Implicit programming
and
computable optimal fixed points

Anna Patterson and Tom Costello

Computer Science Department, Stanford University, Stanford, CA 94305,

Email {annap,costello}@cs.Stanford.EDU

Abstract

If a program has a unique solution then programming semantics should return that solution. Optimal fixed points represent this unique solution; that is, if the only choice is between undefined and a specific defined value then we choose the defined value. However, optimal fixed points can be uncomputable. For this reason they have been relatively underutilized.

Here we provide a method which computes the optimal fixed point when it is computable. To do this we use proofs of uniqueness in an extension of intuitionistic logic together with its associated Curry-Howard-De Bruijn isomorphism. Using this correspondence, we extract programs in a confluent strongly normalizable lambda calculus that computes the optimal fixed point of a recursively defined function. When this uniqueness proof exists, the optimal fixed point is computable.

We give several examples of where this fixed point gives intuitive answers. This new semantics is especially natural for functions on co-recursively defined data-types, like streams or circular lists.

1 Introduction

We present a new semantics for recursive programs. Rather than choose the least defined solution, we suggest that if there is a uniquely defined solution, then that solution should be chosen. We show that this uniquely defined solution corresponds to Manna and Shamir's optimal fixed point [MS76, MS77].

The optimal fixed point is uncomputable in general. However, using notions from proof theory [TS96], we show how a constructive proof of an implicit definition, in a given system, corresponds to a function. We use the Curry-Howard-De Bruijn isomorphism in the first order case, and show how least fixed points correspond to proofs of existence, while constructive optimal fixed points correspond to proofs of uniqueness.

An assignment of terms to proofs gives us a lambda calculus in which these optimal fixed points are calculated. We give examples throughout the paper to show the usefulness of the technique.

Some of the examples are functions on the natural numbers, and other examples are functions which act upon co-recursively defined data-structures, circular lists up to bisimulation. Implicit definitions of programs are very natural for functions whose arguments could include co-recursively defined data-structures [BM96, Fio96, MD96, Par83, MT91, Pit94, Gor95].

1.1 Fixed Points

Many programming languages are naturally characterized in terms of the least fixed point of a recursive operator. The least fixed point of a recursive operators is computable.

Consider a simple example of the factorial function.

$$\begin{aligned} &f(0) = 1 \\ &f(n+1) = (n+1) * f(n) \end{aligned}$$

This is a recursive definition of the factorial function. We can make this definition into an operator, that is a function on functions as follows.

$$\Phi(f) = \lambda x.\ \begin{aligned} &\text{if } x = 0 \text{ then } 1 \\ &\text{else } (x+1) * f(x) \end{aligned}$$

This is a functional on functions. If we apply Φ to the totally undefined function $\lambda x.\bot$, we get $\lambda x.$ if $x = 0$ then 1 else $\bot$. If we apply Φ to the result, we get $\lambda x.$ if $x = 0$ then 1 if $x = 1$ then 2 else $\bot$. We can ask, what is the least defined function g which has the property that $\Phi(g) = g$. In this case, the answer is the factorial function. In general, the least fixed point of an operator is the function that is a fixed point of the operator and is undefined as often as possible.

1.2 Optimal Fixed Points

Other fixed points are possible, such as the optimal fixed point. Optimal fixed points are sometimes more natural than the least fixed point for certain classes of programs. We pick an example to illustrate the difference between optimal and least fixed points. We need a different example than the factorial example presented above because for primitive recursive functions, optimal fixed points agree with least fixed points.

Consider the following recursive definition of a function.

$$f(x) = 0 * f(x).$$

There is a unique total function that satisfies this equation — the function $\lambda x.0$. However, there are many partial functions that also satisfy this equation; those are the partial functions that return 0 for some inputs. The operator corresponding to this recursive definition is

$$\Phi_0(f) = \lambda x.0 * f(x)$$

The least fixed point of this operator is $\lambda x.\bot$ — the totally undefined function. The constant zero function seems a more informative answer. A natural intuition is that if there is a unique answer, then the program should return that answer, rather than not return at all. The optimal fixed point of the operator Φ_0 in the example above is the function $\lambda x.0$.

Definition 1 *Let L be a signature not mentioning the function symbols f and f'. Let $\Gamma(f)$ be a set of sentences in $L \cup \{f\}$ and let $\Gamma(f')$ be the same set of sentences replacing f by f'.*

$\Gamma(f)$ implicitly defines f *if and only if* $\Gamma(f) \cup \Gamma(f') \models \forall \bar{x}.\forall y.[f(\bar{x}) = y \iff f'(\bar{x}) = y]$.

