BRICS

Basic Research in Computer Science

Constructing Fixed-Point Combinators Using Application Survival

Mayer Goldberg

BRICS Report Series RS-95-35

ISSN 0909-0878 June 1995

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Constructing Fixed-Point Combinators Using Application Survival *

Mayer Goldberg
Computer Science Department
Indiana University †
(mayer@cs.indiana.edu)

June 26, 1995

Abstract

The theory of *application survival* was developed in our Ph.D. thesis as an approach for reasoning about application in general and self-application in particular. In this paper, we show how application survival provides a uniform framework not only for for reasoning about fixed-points, fixed-point combinators, but also for deriving and comparing known and new fixed-point combinators.

1 Introduction

Fixed-points have long been studied in the λ-calculus, and numerous fixed-point combinators are known, as well as many theorems about the existence of single and mutual fixed points. There does not seem to be, however, a single unified approach to characterise and to construct fixed points and fixed-point combinators. On the one hand, the behaviours of fixed-point combinators are characterised by Böhm trees, but on the other hand, they give us little information as to how to go about constructing a fixed-point combinator. Böhm's combinator [Barendregt 85, Item 6.5.4, Page 142], [Stoy 77, Problem 9, Page 77] allows us to generate an infinite sequence of distinct

*This work was completed while visiting BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).
†Bloomington, IN 47405, USA.

fixed-point combinators starting from a single fixed-point combinator. Although this approach relates Curry's and Turing's fixed-point combinators, it does not give us any clues as to how to construct a fixed-point combinator "from scratch." Klop captured some intuition about constructing fixed-point combinators, and his playful example [Barendregt 85, Problem 6.8.14, Page 149] can help in constructing a class of fixed-point combinators. As for the rest, we are on our own.

We wish for a unified approach that will allow us to generate various fixed-points and fixed-point combinators directly from an equational specification, as well as to explore the various fixed-point combinators and to characterise the differences between them. We believe that some progress towards a unified approach to fixed points is offered by the theory of *application survival*, developed in our Ph.D. thesis as a tool for reasoning about application in the λ-calculus [Goldberg 95]. It has been successful in dealing with questions that occur in untyped settings, such as questions related to self-application. The theory of application-surviving agents has been applied so far to fixed-point theory, Gödelisation in the untyped λ-calculus, and solving systems of equations.

In this paper, we show how application survival can be used to derive fixed points and fixed-point combinators from equational specifications, as well as to study the differences between various fixed-point combinators.

Section 2 provides a brief overview of application survival. In section 3 we present fixed points in light of application survival.

We shall assume a familiarity with the (untyped) $\lambda K \beta \eta$-calculus, λ-definability, and Böhm trees [Church 41], [Barendregt 85].

The following notation is assumed: $\ulcorner n \urcorner$ is the n-th Church numeral, S^+ is the successor, and P^- is the predecessor on Church numerals [Church 41, Pages 28-30]. **zero?** is the predicate testing for equality with $\ulcorner 0 \urcorner$. Throughout this paper we use the combinators $\mathbf{I} = \lambda x.x$ and $\mathbf{K} = \lambda xy.x$ [Church 41, Pages 10, 58]. We follow [Barendregt 85, Item 2.1.3 (i), Page 22] and let $\vec{x} = x_1, \ldots, x_n$, where n is implied by the context. Finally, $\mathfrak{x}$ and $\mathfrak{y}$ are the Gothic (Fraktur) letters x and y.

2 Agents

2.1 Preliminaries

2.1.1 DEFINITION: *Set of Agents.* Let $\mathcal{A}$ be a set of λ-expressions, and let $\mathcal{R}$ be a set of rules of the form $\{\langle \mathfrak{x}, f, \mathfrak{y} \rangle : \exists n, m \in \mathbb{N}, \mathfrak{x} \in \mathcal{A}^n, \mathfrak{y} \in \mathcal{A}^m, f \in \Lambda\}$.

We call $\mathcal{A}$ a *set of agents for* $\mathcal{R}$, if for all $\langle \mathfrak{x} = \langle x_1, \ldots, x_n \rangle, f, \mathfrak{y} = \langle y_1, \ldots, y_m \rangle \rangle \in \mathcal{R}$, we have $(x_1 \cdots x_n) = (f\ (y_1 \cdots y_m))$.

Note that by selecting $\mathcal{R} = \emptyset$ or choosing $\mathcal{R}$ such that $\forall \langle \mathfrak{x}, f, \mathfrak{y} \rangle \in \mathcal{R}$ we have $\mathfrak{x} = \mathfrak{y}$, etc., we can construct trivial sets of agents. This, however, does not trivialise the notion of an agent.

