UPSILON: Universal Programming System with Incomplete Lazy Object Notation

Brian Postow, Kenneth Regan and Carl H. Smith
bpostow@cs.umd.edu, regan@cse.buffalo.edu, smith@cs.umd.edu

September 6, 2000

Abstract

This paper presents a new model of computation that differs from prior models in that it emphasizes data over flow control, has no named variables and has an object-oriented flavor. We prove that this model is a complete and confluent acceptable programming system and has a usable type theory. A new data synchronization primitive is introduced in order to achieve the above properties. Subtle variations of the model are shown to fall short of having all these necessary properties.

1 Introduction

Different models of computation have spawned different paradigms of programming languages. Their differences in focus can have great effects on the style and uses of the resulting languages. One of the most important differences is the model's place in the program/data continuum. This continuum arises because there is no rigid distinction between program and data. The universal Turing machine exemplifies this as it is a program that takes another program as input (data) and then runs it (treats it like a program). The stored-program computer realizes the interchangeability of program and data concretely. Nevertheless, every model stakes out a position in how "program" and "data" are treated, and this position greatly affects its design.

The λ-calculus is a simple model of computation because it lies at one end of this continuum. It only has functions. There are no inherent values in this model— rather, functions can be treated as values. Being completely on the program end of the continuum determines the style of programming in languages based on this model. It also has a significant effect on the techniques that are used to prove things about the languages and the programs written in them. Indeed, they are said to represent the functional paradigm of programming.

In the middle of the continuum lie machines in the random access (RAM) model. RAM machines obviously have data, which they store in registers, and have a separate program with flow controled by goto statements. This has as significant an effect on the style of programming in imperative languages, and the proofs on those languages, as the extremism of the λ-calculus does on functional programming.

The other end of the continuum should naturally be held down by *data-driven programming*. However, currently there seems to be no corresponding fundamental model at the level of λ-calculus or Turing or RAM machines. (We will consider the Abadi-Cardelli model [AC96a] and related record calculi later.) We were motivated to look at this problem in relation to object-oriented programming (OOP) languages. OOP is unique among language paradigms in that it did not stem from an already existing model of computation. Quite the opposite, it was founded out of a combination of techniques from artificial intelligence, imperative programming, systems, and many other areas of computer science. Just as "functional" programming is focused on the functions, i.e. on programs, we feel that "object-oriented" programming should spring organically from objects, i.e. data, themselves. Thus by creating a model of computation based on data with little or no control flow, we seek to fill two niches simultaneously: a model at the data-driven end of the continuum and a foundation for OOP. Our model begins with a low-level representation of objects, but attains enough of an object-oriented flavor to merit borrowing freely from OOP terminology in our description of it.

Before we determined how we were going to effect computation with no program, we scrutinized what our data objects would look like. In trying to model object-oriented programming, we worked from the low-level representation of objects as nestable records with named fields. We simplified this by removing the names to leave nestable tuples i.e. tuples whose fields can be objects. In our model there is no distinction between "data field" and "method field"—though some fields can be objects that perform some form of computation as methods do. Finally, in keeping with the goal of modeling OOP languages, the methods should have some way to access the object in which they reside. Normally this is done with either a syntactic keyword (e.g. `this` or `self`) or through a hidden argument to the method. We will show that this has in fact never been necessary. Through appropriate use of the recursion theorem [Kle38] we will find that access to the self object was inherently available all the time.

Once we had a definition of objects, we had to figure out how to perform computation on them without any program. The concept of "incomplete objects" was our solution to this problem. An incomplete object is an object with "holes" in it. A hole (here called a *port* and denoted by $\ominus$) can be a field or part of a larger term. The concept of of computation is taking an incomplete object and completing it by filling in the ports with other objects. A method is therefore any field of an object that can perform some sort of computation, i.e. that has holes that can filled themselves. This concept of objects with holes is very similar to the concept of contexts as described in [Bar84], and studied by, among others, [LF96], [Tal93], [San98], [ACCL91], and [HO98].

These insights lead to a model of computation that is Turing-complete and confluent, but that has no control flow, and is completely data-flow. In addition to its novelty, this is interesting in three distinct lights. Toward our initial goal, once a sufficiently complex typing system with subtyping is added it may yield a fundamental model for OOP. Second, our model may have useful things to say about *contexts*, as mentioned above. Our model is effectively a context calculus, with nothing other than contexts to compute with. Third, ours is effectively also a data-flow language. As such it may be interesting as a different view of how data flows into contexts.

2 Prior Work

Models of computation have been a vital area of theory of computation research since 1936, when Turing [Tur36] introduced Turing machines, the same year that Church [Chu36] published the λ-calculus. However, there have been very few completely novel models of computation in recent times, the π calculus [MPW92] being one such model.

Each of new model of computation was created in order to study a specific area of mathematics or computer science. For example, the λ-calculus was introduced to study formal arithmatic, and the π calculus was introduced to study parallel processes. We intend to use our new model of computation in order to elucidate three areas of theory of computer science: the formalisms beneath object-oriented programming (our original aim), contexts, and data-flow computation.

Object-oriented programming has been studied by many other researchers, but none of them have introduced a truly novel model of computation in their work. Fisher et. al. [FHM94] define a modification of the λ-calculus designed to represent objects. They add records with object extension and field replacement, and develop a type theory for their model. They model *self* as the implicit parameter in every method invocation. Bono et. al. [BBL96] [BBDCL97] extended this, looking more closely at the typing of object extension. Liquori and Castagna [LC96] added explicit types in order to clean up the type system. Abadi and Cardelli [AC96b] developed model similar to Fisher et. al.'s called the ς-calculus in greater depth. They provide deeper description of the typing of inheritance and other pitfalls of object-oriented programming. Kim Bruce [Bru94] designed a whole programming language called TOOPL. He made a detailed denotational semantics in order to study the meaning of inheritance and subtyping which he made into separate notions. All of these papers are still modifications of the λ-calculus. Pierce and Turner [PT94] develop a similar model, except that they only represent *self* at the class level, not at the object level. Their focus is on the type theory of subtyping and on inheritance using existential types. The object-oriented portion of our paper focuses on a new representation of *self* via the recursion theorem, and defines a model of computation that is not directly based on the λ-calculus.