This means that if two models of Γ agree on the interpretation of the L formulas then they must agree on the interpretation of the function symbol f. This can easily be adapted to implicit definitions of relations and even constants.

Now we consider implicit definitions and functions which act on co-recursive data structures. Consider list structures, as in lisp. A natural notion of equality, (called `equal` in lisp) is axiomatized as follows.

$$l = l' \iff car(l) = car(l') \wedge cdr(l) = cdr(l').$$

The least fixed point of `equal` correctly characterizes equality for lists that do not contain loops. Two cyclic lists however are not equal, even to themselves, by the least fixed point of `equal`.

Another possibility is to consider a different equivalence relation on lists: bisimilarity[Mil89, BM96]. Recently this notion of defining data-types in this way, (co recursively) has been considered by many researchers[Gor94, Gor95, Pit94, Fre90].

With bisimilarity as our notion of equivalence of lists, we can consider simple programs on lists. The following function is intended to return a list with every occurrence of a natural number replaced with the symbol '***. Perhaps we wish to remove all numbers from a document before making it public.

$$\begin{aligned} f(x) = \quad & \text{if } \mathrm{nat}(x) \text{ then } '***\\ & \text{elseif } \mathrm{atom}(x) \text{ then } x\\ & \text{else } \mathrm{cons}(f(car(x)), f(cdr(x)))\end{aligned}$$

The domain of the least fixed point of this function f is defined for all lists that do not contain loops. However, for cyclic lists the function is undefined. The natural answer is that every number in a list should be replaced by '***. One thing we can see is that there is only one totally defined solution of the above definition. Further, it is this totally defined function is our intuitively correct answer and this function is defined implicitly. We would like this implicit answer for the output of our program. We now use this intuition to define another fixed point.

Definition 2 (Manna and Shamir) *The* optimal fixed point f_o *of an operator* Φ *is the meet of all of the maximal fixed points of* Φ.

Note: Every maximal fixed point is a total function. We now present two more ways of understanding optimal fixed points.

Theorem 3 *Let* Φ *be an operator, then the following are all equivalent.*

1. f_o *is the optimal fixed point.*
2. f_o *is the only fixed point of* Φ *such that for every value* x*, and for every fixed point* g *of* Φ*, if whenever* $g(x)$ *is defined it has the value* y*, then* $f_o(x) = y$.
3. $\forall x, y \in \mathbb{N}. f_o(x) = y$ *if and only if* $\forall g, g'.$ *that are total functions* $\Phi(g) = g \;\wedge\; \Phi(g') = g' \models g(x) = g'(x) = y$.

The last definition is in the style of an implicit definition.

The optimal fixed point seems more useful than the least fixed point, because it defines functions that return values at more arguments. However, it lacks one important feature of the least fixed point: *computability*. The least fixed point of a recursive operator Φ is computable in the following way. Suppose we wish to evaluate the least fixed point of Φ at a value v. We apply Φ to the totally undefined function $\lambda x.\bot$. As Φ is recursive, this returns a function, which we call f_1. We apply Φ to f_1 to get f_2, f_3, ..., f_n .... If the least fixed point of Φ is defined at v, then for some value n, $f_n(v)$ will be defined. Thus, after a finite number of steps, we will find the value of the least fixed point of Φ at any given value v.

The optimal fixed point cannot be evaluated in this way. In fact, there is no way to evaluate the optimal fixed point, as the function it defines can be uncomputable.

Rather than consider the implicitly defined output values, we consider the functions whose outputs we can constructively prove are implicitly defined. We define this new fixed point by using the relationship between constructive logic and computation.

2 The Curry-Howard-De Bruijn Isomorphism

The Curry-Howard-De Bruijn isomorphism states that proof normalization in propositional intuitionistic logic is isomorphic to reductions in the typed lambda calculus [TS96]. We consider an extension of this isomorphism to a conservative extension of first order intuitionistic logic, N^- [Nel49]. This adds one new connective, an involutory constructive negation, $\neg$, which is read as counterexample.

In the appendix we define the logic N^- and give an assignment of terms in a lambda calculus to proofs in this logic. These terms have a set of reductions, $\triangleright$, associated with them, shown in the following table. This is the λ_{FIRZ} calculus; for first-order intuitionistic logic with recursion and a new Z reduction for zero terms. The assignment of types for this variant is found in the appendix.