2.1.2 TERMINOLOGY: *Interaction.* The application of agents in $\mathcal{A}$ according to a rule in $\mathcal{R}$ is called an *interaction*, to set it aside from other kinds of application.

2.1.3 TERMINOLOGY: *Active Agent, Passive Agent.* Given an interaction $(A_1 \cdots A_n)$, the agent A_1 is called *the active agent.* An agent that is never used as an active agent is called a *passive agent.*

Interactions for non-trivial sets of agents are typically defined recursively, and hence some rules represent the *base case* of a recursion, while others represent an *induction step.* A rule of the form $\langle \mathfrak{x} = \langle x_1, \ldots, x_n \rangle, f, \mathfrak{y} = \langle y \rangle \rangle$, represents the base case of the recursion and corresponds to the statement that $(x_1 \cdots x_n) = (f\ y)$, from which we see that the application of agents on the left-hand side do not reduce to an application of agents; the process of applications of agents reducing to other applications of agents has terminated. Alternatively, a rule of the form $\langle \mathfrak{x} = \langle x_1, \ldots, x_n \rangle, f, \mathfrak{y} = \langle y_1, \ldots, y_{m>1} \rangle \rangle$ represents the induction step of the recursion, and corresponds to the statement that $(x_1 \cdots x_n) = (f\ (y_1 \cdots y_{m>1}))$, from which we see that the application on the left-hand side has resulted in an application of agents; The process of applications of agents resulting in applications of agents has continued. The resulting application $(y_1 \cdots y_m)$ may in turn terminate, by reducing to $(f'\ z)$, or it may continue, by reducing to $(f'\ (z_1 \cdots z_{k>1}))$, where $f' \in \Lambda$ is some λ-expression, and $z, z_1, \ldots, z_k \in A$ are agents, etc. We express the behaviour of the application on the left-hand side in the induction step by saying that application has been *survived.* The entire process is referred to as *recursive application-survival.*

2.2 Some Facts About Agents

2.2.1 NOTATION: The expression $\langle expression \rangle_{\langle variables \rangle}$ is meant to suggest that $\langle variables \rangle$ are free in $\langle expression \rangle$.

2.2.2 CONVENTION: Once we use subscripts to tag a given expression with some of its free variables, we will not tag that expression with other free variables. If new variable names appear as subscripts in any future occurrence of a given expression, then that expression will be understood to have been renamed. For example, if having mentioned M_x, we now mention

M_y in a similar context, then M_y should be taken as an abbreviation for $M_x[x := y]$.

2.2.3 Definition: [Barendregt 85, Problem 8.5.15, Page 184] *A Proper Combinator.* A combinator M is a *proper combinator* if and only if $M \equiv \lambda\vec{x}.\mathcal{P}_{\vec{x}}$ where $\vec{x} = x_1, \ldots, x_n$, and $\mathcal{P}_{\vec{x}} \in \{\vec{x}\}^+$.

The reader may have noticed by now that the kinds of applications allowed by our definition of a set of agents are of the form $(w_1 \cdots w_k)$, i.e. flat, left-associative applications, and may have wondered whether this restriction imposes limitations on what can be expressed in terms of agents. An arbitrary application of λ-expressions from $\{w_1, \ldots, w_k\}$ is some expression $M_{\vec{w}} \in \{\vec{w}\}^+$. The abstraction of $\vec{w}$ over M is the proper combinator $P \equiv \lambda\vec{w}.M_{\vec{w}}$ and specifies an *interaction pattern* of $w_1, \ldots, w_k$, i.e., how we would like these agents to be applied to each other. By combining P with the ordered k-tuple $\langle w_1, \ldots, w_k\rangle$, we obtain the $k+1$-tuple $\langle P, w_1, \ldots, w_k\rangle$, where the corresponding application $(P\ w_1 \cdots w_k)$ β-reduces to the desired application $M_{\vec{w}}$. Our definition is thus sufficiently general to allow for any application of $\langle w_1, \ldots, w_k\rangle$ in $\{\vec{w}\}^+$, either on the right-hand side or the left-hand side of the defining equation for a set of agents.

2.2.4 Observation: It should now become clear why we need to wrap the function f explicitly around the resulting interaction in the definition of an agent, as opposed to choosing a proper combinator that would permute one of the agents to the position of f. The latter approach would require us always to pass f along as one of the agents, and although this is occasionally used, as in for example Turing-style fixed-point combinators, this is not always desirable in general, especially in the case where f is constant over all interactions.

2.2.5 Convention: In light of the above discussion, and in order to simplify the description of an interaction, the proper combinator specifying the pattern of an interaction will remain implicit.