The concept of a *context* is very similar to the founding idea of our model. A context is a term with a hole, a missing part that must be filled in later. Contexts are also almost always studied within the realm of the λ-calculus, where they are used to study variable capture [LF96], program transformations [Tal93], and certain types of operational semantic definitions [San98]. One of the main difficulties in studying contexts is that typically they aren't confluent. Depending on the order of reduction, an unbound variable may be captured by two different bindings. In addition, contexts destroy α-equivalence— if one changes the name of a binding that captures a variable in the hole, the variable is no longer captured. Talcott [Tal93] used contexts in order to study binding structures for term re-writing systems and theorem provers. She designed an algebraic system for manipulating contexts. However, this is for use at the meta-level of a theorem prover, and was meant for computer rather than human use. Abadi et. al. [ACCL91] studied contexts in the setting of trying to make the meta-level issue of substitution an explicit part of the lambda-calculus itself. They don't study the contexts in and of themselves as much as how they are useful for studying substitutions. Lee and Friedmann [LF96] also proposed a calculus including contexts. In their calculus, lambda-bound variables and hole variables are in different name spaces, and they reduce contexts to a different type of binding operator that allows name capture. Their calculus can then be "compiled" down to the normal lambda-calculus. Hashimoto and Ohori [HO98] study first class contexts, and develop a type theory for a language in which normal lambda terms are merely a special case of contexts. They prove a confluence result, but they don't study the hole filling procedure explicitly.

Almost all context research either deals only with terms with one hole (or many holes to be filled with the same argument), or deals with named holes, thus turning contexts into just another binding construct as in [LF96]. We, on the other hand, have holes as our only method of data transfer, and we have no names in our model, so we are not reducing contexts to the lambda-calculus. However, even without names, because of our object-oriented view, we can still study variable capture through use of the concept of self.

Data-flow computation can be viewed as the dual of standard von Neumann control flow computation. In von Neumann models (RAM machines, flow charts, or most computers for example) there is a global, persistent memory that can be accessed by any instruction. Throughout the computation, the locus of control changes, flowing through the program. These programs make heavy use of assignments to global data stores. On the other hand, data-flow computations, as first described in [Ada68] and [Rod69], have no data store, but instead, have data constantly moving through the program. Instead of having a single locus of control flow, computation executes commands (often in parallel) whenever all of their arguments become available (i.e., the appropriate data has flowed in). In addition, data-flow models are usually applicative, with no assignment statements. The reason for this is that the primary use of data-flow models is for modeling parallelism. In order to parallelize a program, one must first make sure that no two threads executing in parallel can interfere with each other. Applicative programs guarantee a certain amount of safety in this regard. Because of the non-linearity inherent in data-flow (any command may execute at any time, as long as all of its arguments are present), these models are typically represented as graphs where a vertex represents a command, and edges represent the flow of data. However, there are textual languages such as VAL (described in [AD79]) and Id (described in [AGP78]).

3 The Model

Besides the guiding principle of being a fundamental model for data-driven programming, we recognized four principles for the final model. We refer to "program" more generally than the contrast drawn in Section 1.

1. The model should have programs for all the computable functions (Turing completeness)
2. The model should have programs that effectively manipulate other programs. For example, there should be a program that takes two programs as input and produces a third that computes the composition of the programs serving as input.
3. The model should be confluent.
4. The model should have a manageable type theory.

Principles 1 and 3 guarantee that the model really is a complete model of computation worth studying. Principle 2 guarantees a whole host of other common program manipulation tools, like recursion in its most general forms. Principle 4 allows us to investigate the object-oriented flavor of the model more carefully.

We distinguish between *base objects*, whose top-level structure has been determined up to the number and current status of fields, and *proper terms*, whose top-level field structure has not yet been determined, but which have other structure that can be operated on. Base objects are notated as tuples with outermost square brackets [...], and every other object is a proper term.

The two simplest elements, the empty base object [ ] and the empty *port* $\ominus$, illustrate our ideas. The former is a completed empty object, while the latter is a hole that can be filled with any object. Indeed, it is important for our interpretation that a $\ominus$ is *not* an entity, but rather denotes the *absence* of an entity. It is a place that an entity can go. A term such as $\pi_1(\ominus)$, however, is an entity.

There are four operators that are used to build more complicated objects. The first is tupling, taking several objects and making them fields in a larger object. If A, B, and C are objects, then $[A, B, C]$ is a tuple with them in its three fields. The next is the usual append, denoted by @ as in ML, that takes two objects and merges them by taking the fields of one followed by the fields of the other in the order of appearance. Fields of an object are extracted via a family of projection operators π_i. So $\pi_i(A)$ evaluates to (reduces to) the i^{th} field of the object A. If A should have fewer than i fields, then a run time error occurs. More importantly, if there are enough fields in A, and the i^{th} field is a port, then the computation *stalls*. Intuitively, the i field has yet to be specified, so it cannot be extracted.

The fourth operator, denoted by infix $\Upsilon^\dagger$, plugs a list of objects into the *pluggable ports* of another object. All ports are pluggable except those inside an object to the left of another Υ or those inside a base object that is inside another base object. Explanations for why these latter two kinds of ports are *shielded* are given below, and pluggable ports are defined formally in Appendix A.1. The number of pluggable ports is called the *valence* of the target object.

Plugging is done by textual substitution of the port by the corresponding object in the argument list, taking ports and arguments in left-to-right order. For example, if $A = [\ominus, \ominus]$ and B, C are any objects (not ports), then $A \ \Upsilon \ B \ C$ yields $[B, C]$.