Intuitionistic Logic Reductions	
$\pi_i(\langle a_1, a_2\rangle) \triangleright a_i$	$(\lambda X.a\ b) \triangleright a[b/X]$
$(\boldsymbol{\lambda} x.a\ u) \triangleright a[u/x]$	$D_{X_1,X_2}(k_i\ a, b_1, b_2) \triangleright b_i[a/X_i]$
$E_{x,Y}(\langle v, a\rangle, b) \triangleright b[v/x, a/Y]$	
Recursion Reductions	
$\mathbf{r}ab0 \triangleright a$	$\mathbf{r}abSn \triangleright ((bn)\mathbf{r}abn)$
Z Reduction	
$(\pi_1(Z)\ Sn) \triangleright \lambda a.\langle Sn, eq(Sn)\rangle$	

Theorem 4 *The lambda calculus λ_{FIRZ} is confluent and strongly normalizable.*

Outline of Proof: We obtain this result by reducing our terms to a standard lambda calculus via propagation of negation inwards so that it only scopes relation symbols. We then introduce new relation symbols for every negated relation. We then use standard results [TS96, TvD88a, TvD88b, Min96]. The Z terms cannot be reduced by any other reduction; thus the standard results hold with its addition. ∎

2.1 Translations

We need a language in which to define recursive functions. We consider our language to be general recursive equations [Fef86, Kle67, McC63] which by the MRDP theorem is Turing complete. By reducing the Kleene normal form via a Maslov-style transformation, it suffices to define the translation for equations which have function calls at nesting depth two. We consider two translations, one that captures the positive information (the $\exists$ translation), and another that captures the negative information (the $\forall$ translation).

Definition 5 (Translations) *Let e be a set of equations where each equation is of the form $f(\bar{x}) = c$, or $f(\bar{x}) = g(\bar{x}, \bar{c})$ or $f(\bar{x}) = g(h(\bar{x}), \bar{c})$, where $\bar{x}$ is a tuple of variables, or the successor applied to variables or constants or constant terms and $\bar{c}$ is a tuple of constants, or constant terms (e.g. $S(0)$).*

We define the $e^{\exists}$ translation of e as follows: We introduce an $n+1$-ary relation R_g for every n-ary function symbol g, and for each equation a in e we have a formula, $a^{\exists}$

$$
\begin{aligned}
&(f(\bar{x}) = c)^{\exists} \stackrel{def}{=} \forall \bar{x}.R_f(\bar{x}, c)\\
&(f(\bar{x}) = g(\bar{x}, \bar{c}))^{\exists} \stackrel{def}{=} \forall \bar{x}, z.[R_f(\bar{x}, z) \equiv R_g(\bar{x}, \bar{c}, z)]\\
&(f(\bar{x}) = g(h(\bar{x}), \bar{c}))^{\exists} \stackrel{def}{=} \forall \bar{x}, z.[R_f(\bar{x}, z) \equiv \exists y.R_g(y, \bar{c}, z) \wedge R_h(\bar{x}, y)]
\end{aligned}
$$

We define another translation the $\forall$ translation, for the same equations as follows,

$$
\begin{aligned}
&(f(\bar{x}) = c)^{\forall} \stackrel{def}{=} \forall \bar{x}, z.[\neg R_f(\bar{x}, z) \equiv z \neq c]\\
&(f(\bar{x}) = g(\bar{x}, \bar{c}))^{\forall} \stackrel{def}{=} \forall \bar{x}, z.[\neg R_f(\bar{x}, z) \equiv \neg R_g(\bar{x}, \bar{c}, z)]\\
&(f(\bar{x}) = g(h(\bar{x}), \bar{c}))^{\forall} \stackrel{def}{=} \forall \bar{x}, z.[\neg R_f(\bar{x}, z) \equiv \forall y.\neg R_g(y, \bar{c}, z) \vee \neg R_h(\bar{x}, y)]
\end{aligned}
$$

where $A \equiv B$ is $A \supset B \ \wedge \ B \supset A$.

Consider the set of equations we had earlier for factorial. We can translate them using the $\exists$ translation to:

$$
\begin{array}{ll}
R_{factorial}(0, 1) & A_1\\
\forall x, y.[R_{factorial}(S(x), y) \equiv \exists z.R_{factorial}(x, z) \wedge y = z * S(x)] & A_2
\end{array}
$$

which cheats a bit because we represented $*$ as a function instead of a relation.

