *lfilter* is uniquely defined.

We have also proved this fact formally in Isabelle. After inductively proving various simple lemmas about *firstPelemAt*, *never*, and *firstPelem*, we were able to prove that *lfilter* is uniquely defined in five steps. We first translated the recursive equations above into a contracting function F. We used C' prove that F is contracting, first by expanding the definition of F and simplifying, and then by performing a case analysis (no induction required!) on whether the *nat* component of the current resolution was equal to zero. It took Isabelle two seconds to perform the proof.

Although we had to prove lemmas about *firstPelemAt*, *never*, and *firstPelem*, the proofs are not hard and it turns out we can reuse these results when defining other functions that perform unbounded search on lazy lists. For example, the *lflatten* function takes a lazy list of lazy lists, and flattens all of the elements into a single lazy list. The *lflatten* function can also be uniquely defined using *never*:

$$\begin{array}{lll} \mathit{lflatten}\, xss & = [], & \text{if } \mathit{never}\, (\lambda xs\,.xs \neq [])\, xss \\ \mathit{lflatten}\, (xs\#xss) & = \mathit{lappend}\; xs\, (\mathit{lflatten}\, xss), & \text{otherwise} \end{array}$$

The proof proceeds in Isabelle exactly as it does for *lfilter* except that we perform one additional case analysis on whether $xs = []$. The proof takes three seconds to complete.

7 Proof of the main result

Although the proof of the main theorem is too lengthy to describe here, we will provide a rough outline. Given a CER with resolution space ρ, target space τ, well-founded relation $(<)$, indexed equivalence relation $(\approx)$, and an arbitrary contracting function F of type $\tau \to \tau$, the technique will be to construct an approximation map *apx* F that converges globally to the desired fixed point. We then prove that this fixed point is unique by showing that any two fixed points of F are equal.

The function *apx* of type $(\tau \to \tau) \to \rho \to \tau$ that builds an approximation map from a contracting function is defined by well-founded recursion on $(<)$ as follows:

$$\mathit{apx}\, F\, i \equiv F\, (\mathit{local\_limit}\, (\mathit{cut}\, (\mathit{apx}\, F)\, i)\, i)$$

where

$$\mathit{cut}\, f\, i\, x \equiv \text{if } x < i \text{ then } f\, x \text{ else } \mathit{arbitrary}.$$

At each resolution i, the function *apx* uses *local_limit* to obtain the best possible approximation of $\mathrm{fix}\, F$, given the approximations it has already computed at all lower resolutions[4]. The result of calling *local_limit* may still not be close enough at resolution i, so *apx* maps the local limit through F, which will bring the result close enough. The helper function *cut* is used to ensure that the recursive call to $apx\, F$ is only made at lower resolutions than i, ensuring well-foundedness. If *local_limit* attempts to invoke $cut\, (apx\, F)\, i$ at any other resolution, then *cut* returns an arbitrary element instead.

Once we have proved by well-founded induction that *apx* is well defined, the next step is to establish that $apx\, F$ is convergent up to each resolution i. To do this we prove several lemmas, such as: if an approximation mapping f converges up to a local limit element z at resolution i, and also converges up to a local limit element z' at the same resolution, then z and z' are equivalent at all resolutions $i' < i$. With this, and the fact that F is contracting, we can show that if $x \overset{i}{\approx} y$, then $F\, x \overset{i}{\approx} F\, y$. We then eventually show for all resolutions i that if $apx\, F$ converges up to local limit element $apx\, F\, i$ at resolution i, then $apx\, F\, i \overset{i}{\approx} F\, (apx\, F\, i)$. This lemma is the key to showing by well-founded induction over i that $apx\, F$ does in fact converge up to $apx\, F\, i$ at resolution i, and is also used to show that $global\_limit\, (apx\, F) \overset{i}{\approx} F\, (global\_limit\, (apx\, F))$ at each resolution i, and are thus equal by (6). This result establishes that a fixed point exists for F. We then show that any two fixed points x and y of F are equivalent at all resolutions by well-founded induction, and thus are equal, again by (6).

8 Conclusion

Related work The support for and application of well-founded induction and general coinduction has seen wide acceptance in the HOL theorem proving community. The well-founded definition package TFL used in HOL98 and Isabelle was written by Slind[13]. It can handle nested pattern matching in rule definitions, nested recursion in function bodies, and generates custom induction rules for each definition[14]. The PVS theorem prover[12] also uses well-founded induction as a basic definitional principle. A general theory of inductive and coinductive sets in Isabelle was developed by Paulson[10], based on least and greatest fixed points of monotone set-transforming functions, as well as a package for defining new inductive and coinductive sets by user-given introduction rules. The package avoids syntactic restrictions in the introduction rules by reasoning about each rule's underlying set-transformer semantics.