If, as in the example above, A has exactly two pluggable ports, then clearly B goes into the leftmost such port and C goes into the other one. However, what happens if A has more than two ports? We considered having B and C plug the leftmost two ports, leaving the rest unfilled. Since information could never be replicated otherwise, this required having operators Υ^k, $k \geq 2$, that duplicate the argument list k times. For instance, $A \ \Upsilon^2 \ B \ C$ would reduce as $A \ \Upsilon \ B \ C \ B \ C$. The resulting model satisfies Principles 1–3, but as we prove in Section 8.1, it violates 4 because the type notation for terms becomes exponential in their size. The intuitive reason is that if an object of valence v doesn't know how many ports will be filled with each plug, the type system needs to keep track of all v possibilities of ports being unfilled, multiplying the size of type expressions by factors of v that spiral exponentially.

The natural alternative is the "cycling rule" by which the argument list is replicated enough times to fill all pluggable ports, discarding any leftover arguments from the last cycle. For instance, if A has three ports in $A \ \Upsilon \ B \ C$, then B goes into the first and third ports and C into the second, with the second C discarded. It is consistent with this to treat the case where the argument list has *more* than v arguments by plugging the first v arguments and discarding the rest, so that $A \ \Upsilon \ B \ C \ B \ C$ reduces as $A \ \Upsilon \ B \ C \ B$, rather than as a run-time error. The type theory becomes tamed, adhering to Principle 4, and Principles 1 and 3 remain intact. However, Principle 2 now fails, because the model no longer has an effective composition operator, and thus fails to be an *acceptable programming system* (APS) as defined by Rogers [Rog58] (see also [Smi94]). This is curious because all previous models that are Turing complete but not APS, most notably that of Friedberg [Fri58], have been deliberately constructed that way by means regarded as "artificial," such as by priority and simulation arguments. We are not the first researchers to create such a model, but perhaps the first to find this combination of properties arise naturally and by accident. This is proved in Section 8.2

The intuitive reason for the second problem is that filling all ports by cycling arguments leaves no way to curry or do partial closures. Our solution introduces a second kind of port, called a *locked port* and denoted by $\oslash$. The difference is that the first attempt to fill a locked port only serves to unlock the port, turning it into a regular port. The data directed to a locked port is discarded. For example, $[\ominus, \ominus, \oslash] \ \Upsilon \ B \ C$ and $[\ominus, \ominus, \oslash] \ \Upsilon \ B \ C \ D$ both yield $[B, C, \ominus]$. Locked ports do count toward the valence and are said to be plugged, by B or D in these two instances. All locked ports in examples in this paper are rightmost and used only for timing purposes, but this restriction is not required by the model. It is worth remarking that our type

†The Greek letter Υ is used because it connotes one thing becoming many, and spreading out, plus, we feel a little sorry for the letter since it isn't used much. Its name supplies the acronym used as the title of this paper.

system (Section 7) is not inconvenienced by uncertainty about the source *position* of a plugging argument, where e.g. the fourth port of $[\ominus, \ominus, \ominus, \ominus]\ \Upsilon \ldots$ will receive the first element of a one- or three-argument list, the second of a two-argument list, and the fourth of a four-argument (or longer) list. The point is that a pluggable $\ominus$ knows it will receive data when the Υ is expanded, while a locked port $\oslash$ knows it will discard data.

Combining locked and unlocked ports with the cycling rule yields a model that satisfies all four design principles above. This is the model that we call UPSILON, and that we describe in the rest of this paper. The syntax of UPSILON is simple and context-free:

$$\texttt{obj} ::= \ominus \ \mid\ \oslash \ \mid\ [\texttt{obj list}] \ \mid\ \pi_i(\texttt{obj}) \ \mid\ \texttt{obj@obj} \ \mid\ \texttt{obj}\ \Upsilon\ \texttt{obj list},$$

where i is an arbitrary positive integer. To help tell base objects apart from lists to the right of an Υ, we write the latter without commas. Under the cycling rule it is formally unnecessary to use superscripts "Υ^k" to tell how many times the argument list is cycled, but we do so for reader clarity.

3.1 Operation of the Model

An object is called *reduced* if it cannot reduce any further. This can happen in two ways: by being a base object whose fields are reduced, or by being a term whose operation is inhibited by incompleteness of some component objects. The following formal inductive definition of reduced objects shows that the latter case ultimately involves an operator being "paused" because some left or right argument is a $\ominus$.

Definition 3.1.
(a) A stand-alone port $\ominus$ or $\oslash$ is reduced.

(b) A base object is reduced if every field is reduced.

(c) A term $\pi_i(A)$ is reduced if A is reduced and not a base object, or if A is a reduced base object whose ith field is a port.

(d) A term $A@B$ is reduced if A and B are reduced and at least one of them is not a base object.

(e) A term $L\ \Upsilon\ RHS$ is reduced if L and all elements of RHS are reduced, and either L or some element of the list RHS is a port.

To motivate (c) and (d), clearly $A@B$ and $\pi_i(A)$ simply cannot operate when A or B is a proper term or a port. These are not error conditions, because the terms or ports may later evaluate to or be filled with base objects. The reason why a term such as $\pi_2([A, \ominus, B])$ is not allowed to reduce to a stand-alone $\ominus$ is that a $\ominus$ by itself denotes an *absence*, and one cannot create a pure absence from an entity. The explanation for (e) is similar—if L is a port then there is no entity to plug into, while a port in RHS denotes the absence of a required argument. Thus for example $\ominus\ \Upsilon\ A$ does *not* reduce to A, nor $[\ominus, \ominus]\ \Upsilon\ A \ominus$ to $[A, \ominus]$. We motivate the restriction in (e) further below in Section 3.3. If an object is not reduced, then it has a sub-term that falls into one of the following cases, in which we follow the standard practice of saying that the sub-term/operator involved is a *redex*.

Definition 3.2.
(i) $\pi_i([A_1, \ldots, A_m])$ reduces in one step to A_i, provided $0 < i \le m$ and A_i isn't a port.

(ii) $[A_1, \ldots, A_k]@[B_1, \ldots, B_m]$ reduces in one step to $[A_1, \ldots, A_k, B_1, \ldots, B_m]$, preserving the ordering of the fields.