We can prove that $\forall x.\exists y.R_{factoral}(x, y)$, from A_1 and A_2 using the axiomatization in the appendix. If we consider the term associated with this proof we find that it is a function, from natural numbers, that returns a pair. The first projection of this pair is the answer that we seek. We show the proof of $\forall x.\exists y.R_{factoral}(x, y)$ below.

$$
\dfrac{\dfrac{R(0,1)}{\exists y.R(0,y)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{R(z,b) \quad Sz*b = Sz*b}{R(z,b) \wedge Sz*b = Sz*b}}{\exists z'.R(z,z') \wedge Sz*b = Sz*z'} \quad \dfrac{\dfrac{\forall x, y.R(Sx,y) \equiv \exists z'.R(x,z') \wedge y = Sx*z'}{R(Sz, Sz*b) \equiv \exists z'.R(z,z') \wedge Sz*b = Sz*z'}}{\exists z'.R(z,z') \wedge Sz*b = Sz*z' \supset R(Sz, Sz*b)}}{R(Sz, Sz*b)}}{\exists y.R(Sz,y)} \qquad \exists y.R(z,y)}{\exists y.R(z,y) \supset \exists y.R(Sz,y)}}{\forall x(\exists y.R(x,y)) \supset \exists y.R(Sx,y)}}{\dfrac{\exists y.R(x,y)}{\forall x \exists y.R(x,y)}}
$$

It has as its associated term, if we assign A_1 and A_2 as the terms of the two axioms,

$$
\boldsymbol{\lambda} x.\mathbf{r} \ \langle 1, A_1 \rangle \ \boldsymbol{\lambda} y.\lambda m.E_{u,b}(m, \langle Sz * b, (((\pi_2 A_2 \ z) \ Sz * b) \ \langle b, \langle u(b), eq(Sz * b)\rangle\rangle)\rangle) \ \ x
$$

This function, applied to 2 reduces to

$$\langle 2, (((\pi_2 A_2\ 1)\ 2)\ \langle 1, \langle (((\pi_2 A_2\ 0)\ 1)\ \langle 1, \langle A_1\ eq(1)\rangle\rangle), eq(2)\rangle\rangle\rangle.$$

Its first projection is 2, and thus, as was claimed, factorial(2) = 2.

With the translation of the equation $(f(x) = 0 * f(x))^{\exists}$, we cannot prove prove $R_f(n, 0)$ for any n. This failure is because the least fixed point of this equation is undefined.

In general, given a set of equations recursively defining a function f, we can prove $R_f(n, m)$ in our axiomatization if and only if the least fixed point of the associated operator Φ of the recursive definition of f is defined at n and equal to m.

Thus the values that we can prove are defined in intuitionistic logic correspond to the least fixed point. Given a proof that a value exists, via the Curry-Howard-De Bruijn isomorphism, we can extract that value, by considering the proof as a program that calculates that value.

Theorem 6 *Given a proof of $\exists x.\phi(x)$ in the formalization of arithmetic given in the Appendix, where ϕ has no free variables besides x, we can extract a number n such that $\phi(n)$.*

Outline of Proof: Apply cut-elimination which is only a slight modification of [TvD88a, Min96]. The only modification is that care must be taken with the Z reduction, since it contains an existential. If the Z reduction occurs in the main branch it normalizes to a value, if not then it does not participate in the output value. Thus after cut-elimination the last rule must be $\exists$ introduction, thus the previous line must be $\phi(t)$ for some closed term t. ∎

Theorem 7 *The least fixed point of a program f, defined by a set of equations e, is computed by the term associated with a proof of*

$$\forall \bar{x} \exists y. R_f(\bar{x}, y)$$

from the sentences in $e^{\exists}$ using the axiomatization in the Appendix.

Outline of Proof: We instantiate the universal and prove $\exists y.R_f(\bar{c}, y)$. By the previous theorem, we can extract a value by normalizing the proof. The function associated with the proof is the least fixed point because of the equivalence of least fixed points and Kleene's modified realizability [Kle73] and the equivalence of Curry-Howard-De Bruijn and modified realizability [Min92]. ∎

The property that we can extract a value m satisfying ϕ from a proof of $\exists x.\phi(x)$ is called the *disjunction property*.

We consider a constructive logic that has the disjunction property and which adds a new negation. This new negation $\neg$ is negation as constructing a counterexample and thus differs from intuitionistic logic's negation $\sim$ which is negation as leading to contradiction. This new negation $\neg$ allows us to state uniqueness constructively.

The usual statement that there is a unique value x that satisfies ϕ is

$$\forall x, y.[\phi(x) \wedge \phi(y) \supset x = y]$$

we can re-express this as

$$\exists x.\forall y.[x \neq y \supset \neg\phi(y)].$$

The formula implies that for any number y not equal to x, we can construct a counterexample to y being a solution of ϕ. It is this information that we need for computable optimal fixed points. As we have a logic with the disjunction property we can extract the value of x and actually compute the optimal fixed point.