A coinductive theory of streams (infinite-only lists) was developed by Miner[7] in the PVS theorem prover. Miner used this theory to model synchronous hardware circuits as corecursively-defined stream transformers. Using coinduction, he was able to optimize the implementation of a fault-tolerant clock synchronization circuit and a floating-point division circuit.

[4] Here the definition of *local_limit* using Hilbert's choice operator seems essential.

A well-known alternative to coinductive types is the mathematical framework of *pointed complete partial orders* and *continuous functions*, also known as *domain theory*[4, 15]. This theory is supported by the HOLCF[8] object-logic in Isabelle, and also allows one to define infinite data structures such as lazy lists and trees. A wide variety of functions over these structures can then be recursively defined. The primary disadvantage of this approach is that one must add "extra" bottom-elements to the structures being defined. These extra elements are used to indicate that a function is non-terminating on its arguments. For example, the lazy filter function *lfilter* can be defined recursively in HOLCF, but the expression *lfilter* P *xs* returns $\perp$ instead of [] when *xs* is an infinite list containing no elements satisfying P. Also, only so-called *admissible* predicates can be reasoned about inductively in domain theory, and it can be quite challenging to prove that a desired predicate is admissible. A comparison of the HOLCF approach to several other encodings of lazy lists is presented by Devillers et al[2].

The theory of topology[1, 11] provides another well-established definition mechanism. The notions of Cauchy sequences, complete metric spaces, and contractions inspired much of this work. We have not worked out the exact relationship between converging equivalence relations and Cauchy metric spaces; although one can construct a distance function for every *nat*-indexed CER, it is not clear that distance functions can always be constructed for more complex resolution spaces. Also, the conditions under which a function F is contracting in a CER seem to be less restrictive than the corresponding conditions in a metric space. More importantly from a verification perspective, well-founded induction seems easier to apply in current theorem provers than does the continuous mathematics required for metric spaces.

Current and future work We are currently using CERs to specify and reason about processor microarchitectures as recursively defined stream transformers. This work is part of the Hawk project[6], which is developing a domain-specific functional language for specifying, simulating, and reasoning about such microarchitectures at a high level of abstraction. We have been able to use CERs and the unique fixed point lemmas in Sect. 3.2 to develop a domain-specific *microarchitecture algebra*[5] in Isabelle, which we use to verify Hawk specifications.

Although we have defined CERs over streams and lazy lists, many structures in language semantics and process algebras can be seen as coinductive trees. It would be interesting to define some of these structures recursively and reason about them inductively, as we did for *lappend* in Sect. 5.3.

9 Acknowledgements

We wish to thank Byron Cook, Sava Krstić, and John Launchbury for their valuable contributions to this research. We are also grateful for the anonymous referees' helpful suggestions. The author is supported by a graduate research fellowship with the National Science Foundation, and grants from the Air Force Material Command (F19628-93-C-0069) and Intel Strategic CAD Labs.

References

1. Buskes, G., and van Rooij, A. *Topological Spaces: from distance to neighborhood.* UTM Series. Springer, New York, 1997.
2. Devillers, M., Griffioen, D., and Müller, O. Possibly infinite sequences in theorem provers: A comparative study. In *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97* (Murray Hill, NJ, Aug. 1997), vol. 1275 of *LNCS*, Springer-Verlag, pp. 89–104.
3. Gordon, M. J. C., and Melham, T. F. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
4. Gunter, C. A. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Science. The MIT Press, 1992.
5. Matthews, J., and Launchbury, J. Elementary microarchitecture algebra. To appear in CAV99, International Conference on Computer Aided Verification, July 1999.
6. Matthews, J., Launchbury, J., and Cook, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 90–101.
7. Miner, P. *Hardware Verification Using Coinductive Assertions.* PhD thesis, Indiana University, 1998.
8. Müller, O., Nipkow, T., v. Oheimb, D., and Slotosch, O. HOLCF = HOL + LCF. To appear in Journal of Functional Programming, 1999.
9. Paulson, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.
10. Paulson, L. C. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation 7*, 2 (Apr. 1997), 175–204.
11. Rudin, W. *Principles of Mathematical Analysis*, 3 ed. McGraw-Hill, 1976.
12. Rushby, J., and Stringer-Calvert, D. W. J. A less elementary tutorial for the PVS specification and verification system. Tech. Rep. SRI-CSL-95-10, SRI International, Menlo Park, CA, June 1995. Revised, July 1996.
13. Slind, K. Function definition in higher order logic. In *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL* (Turku, Finland, Aug. 1996), J. Von Wright, J. Grundy, and J. Harrison, Eds., vol. 1125 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 381–398.
14. Slind, K. Derivation and use of induction schemes in higher-order logic. *Lecture Notes in Computer Science 1275* (1997), 275–290.
15. Tennent, R. D. *Semantics of Programming Languages.* Prentice Hall, New York, 1991.