2.2.6 Definition: *Information Content.* The *information content* of an agent is the number of free variables it contains. An agent with n free variables is designated as an "n-information agent." Accordingly, if an agent is a combinator, it is a "0-information agent."

2.2.7 Theorem: *Any n-information agent, where $n > 0$, can be written as a 1-information agent.*

Proof: Let $A_{v_1,\ldots,v_n}$ be some n-information agent. We aggregate $v_1, \ldots, v_n$ into a *store* σ represented by the ordered n-tuple $[v_1, \ldots, v_n]$ with selectors $\pi_1^n, \ldots, \pi_n^n$. We now define a 1-information agent A_σ equivalent to $A_{v_1,\ldots,v_n}$

as follows:

$$\begin{array}{rcl} A_\sigma & = & ((\lambda v_1 \cdots v_n . A_{v_1,\ldots,v_n})\ (\pi_1^n\ \sigma) \cdots (\pi_n^n\ \sigma)) \\ & \longrightarrow_\beta & A_{v_1,\ldots,v_n}[v_1 := (\pi_1^n\ \sigma)] \cdots [v_n := (\pi_n^n\ \sigma)] \end{array}$$

Our theorem follows. ■

Active, 0-information agents act as permutors and associators, taking other agents as arguments in the interaction, and "rearranging" them. The behaviour of active n-information agents, on the other hand, might depend on its free variables: We regard free variables in an agent as information to be accessed and perhaps acted upon by the agent, as well as shared with other agents.

What principles guide the choice between 0-information and n-information agents? The answer depends on the information our agents need to access. We would prefer to use 0-information agents when the information changes from one interaction to another. When the information remains constant between interactions, it would be simpler to code the information into the agent (i.e., use an n-information agent), rather than to apply the agent to the same information repeatedly. In cases where an agent needs to know both variable as well as constant information, we combine the two approaches.

3 Fixed-Point Combinators

A fixed-point combinator is a combinator Φ with the property that for any λ-expression f, the following equality holds:

$$(\Phi\ f) = (f\ (\Phi\ f))$$

It is usually a simple matter to verify that an expression is indeed a fixed-point combinator (See Theorem 3.4.1); it is not as simple to actually derive one from the above equational specification.

As we establish in 3.2.4, any fixed-point combinator can be described in terms of a non-trivial interaction of agents. We offer below the derivation and analysis of the two best known fixed-point combinators.

3.1 Curry's and Turing's Fixed-Point Combinators

3.1.1 Example: *Curry's Y-Combinator.* Pick f. We want to find an agent u_f that satisfies the single rule $\langle\langle u_f, u_f\rangle, f, \langle u_f, u_f\rangle\rangle$, which specifies that $(u_f\ \underline{u_f}) = \underline{\underline{(f\ (u_f\ u_f))}}$. So:

$$\begin{aligned}
\mathcal{A} &= \{u_f\} \\
\mathcal{R} &= \{\langle\langle u_f, u_f\rangle, f, \langle u_f, u_f\rangle\rangle\}
\end{aligned}$$

We notice that the right-hand side of the equation can be written in terms of the argument to u_f by abstracting over it. We can now derive a definition for u_f from the underlined and double-underlined expressions: $\lambda \underline{u}.\underline{\underline{(f\ (u\ u))}}$. The application $(u_f\ u_f)$ is a fixed-point of f. In order to get Curry's fixed-point combinator we need only abstract over f:

$$\begin{aligned}
Y_{\text{Curry}} &= \lambda f.(u_f\ u_f) \\
&= \lambda f.((\lambda u.(f\ (u\ u))) \\
&\qquad\quad (\lambda u.(f\ (u\ u))))
\end{aligned}$$

The approach taken in constructing Curry's fixed-point combinator is to let u_f "know" about f, i.e. to let f be a free variable in the definition of u_f: When Y_{Curry} gets applied to f, u_f becomes *specialised* for this particular f. In order to generate a new interaction, u_f needs to have access to its own definition, which is then passed on to its single argument in its interaction.