However, we need the $\forall$ translation, as the $\exists$ translation of the recursive definition of factorial contains no information about when factorial is not true of an input-output pair; it is this negative information we now need.

The $\forall$ translation of $f(x) * 0 = f(x)$ is

$$\forall x, z.[\neg R_f(x, z) \equiv \forall y.\neg R_f(x, y) \vee z \neq y * 0] \qquad A$$

Using axiom A, we can prove

$$\forall x.\exists y.\forall z.[z \neq y \supset \neg R_f(x, z)]$$

using the following proof,

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{z \neq 0 \quad 0 = u * 0}{z \neq u * 0}}{\neg R_f(x, z) \vee z \neq u * 0}}{\forall u.\neg R_f(x, u) \vee z \neq u * 0} \quad \dfrac{\dfrac{\forall xz.(\forall u.\neg R_f(x, u) \vee z \neq u * 0) \equiv \neg R_f(x, z) \quad A}{(\forall u.\neg R_f(x, u) \vee z \neq u * 0) \supset \neg R_f(x, z)}}{(\forall u.\neg R_f(x, u) \vee z \neq u * 0) \supset \neg R_f(x, z)}}{\neg R_f(x, z)}}{z \neq 0 \supset \neg R_f(x, z)}}{\forall z.z \neq 0 \supset \neg R_f(x, z)}}{\exists y \forall z.z \neq y \supset \neg R_f(x, z)}}{\forall x.\exists y \forall z.z \neq y \supset \neg R_f(x, z)}
$$

We can associate a term with this proof

$$\boldsymbol{\lambda} x.\langle 0, \boldsymbol{\lambda} z.\lambda W.(\pi_2(((A\ x)\ z)), \boldsymbol{\lambda} u.k_2\ ((sub_{x \neq 0, x}(0, u * 0), (eq_0\ u)))\ W)\rangle$$

where we avoid an axiomatization of multiplication be denoting $\forall x.(0 = 0 * x)$ as eq_0. The term sub is defined in the Appendix.

As before, to evaluate the function, we apply the term to the given value, and take the first projection. It is clear that this gives the constant zero function.

Theorem 8 *The optimal fixed point of a program f, where f is defined by a set of equations e, is computed by the term associated with a proof of*

$$\forall \bar{x} \exists y \forall z.z \neq y \supset \neg R_f(\bar{x}, z)$$

from the sentences in $e^{\forall}$ using the axiomatization in the appendix.

Outline of Proof: As before, we normalize the proof and extract the value. This is the optimal fixed point, because the definition of an optimal fixed point aligns with the formula because to prove the formula we must show that the function f must have the value y when it is defined. ∎

The formula implies that for any number z not equal to y, we can construct a counterexample to z being a solution of f. This means that f cannot help but have y as its value when it has a value. The strong normalization property implies that when we have a constructive proof of uniqueness, then the optimal fixed point is computable.

Corollary 9 *If from a set of sentences* $e^\forall$ *we can prove* $\forall \bar{x} \exists y \forall z. z \neq y \supset \neg R_f(\bar{x}, z)$*, then the optimal fixed point is computable.*

Therefore, we have shown that we can extract programs that compute optimal fixed points from proofs of uniqueness, analogously to how least fixed points arise from proofs of existence.

In the full paper we will discuss what optimal fixed points we can prove in this way. We know that for a given function whose optimal fixed point is computable, there is an axiomatization whose proof calculates the optimal fixed point. This depends heavily on the choice of the axiomatization, in particular the strength of the induction principle. This topic will be discussed further in the full version.

3 Examples

We now consider two examples which show the applicability of this approach.

This first example is included because it differs from standard methods of partial evaluation.

Example: 1 *Consider the following recursively defined function,*

$$2 * f(x) = f(x).$$

We cannot show,

$$\forall x \exists y. R_f(x, y),$$

from the existential translation

$$(2 * f(x) = f(x))^\exists \quad = \quad \forall xy. R_f(x, y) \equiv \exists z. R_f(x, z) \wedge y = 2 * z,$$

but we can prove,

$$\forall x \exists y \forall z. z \neq y \supset \neg R_f(x, z).$$

from the universal translation

$$(2 * f(x) = f(x))^\forall \quad = \quad \forall xy. \neg R_f(x, y) \equiv \forall z. \neg R_f(x, z) \vee y \neq 2 * z$$

We prove this by proving,

$$\forall z. z \neq 0 \supset \neg R_f(x, z).$$

We prove this by considering the fact that every non-zero number is the product of an odd number y *and a power of* 2*.*

$$\forall x. x \neq 0 \supset \exists yn. x = y * 2^n \wedge \forall u. y \neq 2 * u$$

Using this fact, and the assumption that $z \neq 0$*, we can prove, by induction on* n*, that* $\neg R_f(x, z)$*.*

We now consider an example that uses co-recursively defined data-structures, e.g. lists equivalent up to bisimulation. The following is a natural program that extracts the leftmost path through a list structure. We give it as an example of how programs on co-recursive data-structures, that arise as optimal fixed points, may adjust the pointer structure. The least fixed point of this program is undefined on all lists whose leftmost branch is circular.