(iii) $L\ \Upsilon\ R_1\ \cdots\ R_m$, where L is reduced, and neither L nor any R_i is a port, reduces in one step to $L^{\underbrace{R_1,\ldots,R_m,\ldots,R_1,\ldots,R_m}_{n}}$. Here L^{RHS} represents the object created when one plugs the elements of the list RHS into L, and $n = \lceil v/m \rceil$ where v is the valence of L. Most of the rest of this subsection is an informal description of L^{RHS}. A formal inductive definition in a slightly expanded context is given in Appendix A.1.

An object may of course have more than one redex. In Appendix A.2 we extend this to a formal relation that applies to all objects, not just to redexes, and we prove the confluence of this relation in Section 3.2.

Note the restriction in (iii) that L be reduced. The technical reason is that reducing L can change the number and identity of "pluggable ports" as defined next, destroying confluence. Under the analogy between Υ and application in the λ-calculus, this is like saying the body of a function must be in final form before it can be called, and might be called "semi-eager" evaluation. We contend that this does not upset practically important cases of lazy evaluation or trivialize the confluence property to follow. This said, we can now define the *list of accessible ports* (or "pluggable ports") of an object L as follows.

The list $pp(L)$ of pluggable ports in an object L includes all ports in L *except*

(i) Ports in a base object that is part of another base object,

(ii) Ports inside an object to the left of an Υ.

Regarding (ii), in $(\ominus\ \Upsilon\ R_1)\ \Upsilon\ R_2$, the leftmost $\ominus$ is *not* pluggable by R_1—indeed the inner Υ cannot reduce—but is pluggable by R_2. This is because a stand-alone port is not an object and cannot be plugged in isolation — see motivations below, in Section 3.3. Additionaly, recall from data-flow models as described in Section 2 how they motivate our rule that (e.g.) $R_1\ \Upsilon\ R_2\ \ominus$ cannot evaluate, because the second argument is unavailable. Although data-flow models by and large do not have the situation where an operator is unavailable (since it is a command not a data object), we apply the intent symmetrically by prohibiting $\ominus\ \Upsilon\ R_1$ from evaluating because the target object is unavailable. To illustrate, using subscripts to distinguish between ports, we have:

$$
\begin{aligned}
pp(\ominus\ \Upsilon\ A\ B) &= \ominus, pp(A), pp(B);\\
pp(\pi_1(\ominus)\ \Upsilon\ A\ B) &= pp(A), pp(B);\\
pp([\pi_1(\ominus_a), [A, \ominus_b], \oslash_c @ \pi_1[\ominus_d], \ominus_e]) &= \ominus_a, \oslash_c, \ominus_e.
\end{aligned}
$$

Finally, to define L^{RHS}, the pluggable ports $pp(L)$ in L are determined from these rules, and their number is called the valence of L, as mentioned above. The ports are substituted by the corresponding elements of *RHS*, in left-to-right order, cycling if necessary. The result is always a valid object. For any objects A, B, C, recalling that $\ominus$ and $\oslash$ are not objects, we have for example:

- $[\ominus, [A, \ominus], \ominus]\ \Upsilon\ B\ C$ evaluates to $[B, [A, \ominus], C]$;
- $[\ominus, \ominus, \ominus, \ominus]\ \Upsilon\ B\ C$ equals $[B, C, B, C]$, not $[B, B, C, C]$;
- $[\ominus, \ominus, \ominus, \ominus, \ominus]\ \Upsilon\ B\ C$ equals $[B, C, B, C, B]$; and
- $[\ominus, \ominus]\ \Upsilon\ A\ B\ C$ equals $[A, B]$.

In the first example, the port to the right of A is not a (top-level) field. The manner of collating arguments in the second is natural for multi-ary functional composition. In the third, the last cycle does not need to be complete, so the number of arguments need not evenly divide the number of pluggable ports. In the fourth, where the number of arguments is smaller than the number of pluggable ports, the remaining arguments are lost. This is not as strange as it might first seem. There is information loss in many aspects of computing, including our own projection functions which discard the entire object except for the field which is being projected out.

The above rules can be summarized by saying that to reduce $L\ \Upsilon\ RHS$, (i) L must be reduced, (ii) L and all elements of *RHS* must not be a port, (iii) one cannot cross two "["s to plug into a base object within a base object, and (iv) one cannot plug into a sub-term before an Υ unless that sub-term is a port. Note that the interpretation of a solitary port as denoting absence rather than an entity dictates much of these rules.

Plugging a locked port works in exactly the same way, except that instead of actually filling the hole, it discards the argument and unlocks the port, leaving $\ominus$. For example, for any objects A and B:

- $[\ominus, \oslash] \Upsilon A B$ reduces to $[A, \ominus]$, as B unlocks the second port; and
- $(\ominus \Upsilon \pi_1[\ominus] \oslash) \Upsilon A B$ reduces to $A \Upsilon \pi_1[B] \ominus$, as A unlocks the locked port, and then to $A \Upsilon B \ominus$, when the rightmost $\ominus$ inhibits any further reduction.
- For comparison, $(\pi_2[\oslash, \ominus,] \Upsilon \pi_1[\ominus] \ominus) \Upsilon A B$ reduces to $(\pi_2[\oslash, \ominus] \Upsilon \pi_1[A] B)$, because the sub-term to the left of the Υ is not a naked port, we can't plug directly into it. Then one can take the "long road" of expanding the π_1 redex to get $(\pi_2[\oslash, \ominus] \Upsilon A B)$ and then $\pi_2[\ominus, B]$, or the "short road" of expanding the Υ first to get $\pi_2[\ominus, B]$, both ultimately reducing to B.

3.2 Confluence

Now we establish that the resulting calculus has the important properties of confluence and uniqueness of reduced forms. The proof is in the Appendix because its technical machinery, including a formal inductive definition of L^{RHS} and the extension of the "reduces in one step" relation from redexes to whole objects, is not needed in the rest of the paper.

Theorem 3.1.
(a) UPSILON *is confluent, i.e. if an object A is reducible to an object B and to another object C, then there is an object D such that B reduces to D and C reduces to D.*

(b) If A reduces to B and B reduces to A, then there is an object C distinct from A, B such that A and B both reduce to C.

(c) If A reduces to B and B is reduced, then B is the only reduced object that A reduces to.