3.1.2 Example: *Turing's Y-Combinator.* Pick f. We start with the system

$$\begin{aligned}
\mathcal{A} &= \{u, f, P\} \\
\mathcal{R} &= \{\langle\langle u, u, f\rangle, \mathbf{I}, \langle P, u, f\rangle\rangle\}
\end{aligned}$$

where P is the proper combinator $\lambda uf.(f\ (u\ u\ f))$, and is used to permute the agents. We have only a single rule to satisfy, so that $(u\ \underline{u\ f}) = \underline{\underline{(\mathbf{I}\ (P\ u\ f))}} = \underline{\underline{(P\ u\ f)}} = \underline{\underline{(f\ (u\ u\ f))}}$. We therefore need to have $P \equiv \lambda uf.(f\ (u\ u\ f))$. This last equation specifies, as was the case with Curry's Y-Combinator, a definition for u:

$$\begin{aligned}
u &= \lambda \underline{uf}.\underline{\underline{(\mathbf{I}\ (P\ u\ f))}} \\
&= \lambda \underline{uf}.\underline{\underline{(P\ u\ f)}} \\
&= \lambda \underline{uf}.\underline{\underline{(f\ (u\ u\ f))}}
\end{aligned}$$

Notice how the underlined and double-underlined expressions in the equation that defines the interaction correspond respectively to the underlined and double-underlined expressions in the definition of u. Now:

$$\begin{aligned} Y_{\text{Turing}} &= \lambda f.(u\ u\ f) =_\eta (u\ u) \\ &= ((\lambda uf.(f\ (u\ u\ f))) \\ &\quad\ (\lambda uf.(f\ (u\ u\ f)))) \end{aligned}$$

In this approach, the active agent u neither knows of its own definition, nor the definition of f, and receives them both as arguments (recall Observation 2.2.4). Since f is passed to u, f is treated as an agent, and in the derivation of Y_{Turing} it is indeed included in the set of agents $\mathcal{A}$. It is, however, explicitly excluded from functioning as an active agent by the one rule in $\mathcal{R}$.

3.2 Various Fixed-Point Combinators

Let us begin by considering how we derived the fixed-point combinators in the previous section. The fixed-point of an expression f is an expression of the form: $\varphi_f \longrightarrow_{\beta\eta} (f\ (f \cdots))$, and has no normal form. To generate an expression that converts to φ_f we came up with agents that upon interaction generated an application of f wrapped around an interaction of agents, which in turn would reduce in a similar way to the first interaction, et cetera. In order to derive Y_{Curry} and Y_{Turing}, we defined our agents to generate one application of f after each interaction. If we were merely interested in generating *a* fixed-point combinator, any number of applications of f would do. Both u_f and u need to be able to place f around some interaction, and so they both need access to f. These observations suggest how to combine the effects of Curry's and Turing's fixed-point combinator.

3.2.1 Example: *A Curry-Turing Fixed-Point Combinator.* Let us derive two variations, Φ_1 and Φ_2, of a third kind of fixed-point combinator, which *combines* both Curry- and Turing-style agents, with their different information-content properties: The Curry-style agent will "know" of the function f, and will interact with two other agents. The Turing-style agent will not "know" of f, and will therefore interact with three other agents. Thus, the pattern of the interactions will be as follows: An application of three agents reduces to an application of four agents, with f wrapped around it; The interaction of four agents reduces to an interaction of three agents, with f wrapped around it, etc. ad infinitum.

Pick f. We want to construct a set of agents $\mathcal{A} = \{u, v_f, f, P\}$ for the set of rules

$$\begin{aligned}\mathcal{R} = \{&\langle\langle u, u, v_f, f\rangle, \mathbf{I}, \langle P, u, v_f, f\rangle, \\ &\langle\langle v_f, u, v_f\rangle, f, \langle u, u, v_f, f\rangle\rangle\}\end{aligned}$$

where $P = \lambda uvf.(f\ (v\ u\ v))$ is a proper combinator we use to permute the agents in one of the interactions. We now have:

$$\begin{aligned}(u\ \underline{u\ v_f\ f}) &= \underline{\underline{(f\ (v_f\ u\ v_f))}} \\ (v_f\ \underline{u\ v_f}) &= \underline{\underline{(f\ (u\ u\ v_f\ f))}}\end{aligned}$$

We can now define u and v_f accordingly, as follows:

$$\begin{aligned}u &\equiv \lambda \underline{uvf}.\underline{\underline{(f\ (v\ u\ v))}} \\ v_f &\equiv \lambda \underline{uv}.\underline{\underline{(f\ (u\ u\ v\ f))}}\end{aligned}$$

As usual now, the underlined and double-underlined expressions in the set of equations correspond to the underlined and double-underlined expressions in the definitions of the agents u and v_f. Finally, we can define two fixed-point combinators, the first initially passing control to the agent u and the second initially passing control to the agent v_f:

$$\begin{aligned}\Phi_1 &\equiv \lambda f.(u\ u\ v_f\ f) \\ &\equiv \lambda f.((\lambda uvf.(f\ (v\ u\ v))) \\ &\qquad (\lambda uvf.(f\ (v\ u\ v))) \\ &\qquad (\lambda uv.(f\ (u\ u\ v\ f)))\ f)\end{aligned}$$

$$\begin{aligned}\Phi_2 &\equiv \lambda f.(v_f\ u\ v_f) \\ &\equiv \lambda f.((\lambda uv.(f\ (u\ u\ v\ f))) \\ &\qquad (\lambda uvf.(f\ (v\ u\ v))) \\ &\qquad (\lambda uv.(f\ (u\ u\ v\ f))))\end{aligned}$$