Example: 2 *Consider the Lisp program:*

```
(defun f(l)
  (cond ((atom l) l)
    (t (car (f l)))))
```

The above Lisp program can be written pseudo-*equationally as,*

$$f(cons(a, b)) = cons(f(a), nil)$$
$$atom(a) \rightarrow f(a) = a$$

The optimal fixed point is computable, and selects the leftmost branch. The $\forall$ translation of the above definition gives,

$$\forall xyz.\neg R_f(cons(x, y), z) \equiv \forall u.\neg R_f(x, u) \vee cons(u, nil) \neq z$$
$$\forall xz.atom(x) \supset (\neg R_f(x, z) \equiv x \neq z)$$

From these two sentences and some basic facts about *cons* and list equality, we can prove

$$\forall x \exists y.\forall z.z \neq y \supset \neg R_f(x, z).$$

The associated lambda calculus term is a program that correctly calculates the optimal fixed point of f, that is, it correctly extracts the leftmost branch.

One extra reduction is needed to guarantee that values can be extracted from proofs of existential sentences in this axiomatization. This relates to how inequality of lists is defined in terms of bisimulation. This is considered in the full paper.

4 Conclusion

Programs are naturally represented using recursive definitions of functions [McC63]. These sets of equations can be given a least fixed point semantics [SS71]. Least fixed point semantics suggest that a function should be undefined as often as possible.

Another possible semantics is the optimal fixed point [MS77]. This suggests that if a function could only have one value, then we should choose that value, rather than choose undefined. The optimal fixed point is always as defined as the least fixed point, and thus can extract more content from our programs.

The optimal fixed point therefore seems like a much better choice of semantics. There remains but one obstacle, sometimes the optimal fixed point is not computable.

We avoid this pitfall by looking at the problem in another way, using the Curry-Howard-De Bruijn isomorphism. This relates programs in a lambda calculus with proof normalization in a constructive logic. The computation of the least fixed point, expressed in lambda calculus, gives a proof that the function is defined. We show that a proof that the function is uniquely defined corresponds to a computation of the optimal fixed point.

The constructive logic that we use is N^-, and the lambda calculus is a variant of that used for normalizing intuitionistic proofs. We show that the calculus is confluent and strongly normalizable, and that it correctly computes the optimal fixed points.

Our approach gives a way of computing the optimal fixed point of recursive equations. This allows a computational interpretation of optimal fixed points, improving over the commonly used least fixed point.

Optimal fixed points are especially useful for co-recursively defined data-structures, but can differ from the least fixed points even on recursively defined data-structures like the natural numbers.

Acknowledgments

We would like to thank Grisha Mints and Carolyn Talcott for many helpful discussions. We would like to thank Sol Feferman, John McCarthy, Ugo Montanari, Gordon Plotkin, Vaughan Pratt and Johan van Benthem for encouragement and helping to clarify many of the concepts and Viviana Bono for many readings of the manuscript.

References

[BM96] J. Barwise and L. Moss. *Vicious Circles.* CSLI, 1996.

[Fef86] S. Feferman, editor. *Kurt Gödel Collected Works.* Oxford University Press, Oxford, UK, 1986.

[Fio96] M. Fiore. A coinduction principle for recursive data types based on bisimulation. *Information and Computation*, pages 186–198, 1996.

[Fre90] P. Freyd. Recursive types reduced to inductive types. *IEEE Symposium on Logic in Computer Science*, pages 498–508, 1990.

[Gor94] Andrew Gordon. A tutorial on co-induction and functional programming. In *GLA*, pages 78–95. Springer Workshops in Computing, 1994.

[Gor95] Andrew Gordon. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[Kle67] S.C. Klenne. *Mathematical Logic.* Wiley, 1967.

[Kle73] S.C. Kleene. *Realizability: a retrospective survey*, volume 337 of *Lecture Notes in Mathematics.* Springer-Verlag, 1973.

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D Herschberg, editors, *Computer programming and formal systems*, pages 33–70. North-Holland, 1963.

[MD96] L. Moss and N. Danner. On the Foundations of Corecursion. Technical report, Indiana University, Bloomington, 1996.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Min92] G.E. Mints. *Selected Papers in Proof Theory.* Bibliopolis/North-Holland, 1992.