In (c), we get that B is syntactically unique—because in the absence of names we need not say "up to α-equivalence." As usual we call B the *normal form* of A, and write $B = NF(A)$. In the present calculus (simple, untyped), there are terms with no normal forms, as we show later.

If $NF(A) = NF(B)$, then A and B are "confluent" or "equivalent" and we write $A \equiv B$; where there is no confusion between syntactic and "semantic" equality, we write simply $A = B$. Two useful facts are:

Proposition 3.2. *(a) The only terms A such that $NF(A)$ is a port are $A = \ominus$ and $A = \oslash$.*

(b) If B is a base object, then $NF(B)$ is also a base object.

Intuitively, these say (a) that an object cannot disappear, and (b) that a base object can never be "de-based." Part (a) is intuitively desirable, and partly motivates our prohibitions of plugging a port into a port or projecting out a port. The result follows directly from these prohibitions. Part (b) is immediate on inspecting the formal definition of reductions given in the Appendix (Section A.2).

The rule that L be reduced when expanding $L \Upsilon^n RHS$ appears to be vital for confluence in our system. Otherwise, a term such as

$$([\ominus, \ominus] \Upsilon^2 \pi_1(\ominus)) \Upsilon A B$$

could be reduced by expanding the right-hand Υ to get $[\ominus, \ominus] \Upsilon^2 \pi_1(A)$ (losing the B because there is only one visible port) and finally $[\pi_1(A), \pi_1(A)]$, instead of the correct reduction to $[\pi_1(\ominus), \pi_1(\ominus)] \Upsilon A B$, and thence to $[\pi_1(A), \pi_1(B)]$. There are further technical issues that might be explored. We argue next that our rules are well-motivated and yield distinctive and interesting program constructs.

3.3 Design Motivations

The Υ operator resembles application in the λ-calculus, but with some notable differences. We do not reduce application to the case of a single right-hand argument, and consider the $A_1, \ldots, A_k$ to be plugged *in parallel*. Additionally, whereas in the λ-calculus the called function determines which arguments are used where via naming, we move this control to the plugging expression by allowing any number of arguments to be plugged into an object, and cycling through them as needed to fill all of the ports in the plugged object. The same object, when plugged with one object will have a very different result than the same object plugged with two different objects. This is a side effect of the fact that UPSILON has no named variables.

The rules on expanding $L \Upsilon RHS$ can be interpreted by analogy with application and function calls, although this is not the interpretation we generally intend. Then $L = \ominus$ would mean there is no procedure body at all, hence nothing to "run." The body is not an argument, but rather needs to be provided from outside the list of arguments. Having $\ominus$ be a member of RHS would be like failing to supply an argument for a (non-optional) formal parameter of the procedure. This is generally illegal even in real-world programming languages that use lazy evaluation. They require each argument to be accessible at activation time *in case* the argument needs to be evaluated. However, this requirement is a common concept in data-flow computation. In the data-flow parlance, UPSILON is a strict (as opposed to lenient) data-flow model.

Continuing the analogy, the requirement that L be reduced is like requiring the body of a procedure to be known and stable before it can be activated. This does not rule out lazy evaluation, because the objects in RHS are not required to be reduced. Indeed our "semi-eager" evaluation usefully distinguishes the case of lazy evaluation used most often in practice from the case of a non-reduced target as permitted in the λ-calculus.

The second part of the definition of pp is the rule that requires the most careful motivation. Within a term $L' \Upsilon RHS'$, the pluggable ports in L' are expecting their own arguments to come from RHS'. Allowing them to be filled from outside would violate the correspondence between ports and fillers that allows us to have a compositional semantics for the inner "Υ." The λ-calculus is more liberal in the sense that names are used to distinguish variables that can be filled from outside from those filled in an immediate application, as with w and z in

$$(\lambda w.(\lambda xy.(wx)(yz))TU)V.$$

Even here the outer reduction to $(\lambda xy.(Vx)(yz))TU$ is restrained if x or y occurs freely in V. In our calculus, treating L' as $(wx)(yz)$ and RHS' as TU, the outer plugging into L' (using w within L' implicitly) is simply forbidden. This arrangement arises partly because this initial version of UPSILON does not have named fields, partly as a matter of principle, and partly for simplicity in the behavior that results.

The first part of the definition of pp, which prevents plugging into ports in sub-objects of base objects, enforces a sense of modularity. In a real language, if you want to touch a sub-object, either you have to access the sub-object and then manipulate it, or you can call a method within the object to manipulate it for you. More simply put, if object `foo` occupies a field of object `bar`, and `foo` has an integer field `val`, then an operation that grabs `bar` cannot access `val` without also grabbing `foo` in some manner—in familiar terms either by `foo.val` or `foo.getValue(...)`. There is no way to get at `val` without in some way going through `foo`. We find that this restriction gives a more faithful modeling even without considering the issue of whether `bar` is privileged to access the field `val` of `foo` at all.

Another important aspect of most object-oriented languages is a way to access the *self* object. In keeping with the object-oriented flavor of UPSILON, we will also have such a way. However, it will come for free out of the recursion theorem as described in Section 5.3. Because of this, `self` will be syntactic sugar for an equivalent object that knows about its self. For a preview, if $A = [\ldots, \texttt{self}, \ldots]$ then the field with `self` refers to A itself. Also, if $A = [\ldots, [\ldots] \Upsilon \texttt{self}, \ldots]$, then the `self` refers to A. However, if $A = [[\texttt{self}], \ldots]$, then the self refers to the sub-object $\pi_1(A)$. More formally, an instance of `self` is a copy of the object that contains that instance of `self` at its top level — and `self` must occur within some base object. Since `self` is not really an object but a syntactic construct, it does not fit into any of these classifications. Before we do anything with a `self` we will really need to figure out what it is, and it will always end up being a base object. Before we do anything with `self`, we need to figure out what it is, more detail in Section 5.3.

4 Program Constructs in UPSILON

We show how UPSILON models some essential programming primitives, for later use and for comparison to their implementation in λ-calculus, other applicative languages, and object-oriented systems.