3.2.2 Example: *Klop's Fixed-Point Combinator.* The following problem is given in [Barendregt 85, Problem 6.8.14, Page 149] and is due to Klop. Given:

$$\begin{aligned}\pounds &= \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator) \\ \$ &= \underbrace{\pounds}_{\text{26 times}}\end{aligned}$$

Show that \$ is a fixed-point combinator.

It is not difficult to verify that \$ is indeed a fixed-point combinator, and to duplicate such an example. Let us put Klop's problem in the light of our theory of agents.

It should be clear from the Example 3.2.1, that our interactions can be of any size, and that the active agent can interact with several copies of itself. In that case, it has some choice as to which of its arguments to use in constructing the next interaction. The \$ combinator is in fact, a Turing-style fixed-point combinator, in which the active agent not only interacts with the function \$ is fixing (look at the position of r in the expression of $\pounds$), but also with 25 copies of itself, represented by $\{a, \ldots, z\} - \{r\}$. The application *thisisafixedpointcombinator* is nothing more than an application of the active agent t to 25 copies of itself (look at how \$ is defined) followed by the function r, which is what we would expect of a Turing-style fixed-point combinator. It is not a coincidence that r appears only in the last position of *this is a fixed point combinator* because otherwise we would have to guarantee that the above permutation of letters didn't result in r becoming an active agent.

Finally, Klop was able to write \$ as an application containing only $\pounds$ through an η-reduction similar to what was used in Example 3.1.2. This would not be possible in Curry-style fixed-point combinators, where any one of the agents could function as an active agent.

It should be clear now how similar fixed-point combinators of either the Curry or the Turing style are possible.

Let us consider a more challenging example:

3.2.3 Example: *Finding a Fixed Point of an Implicit Function.* We would like to define a combinator Φ, which takes f and g and returns the fixed-point $(f\ (g\ (f\ (g^2\ (f\ (g^3\ \cdots))))))$. Our agent needs to know of both f and g, which both remain constant, as well as the exponent of g, which will be represented as a Church numeral (recall: $(\ulcorner n \urcorner\ g) \longrightarrow_\beta g^n$), and which will be incremented from interaction to interaction. Therefore, our agent "knows" of f and g and is passed the exponent of g with each interaction. We define our interactions as follows:

$$\begin{aligned}
\mathcal{A} &= \{u_{f,g}\} \cup \{\ulcorner n \urcorner\ :\ n \in \mathbb{N}\} \\
\mathcal{R} &= \{\langle\ \mathfrak{x}_n = \langle u_{f,g}, u_{f,g}, \ulcorner n \urcorner \rangle, \\
&\qquad f_n = \lambda x.(f\ (\ulcorner n \urcorner\ g\ x)), \\
&\qquad \mathfrak{y}_n = \langle u_{f,g}, u_{f,g}, \ulcorner n+1 \urcorner \rangle \rangle : n \in \mathbb{N}\}
\end{aligned}$$

where $\mathcal{R}$ implies

$$\begin{aligned}
(u_{f,g}\ \underline{u_{f,g}\ \ulcorner n \urcorner}) &= (f_n\ (u_{f,g}\ u_{f,g}\ \ulcorner n+1 \urcorner)) \\
&= \underline{\underline{(f\ (\ulcorner n \urcorner\ g\ (u_{f,g}\ u_{f,g}\ \ulcorner n+1 \urcorner)))}} \\
&\longrightarrow_\beta (f\ (g^n\ (u_{f,g}\ u_{f,g}\ \ulcorner n+1 \urcorner)))
\end{aligned}$$

As usual, we derive a definition for $u_{f,g}$ by joining the underlined and double-underlined expressions in the above equation:

$$u_{f,g} = \lambda \underline{un}.\underline{\underline{(f\ (n\ g\ (u\ u\ (S^+\ n))))}}$$

Finally we now define Φ in terms of $u_{f,g}$ as follows:

$$\begin{aligned}\Phi &\equiv \lambda fg.(u_{f,g}\ u_{f,g}\ \ulcorner 1 \urcorner)\\ &\equiv \lambda fg.((\lambda un.(f\ (n\ g\ (u\ u\ (S^+\ n)))))\\ &\qquad\qquad (\lambda un.(f\ (n\ g\ (u\ u\ (S^+\ n)))))\\ &\qquad\qquad \ulcorner 1 \urcorner)\end{aligned}$$

Using a pre-existing fixed-point combinator, we would have had to begin with an explicit definition for the function being fixed and then apply to it this fixed-point combinator. Using agents, we simply capture the behaviour we are interested in, without ever bothering to have an explicit definition for the function we're fixing.