[Min96] G.E. Mints. Programs, proofs and bounds. Course Notes, Spring 1996, 1996.

[MS76] Z. Manna and A. Shamir. The theoretical aspects of the optimal fixedpoint. semantics and correctness of programs. *SIAM Journal of Computing*, 5(3):414–426, 1976.

[MS77] Z. Manna and A. Shamir. The optimal approach to recursive programs. *Communications of the ACM*, (11):824–831, 1977.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[Nel49] D. Nelson. Constructible falsity. *Journal of Symbolic Logic*, 14:14–26, 1949.

[Par83] D. Park. The fairness problem and nondeterministic computing networks. In *Foundations of Computer Science IV, Distributed Systems*, volume 159 of *Mathematisch Centrum Tracts*, pages 33–162. Mathematisch Centrum, Amsterdam, 1983.

[Pit94] A. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report Technical Monograph PRG-6, Oxford University Computing Laboratory, 1971.

[TS96] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univeristy Press, 1996.

[TvD88a] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics; An Introduction Vol. 1.*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Amsterdam, 1988.

[TvD88b] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics; An Introduction Vol. 2.*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Amsterdam, 1988.

Appendix

4.1 Definition of N^-

Terms in N^- are formed from variables and constants by functions in the standard way. Atomic formulas are propositional variables and predicate letters applied to terms. There is a constant 0 and a unary function S. We often write $\overbrace{S \cdots S}^{\text{n times}} 0$ as n. Sometimes we choose to have our terms denote lists, and have functions $car, cdr, cons$. There are distinguished predicates $\neq$ and $=$. Formulas are constructed from atomic formulas by means of the binary connectives $\wedge$, $\vee$, and $\supset$, and the unary connective $\neg$. We use $A \equiv B$ to mean $A \supset B \wedge B \supset A$. For simplicity we propagate negation through all the other connectives, via the following table.

$$\begin{array}{rcl rcl}
\neg(A \wedge B) & \rhd & \neg A \wedge \neg B & \neg(A \vee B) & \rhd & \neg A \vee \neg B \\
\neg(A \supset B) & \rhd & A \wedge \neg B & \neg\neg A & \rhd & A \\
\neg \forall x.A & \rhd & \exists x.\neg A & \neg \exists x.A & \rhd & \forall x.\neg A
\end{array}$$

We call a formula where $\neg$ scopes only atomic formulas a positive normal formula

The following table gives a natural deduction formulation of formulas of N^- in positive normal form, together with an assignment of terms to proofs. The terms are taken from a typed lambda calculus. Any deductive term will have a formula as its type. $t \in A$ or t^A stands for t is a term of type A. For any positive normal formula A in N^- there is a countable set of type A variables $X^A, Y^A, \ldots$. Capital letters from the end of the alphabet denote type variables. Lower case letters from the end of the alphabet denote value variables. Lower case letters from the beginning of the alphabet denote terms. An application expression $(b\ a)$ is a term of type B for any two terms of type $A \supset B$ and A respectively. Further, the following type rules are respected, $\langle a^A, b^B \rangle \in A \wedge B$. $\lambda x^A.b^B \in A \supset B$, $\pi_1 c^{A\wedge B} \in A$, $\pi_2 c^{A\wedge B} \in B$, $D_{x,y}(a^A, b^B, c^C) \in A \supset (B \vee C)$. $k_1 c^A \in A \vee B$, $k_2 c^B \in A \vee B$. Free and bound variables are defined as normal. We associate terms with proof rules in the following way.

$$\wedge \text{ Intro} \quad \frac{\Gamma \Rightarrow a\colon A \quad \Gamma' \Rightarrow b\colon B}{\Gamma, \Gamma' \Rightarrow \langle a, b\rangle\colon A \wedge B} \qquad \wedge \text{ Elim} \quad \frac{\Gamma \Rightarrow a\colon A_1 \wedge A_2 \quad i \in \{1,2\}}{\Gamma \Rightarrow \pi_i a\colon A_i}$$

$$\supset \text{ Intro} \quad \frac{a\colon A, \Gamma \Rightarrow b\colon B}{\Gamma \Rightarrow \lambda a.b\colon A \supset B} \qquad \supset \text{ Elim} \quad \frac{\Sigma \Rightarrow a\colon A \quad \Gamma \Rightarrow b\colon A \supset B}{\Gamma, \Sigma \Rightarrow (b\ \ a)\colon B}$$

$$\vee \text{ Intro} \quad \frac{\Gamma \Rightarrow b\colon A_i}{\Gamma \Rightarrow k_i\ b\colon A_1 \vee A_2}\ i \in \{1,2\}$$

$$\vee \text{ Elim} \quad \frac{\Sigma \Rightarrow d_1\colon A \vee B \quad \Gamma, a\colon A \Rightarrow d[a]\colon C \quad \Gamma', b\colon B \Rightarrow d'[b]\colon C}{\Gamma, \Gamma', \Sigma \Rightarrow D_{U,V}(d_1, d[U], d'[V])\colon C}$$