4.1 Booleans

Whereas Abadi and Cardelli [AC96b] assume booleans as a basic type, we build them from more-primitive components, combining aspects of the λ-calculus and *Smalltalk*.

- $\texttt{True} : [\pi_1([\ominus, \ominus])]$. This is an object with one method, a "then" method.
- $\texttt{False} : [\pi_2([\ominus, \ominus])]$. Likewise an object with one method, "else."

To test whether a boolean object A is True or False, we must first access the method via $\pi_1(A)$, so that the two nested ports become pluggable. The test's behavior for if-then-else is shown by the following term.

- $\texttt{If-Then-Else} : (\ominus \ \Upsilon \ \ominus \ \ominus) \ \Upsilon \ \pi_1(\ominus) \ \ominus \ \ominus$.

Only the three rightmost ports of this term are pluggable, and because the last is $\ominus$, three arguments A, B, C are required, with A expected to be Boolean. Then $\texttt{If-Then-Else} \ \Upsilon \ A \ B \ C$ is reducible to $\pi_1(A) \ \Upsilon \ B \ C$. For this to expand, the target $\pi_1(A)$ must first be reduced (this could also have been done before the last plug), and this is how the method is accessed. If $A = \texttt{True}$ the end result is B, while if $A = \texttt{False}$ the result is C.

We freely write **if** A **then** B else C for $\texttt{If-Then-Else} \ \Upsilon \ A \ B \ C$, and employ natural names for other fixed UPSILON objects and terms, such as the following Boolean operations. We also use standard function notation for outermost arguments, e.g. writing $\texttt{Or}(A, B)$ for $\texttt{Or} \ \Upsilon \ A \ B$.

$$\begin{array}{rcl} \texttt{Or} & : & (\ominus \ \Upsilon \ \texttt{True} \ \ominus) \ \Upsilon \ \pi_1(\ominus) \ \ominus \\ \texttt{And} & : & (\ominus \ \Upsilon \ \ominus \ \texttt{False}) \ \Upsilon \ \pi_1(\ominus) \ \ominus \\ \texttt{Not} & : & (\ominus \ \Upsilon \ \texttt{False} \ \texttt{True}) \ \Upsilon \ \pi_1(\ominus) \ \ominus \end{array}$$

These are all partial completions of If-Then-Else—for instance $\texttt{And}(A, B)$ is **if** A **then** B else False—and are similar to their standard λ-calculus counterparts. Our Not is the only time that we actually use the feature that we can plug more than objects than are needed. The final port in Not is purely for control. If it weren't there, then the plug operator could reduce evaluating to something unexpected. Once the ports are filled, the control port is discarded because there aren't enough ports to fill. Not *can* be done without this trick, as in $((\ominus \ \Upsilon \ \texttt{False} \ \ominus) \ \Upsilon \ \pi_1(\ominus) \ \ominus) \ \Upsilon \ \ominus \ \texttt{True}$, but this is the most elegant version.

Note that in Or, the ports in True are not pluggable because they are "shielded" by two levels of [...], so that when $\texttt{Or}(A, B)$ is expanded, the B goes into the port to the right of True. Here is the expansion of $\texttt{And}(\texttt{True}, \texttt{False})$, with down arrows showing the operator to be reduced.

$((\ominus \ \Upsilon \ \ominus \ \texttt{False}) \ \Upsilon \ \pi_1(\ominus) \ \ominus) \ \overset{\downarrow}{\Upsilon} \ \texttt{True} \ \texttt{False}$ — Now plug the arguments into the method

$\rightsquigarrow (\ominus \ \Upsilon \ \ominus \ \texttt{False}) \ \overset{\downarrow}{\Upsilon} \ \pi_1(\texttt{True}) \ \texttt{False}$

Plug the If-Then-Else method of True and the False object into an evaluation term

$\rightsquigarrow \pi_1(\overset{\downarrow}{\texttt{True}}) \ \Upsilon \ \texttt{False} \ \texttt{False}$ — Expand True

$= (\overset{\downarrow}{\pi_1} \ [\pi_1[\ominus, \ominus]]) \ \Upsilon \ \texttt{False} \ \texttt{False}$ — Extract the If-Then-Else method

$\rightsquigarrow (\pi_1[\ominus, \ominus]) \ \Upsilon \ \texttt{False} \ \texttt{False}$ — Plug the two Falses into the extracted If-Then-Else method

$\rightsquigarrow \pi_1[\texttt{False}, \texttt{False}] \rightsquigarrow \texttt{False}$ — Extract out the False for the final value.

Notice that $\texttt{And}(\texttt{True}, B)$ always follows this pattern and returns exactly B. If B had been True, then $\texttt{And}(\texttt{True}, B)$ would have returned True.

4.2 Simple Integers

An integer can be looked at as an object that knows whether or not it is 0, its predecessor, and how to find its successor. The first two can be constant fields, the third must be a method. The most complicated field will be the successor method: $[\text{False}, \ominus, \pi_3\ominus] \ \Upsilon^2 \ \ominus$. This takes an argument and creates a new object with the argument in the second field. The new integer also inherits its successor method from its predecessor. This yields:

$$
\begin{array}{rcl}
\texttt{Zero} & : & [\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus], \\
\texttt{One} & : & [\texttt{False}, \texttt{Zero}, [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus] \\
 & = & [\texttt{False}, [\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus], \\
\texttt{Two} & : & \texttt{False}, \texttt{One}, [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus] \\
 & = & [\texttt{False}, \\
 & & [\texttt{False}, [\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus], \\
 & & [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus]
\end{array}
.
$$

The effect effect is reminiscent of their familiar set-theory or λ-calculus encodings, but with a definitely object-oriented bent. The term $\texttt{Zero?} = \pi_1(\ominus)$ functions as a test for a number being zero. The successor function can be called with the term $\texttt{Succ} = ((\ominus\ \Upsilon\ \ominus)\ \Upsilon\ \pi_3(\ominus)\ \ominus)\ \Upsilon^2\ n$.