Finally, we extend our derivation of the above fixed-point combinators to *all* fixed-point combinators.

3.2.4 Theorem: *Any fixed-point combinator can be characterised in terms of the behaviour of application-surviving agents.*

Proof: Pick a fixed-point combinator Φ. We know that for all $g \in \Lambda$ we have $(\Phi\ g) = (g\ (\Phi\ g))$. From the left-hand side of the equation we know: $\Phi =_{\beta\eta} \lambda f.M_f$. Because of the Böhm tree of Φ (See [Barendregt 85, Page 217]) we know $M_g =_{\beta,\eta} (g^n\ (P\ Q\ g))$ for some integer n and some $P, Q \in \Lambda$. We construct our set of agents as follows: Let $\mathcal{A} = \{P, Q, g\}$, $\mathcal{R} = \{\langle \mathfrak{x} = \langle P, Q, g\rangle, f = g^n, \mathfrak{y} = \langle P, Q, g\rangle\rangle\}$. Our theorem follows. ■

3.3 Mutual Fixed-Point Combinators

3.3.1 Definition: *Mutual Fixed-Point Combinators.* The λ-expressions $\Phi_1, \ldots, \Phi_n$ are said to be *mutual fixed-point combinators* if for any λ-expressions $x_1, \ldots, x_n$ we have:

$$(\Phi_j\ x_1 \cdots x_n) = (x_j\ (\Phi_1\ x_1 \cdots x_n) \cdots (\Phi_n\ x_1 \cdots x_n)) \quad \text{for all } j \in \{1, \ldots, n\}$$

Multiple fixed-point combinators can be used to express mutual recursion:

3.3.2 Example: *Defining the Predicates* **even**?, **odd**? *with Mutual Fixed-Point Combinators.* Let Φ_1, Φ_2 be mutual fixed-point combinators. Let

$$\begin{aligned} E &= \lambda eon.(\mathbf{zero}?\ n\ \mathrm{T}\ (o\ (P^-\ n)))\\ O &= \lambda eon.(\mathbf{zero}?\ n\ \mathrm{F}\ (e\ (P^-\ n)))\end{aligned}$$

We can now define

$$\begin{array}{rcl}\textbf{even?} & = & (\Phi_1\ E\ O)\\ \textbf{odd?} & = & (\Phi_2\ E\ O)\end{array}$$

Mutual, or multiple fixed-point combinators[1] are derived in much the same way as we derived Y_{Curry} and Y_{Turing}, and similar consideration can be given to the information content of the agents involved. We can thus design agents for solving for mutual fixed-points which are of the Curry-style, the Turing-style, or any hybrid of these two approaches, as in Example 3.3.2.1. We offer below a derivation of a Curry-style multiple (n-ary) fixed-point combinator.

3.3.3 Example: *Deriving n Curry-style Mutual Fixed-Point Combinators.* Pick n λ-expressions $x_1, \ldots, x_n$. We need to find n agents $u_{\vec{x},1}, \ldots, u_{\vec{x},n}$ that satisfy the interaction schema:

$$\mathcal{R} = \{\langle\langle u_{\vec{x},j}, u_{\vec{x},1}, \ldots, u_{\vec{x},n}\rangle, \mathbf{I}, \langle P_j, u_{\vec{x},1}, \ldots, u_{\vec{x},n}\rangle : j \in \{1, \ldots, n\}\}$$

where P_j is given by

$$P_j = \lambda x_j u_1 \cdots u_n.(x_j\ (u_1\ u_1 \cdots u_n) \cdots (u_n\ u_1 \cdots u_n))$$

so that

$$\begin{array}{rcl}(u_{\vec{x},j}\ \underline{u_{\vec{x},1} \cdots u_{\vec{x},n}}) & = & \underline{\underline{(P_j\ u_{\vec{x},1} \cdots u_{\vec{x},n})}}\\ & = & \underline{\underline{(x_j\ (u_{\vec{x},1}\ u_{\vec{x},1} \cdots u_{\vec{x},n}) \cdots (u_{\vec{x},n}\ u_{\vec{x},1} \cdots u_{\vec{x},n}))}}\end{array}$$

As usual, we now derive a definition for $u_{\vec{x},j}$ by joining the underlined and double-underlined parts of the previous equation:

$$u_{\vec{x},j} = \lambda \underline{u_1 \cdots u_n}.\underline{\underline{(x_j\ (u_1\ u_1 \cdots u_n) \cdots (u_n\ u_1 \cdots u_n))}}$$

Finally, we define the j'th mutual fixed-point combinator as follows:

$$\Phi_j = \lambda x_1 \cdots x_n.(u_{\vec{x},j}\ u_{\vec{x},1} \cdots u_{\vec{x},n})$$

[1] See [Barendregt 85, 6§5] for a complete treatment of mutual fixed-points.