To these rules we need to add the quantifier rules. To do this we add to our language of deductive terms a new combinator E for existential, a pairing function for values and terms, $\boldsymbol{\langle} v, t \boldsymbol{\rangle}$, a lambda abstractor for values $\boldsymbol{\lambda}$, and an application for values $\mathbf{(}t\ \ v\mathbf{)}$. $E_{y,Z}(a^{\exists x.A(x)}, c^C[y, Z^{A[y]}]) \in \exists x.A(x) \supset C$ $\boldsymbol{\lambda} x.a^{A(x)} \in \forall x.A(x)$.

$$\forall \text{ Elim} \quad \frac{\Gamma \Rightarrow a\colon \forall x.A}{\Gamma \Rightarrow \mathbf{(}a\ \ v\mathbf{)}\colon A[v/x]} \qquad \forall \text{ Intro} \quad \frac{\Gamma \Rightarrow a\colon A}{\Gamma \Rightarrow \boldsymbol{\lambda} x.a\colon \forall x.A}$$

If x is free for v in A — If Γ does not contain x free

$$\exists \text{ Elim} \quad \frac{\Gamma \Rightarrow t\colon \exists x.A, \quad \Delta, a(y)\colon A[y] \Rightarrow t'[y, a(y)]\colon C}{\Gamma, \Delta \Rightarrow E_{u,V}(t, t'[u/y, V(u)/a(y)])\colon C} \qquad \exists \text{ Intro} \quad \frac{\Gamma \Rightarrow a\colon A[v]}{\Gamma \Rightarrow \boldsymbol{\langle} v, a \boldsymbol{\rangle}\colon \exists x.A[x/v]}$$

If y is new in Δ — If v is free for x in A

4.2 Axioms for Arithmetic

We now have some axioms for equality and inequality. We only allow equality rules for atomic formulas for simplicity.

$$\Rightarrow eq(x)\colon x = x \qquad \Rightarrow tr(x,y,z)\colon x = y \wedge y = z \supset x = z$$

$$\Rightarrow sym(x,y)\colon x = y \supset y = x \qquad \Rightarrow sub_{A(x),x}(y,z)\colon (y = z \wedge A(y)) \supset A(z)$$

$\Rightarrow neq_{t,s}\colon s \neq t$
if s and t are distinct terms. $\Rightarrow nsym(x,y)\colon x \neq y \supset y \neq x$

We have an important rule, induction, and its associated recursor term $\mathbf{r}$.

$$\frac{t\colon A(0) \quad t'\colon \forall x.A(x) \supset A(Sx)}{\mathbf{r}tt'n\colon A(n)}$$

We also have the axioms that characterize the natural numbers, and their associated terms.

$$\begin{array}{ll}
Nsub\colon & \forall x.\phi(x) \wedge \neg\phi(y) \supset x \neq y \\
N0\colon & \forall x.S(x) \neq 0 \\
Z\colon & \forall x.x \neq 0 \equiv \exists y.S(y) = x \\
C0\colon & \forall y(\forall x.S(x) \neq y) \equiv y = 0 \\
Prec\colon & \forall xy.S(x) = S(y) \supset x = y \\
Succ\colon & \forall xy.x \neq y \supset S(x) \neq S(y)
\end{array}$$

4.3 Z Reduction

We also have the additional reduction,

$$(\pi_1(Z)\ Sn) \rhd \lambda a.\langle Sn, eq(Sn)\rangle$$

This corresponds to the proof normalization,

$$\cfrac{\cfrac{\forall x.x \neq 0 \equiv \exists y.S(y) = x}{\forall x.x \neq 0 \supset \exists y.S(y) = x}}{S(n) \neq 0 \supset \exists y.S(y) = S(n)} \qquad \cfrac{\cfrac{\cfrac{\forall x.x = x}{S(n) = S(n)}}{\exists y.S(y) = S(n)}}{S(n) \neq 0 \supset \exists y.S(y) = S(n)}$$

This is the Z reduction that removes the existentials from the main branch of proofs in the normalization process for proofs of existential sentences.