One might feel that to access the predecessor we should be able to do like in **Zero?**, and merely do $\pi_2(\ominus)$. However, since the natural-number predecessor of zero is standardly zero, we must have the more complicated‡

$$
\begin{array}{rcl}
\texttt{Pred} & = & (\texttt{if Zero? then} \ominus \text{ else } \pi_2\ominus)\ \Upsilon^3\ \ominus \\
 & = & (((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\ominus)\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\ominus)\ \ominus\ \pi_2\ominus)\ \Upsilon^3\ \ominus.
\end{array}
$$

We *could* use simply $\pi_2\ominus$ for predecessor if instead of having [ ] as the second field of **Zero**, we could put 0 there. The problem is that defining $\texttt{Zero} = [\texttt{True}, \texttt{Zero}]$ is circular. In Section 5.4 we use the derived availability of **self** to solve this problem.

This is the first example with replication of an argument, and bears some examination. In $\texttt{Pred}(n)$ the outer Υ^3 addresses the first Boolean field, preserves the argument in case it is zero, and extracts its second field in case it is not. The Boolean test thus returns **Zero** if $n = 0$ and the stored predecessor field if not. The Υ^2 in the **Succ** field behaves likewise.

Look at $\text{Succ}(0) \equiv ((\pi_3\ominus)\ \Upsilon\ \ominus)\ \Upsilon^2\ 0 \equiv ((\pi_3\ominus)\ \Upsilon\ \ominus)\ \Upsilon^2\ [\text{True}, [\,], [\text{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus]$. This should be 1. Let us go through this reduction.

$\text{Succ}(0) \equiv ((\ominus\ \Upsilon\ \ominus)\ \Upsilon\ \pi_3(\ominus)\ominus)\ \overset{\downarrow}{\Upsilon}{}^2\ 0$	
$\equiv ((\ominus\ \Upsilon\ \ominus)\ \overset{\downarrow}{\Upsilon}\ \pi_3(0)0)$	Plug the successor method and 0 into an evaluation term
$\equiv (\pi_3(\overset{\downarrow}{0})\ \Upsilon\ 0)$	Expand 0
$\equiv (\overset{\downarrow}{\pi}_3\ ([\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus])\ \Upsilon\ 0)$	Extract the successor method
$\equiv ([\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus)\ \overset{\downarrow}{\Upsilon}\ 0)$	Plug 0 into the extracted successor method
$\equiv ([\texttt{False}, \ominus, \pi_3\ominus]\ \overset{\downarrow}{\Upsilon}{}^2\ 0)$	Plug 0 as the predecessor of 1
$\equiv [\texttt{False}, 0, \pi_3\ \overset{\downarrow}{0}]$	Expand 0
$\equiv [\texttt{False}, 0, \overset{\downarrow}{\pi}_3\ ([\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus])]$	Extract the successor method for 1
$\equiv [\texttt{False}, 0, [\texttt{False}, \ominus, \pi_3\ominus]\ \Upsilon^2\ \ominus]$	

The evaluation of the successor method does require several reductions. However it is still a constant-time operation. In addition, we are breaking down an operation that is often atomic into even smaller, more simplistic, operations.

Notice that the first field of **One** is **False**, $1 \neq 0$. The second field of **One** is actually **Zero** in hand with 0 being the predecessor of 1. The third field is a little more interesting. We see that the object to donate its successor field to 2 will in fact be 1, and 2 will donate its to 3, and so on. We can look at this as each number inheriting its functionality from its predecessor, and all of them inheriting from 0. From this we are able to do arithmetic just like in [AC96b].

‡Note that the shorter $((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1\pi_1(\ominus)\ \ominus\ \pi_2\ominus)\ \Upsilon^3\ \ominus$ is confluent with **Pred**, but not reducible to or from **Pred**.

4.3 Composition and Currying

Composition is a "higher-order" function that, when applied to two (unary) functions f, g, produces a unary function h such that for all arguments x, $h(x) = f(g(x))$. In UPSILON we can write an objective equation for a term C to compute composition: for all objects f, g, and x,

$$(C \Upsilon f\ g) \Upsilon x = f \Upsilon (g \Upsilon x). \tag{1}$$

(Here and below we write "$=$" since semantic equivalence earlier denoted by "$\equiv$", is clearly intended.) In the λ-calculus, where a binary function application $h(x,y)$ can be written hxy (which parses as $(hx)y$) with implicit currying, the equation becomes $(Cfg)x = f(gx)$. This is solved by $C = \lambda fgx.f(gx)$, simply abstracting the right-hand side. In UPSILON, doing similarly would yield

$$C_0 = \ominus \Upsilon (\ominus \Upsilon \ominus).$$

However, while this satisfies $C^{f,g,x} = f^{g^x}$, it does *not* satisfy (1), which entails $(C^{f,g})^x = f^{g^x}$. There are three main reasons why not. First, $C_0^{f,g} = f \Upsilon (g \Upsilon f)$ because all of the ports must be filled by every plug. However, even we made the final port a locked port, so: $C_0^{f,g} = f \Upsilon (g \Upsilon \ominus)$ is reducible—and the result of plugging $(g \Upsilon \ominus)$ into f may be a term T such that T^x does not equal f^{g^x}. For instance, f can be a term which plugs something into its argument, such as $\ominus \Upsilon [\,]$, so that $T = (g \Upsilon \ominus) \Upsilon [\,] = g \Upsilon [\,] = g^{[\,]}$. And $T^x = (g^{[\,]})^x$ not $(g^x)^{[\,]}$ as it should be. Finally, if we try to restrain the reduction by defining

$$C_1 = (\ominus \Upsilon (\ominus \Upsilon \ominus)) \Upsilon \ominus \ominus \oslash,$$

we reach the problem that in $(C_1^{f,g})^x$, the x gets plugged into f, not into g.

Overall, we can abstract the problem as being a "failure of currying": a valence-2 UPSILON term T need not satisfy $T^{hx} = (T^h)^x$ for all (or even any) objects h and x. The following, however, shows how to remedy the defect within UPSILON itself.