3.4 Böhm-style Fixed-Point Combinators

A fixed point x of an expression M must satisfy, by definition, $(M\ x) = x$. The equality predicate is normally taken to be $=_{\beta\eta}$ and so what the above fixed-point equation means is that there exists an expression y, and two finite ordinals m, n, such that y is m $\beta\eta$-reductions away from $(M\ x)$ and n $\beta\eta$-reductions away from x:

$$\begin{tikzcd} (M\ x) \arrow[dr, "\beta\eta"] & & x \arrow[dl, "\beta\eta"'] \\ & m\ y\ n & \end{tikzcd}$$

The finiteness of m and n is built into our notion of equality. This finiteness carries on to the λ-acceptability of the set of all fixed-point combinators:

3.4.1 Theorem: *The set of all fixed-point combinators is λ-acceptable.*

Hint of Proof: Pick a fixed-point combinator Φ. We know that for any λ-expression x we have $(\Phi\ x) = (x\ (\Phi\ x))$. So $\Phi = \lambda x.(x\ (\Phi\ x))$ (by rule ξ followed by rule η). We know there exist finite ordinals m, n such that

$$\begin{tikzcd} \Phi \arrow[dr, "\beta\eta"] & & \lambda x.(x\ (\Phi\ x)) \arrow[dl, "\beta\eta"'] \\ & m\ y\ n & \end{tikzcd}$$

By enumerating all pairs of ordinals, and for each pair $\langle m_j, n_j\rangle$ reducing Φ and $\lambda x.(x\ (\Phi\ x))$ m_j and n_j β-reductions respectively, and comparing the two (possibly very large) respective sets of λ-expressions, we can arrive at the particular $\langle m, n\rangle$ for which syntactic equality holds. It follows that the set of all fixed-point combinators is λ-acceptable. ■

If we remove the restriction that m, n be finite, our equality predicate becomes the $=_{\eta,\text{Böhm}}$ predicate which is satisfied when two expressions have the same Böhm tree modulo any number of η-reductions. What we lose by relaxing this requirement that m and n must be finite is the λ-acceptability of the set of fixed-point combinators, which is not needed for the purpose of doing general recursion.

In the following example, we construct $Y_{\text{Böhm,Curry}}$, a *semantic but not syntactic* fixed-point combinator of the Curry-style. $Y_{\text{Böhm,Curry}}$ will not satisfy the equation $(Y_{\text{Böhm,Curry}}\ x) = (x\ (Y_{\text{Böhm,Curry}}\ x))$, however $(Y_{\text{Böhm,Curry}}\ x)$ and $(x\ (Y_{\text{Böhm,Curry}}\ x))$ have the same Böhm tree. $Y_{\text{Böhm,Curry}}$ is sufficient for expressing general recursion.

3.4.2 Example: *The Construction of a Curry-Böhm-Style Fixed-Point Combinator.* In all the previous examples of fixed-point combinators, the interaction of agents resulted in an identical interaction. One way to make sure that a Böhm-style fixed-point combinator Φ fails to satisfy $(\Phi\ M) = (M\ (\Phi\ M))$, i.e. that there is no λ-expression N such that $(\Phi\ M) \longrightarrow^m_{\beta\eta} N$ and $(M\ (\Phi\ M)) \longrightarrow^n_{\beta\eta} N$, for any integers m and n, is to guarantee that the agent interactions always reduce to a different interaction. One approach is to select agents satisfying the following interaction schema:

$$(u\ u\ \ulcorner n \urcorner) = (\ulcorner n \urcorner\ f\ (u\ u\ \ulcorner n+1 \urcorner))$$

Our set of agents is defined as $\mathcal{A} = \{u\} \cup \{\ulcorner n \urcorner : n \in \mathbb{N}\}$, and our *rule schema* is defined as $\mathcal{R} = \{\langle\langle u, u, \ulcorner n \urcorner\rangle, f^n, \langle u, u, \ulcorner n+1 \urcorner\rangle\rangle : n \in \mathbb{N}\}$. We now need to define u to satisfy the above schema:

$$\lambda un.(\ulcorner n \urcorner\ f\ (u\ u\ (S^+\ n)))$$

And accordingly:

$$\begin{aligned} Y_{\text{Böhm,Curry}} = \lambda f.((&\lambda un.(\ulcorner n \urcorner\ f\ (u\ u\ (S^+\ n)))) \\ &(\lambda un.(\ulcorner n \urcorner\ f\ (u\ u\ (S^+\ n)))) \\ &\ulcorner 1 \urcorner) \end{aligned}$$

It is clear that many other Böhm-style fixed-point combinators are obtainable through simple variations on this scheme. For example, the following is a Turing-Böhm-style fixed-point combinator:

$$\begin{aligned} Y_{\text{Böhm,Turing}} = ((&\lambda unf.(\ulcorner n \urcorner\ f\ (u\ u\ (S^+\ n)\ f))) \\ &(\lambda unf.(\ulcorner n \urcorner\ f\ (u\ u\ (S^+\ n)\ f))) \\ &\ulcorner 1 \urcorner) \end{aligned}$$

It should also be clear that the set of all Böhm-style fixed-point combinators is *not* λ-acceptable.

4 Conclusion

While results on fixed points abound, little is known about how these results were obtained. It is trivial to verify that Y_{Curry} and Y_{Turing} are indeed fixed-point combinators, but mere verification leaves one unsatisfied: How does one come up with such expressions on one's own?

The theory of application survival [Goldberg 95] provides us with a unified approach to reasoning about fixed-points: Not only can we derive the classical, well-known fixed-point combinators, but we can also construct new ones. Application survival can be used to characterise any fixed-point combinator in terms of how it accesses the information it needs, how it sets up new interactions, and what function gets applied in the interaction. This characterisation is specified equationally. It not only allows us to study the differences between various fixed-point combinators, but also to derive any specific fixed-point combinators.

Acknowledgements

I am grateful to BRICS[2] for hosting me this summer and for providing a stimulating environment. Thanks are also due to Olivier Danvy, Daniel P. Friedman, and Larry Moss for their comments and encouragement.

The diagrams of Section 3 were drawn with Kristoffer Rose's XY-pic package.

References

[Barendregt 85] Hendrik P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1985.

[Church 41] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[Goldberg 95] [Forthcoming] Mayer Goldberg. *Recursive Application-Survival in the λ-Calculus.* Ph.D. Thesis, Department of Computer Science, Indiana University, December 1995.

[Stoy 77] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[2] Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

Recent Publications in the BRICS Report Series

RS-95-35 **Mayer Goldberg.** ***Constructing Fixed-Point Combinators Using Application Survival*****. June 1995. 14 pp.**

RS-95-34 **Jens Palsberg.** ***Type Inference with Selftype*****. June 1995. 22 pp.**

RS-95-33 **Jens Palsberg, Mitchell Wand, and Patrick O'Keefe.** ***Type Inference with Non-structural Subtyping*****. June 1995. 22 pp.**

RS-95-32 **Jens Palsberg.** ***Efficient Inference of Object Types*****. June 1995. 32 pp. To appear in *Information and Computation*. Preliminary version appears in *Ninth Annual IEEE Symposium on Logic in Computer Science*, LICS '94 Proceedings, pages 186–195.**

RS-95-31 **Jens Palsberg and Peter Ørbæk.** ***Trust in the λ-calculus*****. June 1995. 32 pp. To appear in *Static Analysis: 2nd International Symposium*, SAS '95 Proceedings, 1995.**

RS-95-30 **Franck van Breugel.** ***From Branching to Linear Metric Domains (and back)*****. June 1995. 30 pp. Abstract appeared in Engberg, Larsen, and Mosses, editors, *6th Nordic Workshop on Programming Theory*, NWPT '6 Proceedings, 1994, pages 444-447.**

RS-95-29 **Nils Klarlund.** ***An $n \log n$ Algorithm for Online BDD Refinement*****. May 1995. 20 pp.**

RS-95-28 **Luca Aceto and Jan Friso Groote.** ***A Complete Equational Axiomatization for MPA with String Iteration*****. May 1995. 39 pp.**

RS-95-27 **David Janin and Igor Walukiewicz.** ***Automata for the μ-calculus and Related Results*****. May 1995. 11 pp. To appear in *Mathematical Foundations of Computer Science: 20th Int. Symposium*, MFCS '95 Proceedings, LNCS, 1995.**

RS-95-26 **Faith Fich and Peter Bro Miltersen.** ***Tables should be sorted (on random access machines)*****. May 1995. 11 pp. To appear in *Algorithms and Data Structures: 4th Workshop*, WADS '95 Proceedings, LNCS, 1995.**

RS-95-25 **Søren B. Lassen.** ***Basic Action Theory*****. May 1995. 47 pp.**