Lemma 4.1 (Currying Lemma). *For any object T of valence $k + \ell$, we can construct an object T' of valence k such that for all objects $x_1, \ldots, x_k$ and $y_1, \ldots, y_\ell$,*

$$(T'^{x_1,\ldots,x_k})^{y_1,\ldots,y_\ell} = T^{x_1,\ldots,x_k,y_1,\ldots,y_\ell}.$$

Proof: Define

$$\texttt{Curry} = (((\ominus_T \Upsilon \ominus_k \ominus_\ell) \Upsilon \pi_1(\pi_1(\ominus_T))\pi_1(\pi_1(\ominus))_k \ominus_\ell) \Upsilon$$

$$\ominus_T [[\ominus] \Upsilon \ominus]_k \oslash_\ell) \Upsilon [[\ominus] \Upsilon \ominus]_T \oslash_k$$

Here the subscripts mean k- respectively ℓ-many copies of the sub-object. The subscript T helps us keep track of which ports are expecting T. Let us first look at the various components. $[[\ominus] \Upsilon \ominus]$ encapsulates its argument into a protective box. Once T and the k arguments are in their boxes, plugging the ℓ arguments can't interfere with them until we are read. $\pi_1(\pi_1(\ominus))$ unboxes the encapsulated arguments. Finally, $(\ominus_T \Upsilon \ominus_k \ominus_\ell)$ is what we expected `Curry` to be in the first place. It plugs the k arguments and the ℓ arguments all into T. Let us look at this back to front, as that is the way it will be evaluated. When we plug T, the object to be plugged, into `Curry`, it first gets plugged into the encapsulater, so it can't be accidentally plugged by anything until we are ready. Then it unlocks the next k ports for the first set of arguments. When we plug the first k arguments, the fill the remaining ports at the top level, and the rightmost Υ can reduce. This leaves the encapsulated T as it is, encapsulates all of the k arguments, and then unlocks ports for the remaining ℓ arguments. So, the second line gets all of the arguments and encapsulates them so that they don't interact. At this point, the rightmost Υ on the top line can evaluate. This unwraps T and the first k arguments, and then passes them all into a term that plugs all of the appropriate arguments into T.

Note that we really need the locked ports here as without them, we would have no way of waiting for the k arguments after we have plugged in T. We prove in Section 8.2 that without $\oslash$'s there is *no* way to do this.

■

For example, we obtain the curried version of $K_0 = \pi_1[\ominus, \ominus]$ by $K = \texttt{Curry}\ \Upsilon\ K_0$, which reduces to

$$(((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon\ \ominus\ [[\ominus]\ \Upsilon\ \ominus]\ \oslash)\ \Upsilon\ [[K_0]]\ \ominus.$$

On applying an object X, we get $K\ \Upsilon\ X =$

$$(((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon\ [[K_0]]\ [[X]]\ \ominus \tag{2}$$

Note how in each step a locked port became a port and still retards further evaluation, and how both K_0 and X are encased in double shielding. Now applying this to a second object Y gives:

$$\begin{aligned}((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon\ [[K_0]]\ [[X]]\ Y &=\\ (((\ominus_T\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1([[K_0]]))\ \pi_1(\pi_1([[X]]))\ Y) &=\\ ((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ K_0\ X\ Y &=\\ K_0\ \Upsilon\ X\ Y &=\\ X.\end{aligned}$$

Objects A of any valence k may be curried to take one argument at a time. The object $\texttt{Curry}_k\ \Upsilon\ A$ has valence 1, and all successive applications of one argument at a time have valence 1 as well.

Corollary 4.2. *For any $k \geq 1$ and object T of valence k, we can construct an object $T' = \texttt{Curry}_k(T)$ such that for all objects $x_1, \ldots, x_k$,*

$$T^{x_1,\ldots,x_k} = (\cdots(T'^{x_1})^{x_2})\cdots)^{x_k}.$$

Proof: Lemma 4.1 with $\ell = 1$ gives a term T_1 such that $(T_1^{x_1,\ldots,x_{k-1}})^{x_k} = T^{x_1,\ldots,x_k}$. Applying it again to T_1 gives T_2 such that $(T_2^{x_1,\ldots,x_{k-2}})^{x_{k-1}} = T^{x_1,\ldots,x_{k-1}}$. Iterating in this matter $k-1$ total times gives T'. ■

We can use a very similar construction for composition:

$$\texttt{Compose} = (((\ominus_f\ \Upsilon\ (\ominus_g\ \Upsilon\ \ominus_x))\ \Upsilon\ \pi_1(\pi_1(\ominus_f))\pi_1(\pi_1(\ominus_g))\ominus_x)\ \Upsilon\ [[\ominus]\ \Upsilon\ \ominus_f][[\ominus]\ \Upsilon\ \ominus_g]\oslash_x)$$

Where now the subscripts say which value each port is expecting. This uses the same components as `Curry`, again, encapsulating f and g unlocking the port for x. When x is plugged in, the outermost Υ can evaluate, unwrapping f and g and plugging them into the final term, which is C_0. This gives us $(\texttt{Compose}^{f,g})^x = C_0^{f,g,x} = f^{g^x}$, so `Compose` satisfies (1). In like manner, we claim the construction for all $k, n \geq 1$ of terms $\texttt{Compose}_{k\longleftarrow n}$ that satisfy for all objects f of valence k and $g_1, \ldots, g_k$ of valence n, and all $x_1, \ldots, x_n$:

$$\left(\texttt{Compose}_{k\longleftarrow n}^{f,g_1,\ldots,g_k}\right)^{x_1,\ldots,x_n} = f^{g_1^{x_1,\ldots,x_n},\ldots,g_k^{x_1,\ldots,x_n}}. \tag{3}$$

In the standard world of functions, $f \circ (g_1, \ldots, g_k)$ is said to be the "functional composition" of f and $g_1, \ldots, g_k$.

In Lemma 4.1, it is not possible to make T' an object of valence $k+\ell$ instead of k that is itself curryable, meaning (for $k = \ell = 1$) that $T'^{x,y} = (T'^x)^y$, as well as $(T'^x)^y = T^{x,y}$. This is because of the requirement that all ports be filled after each plug. So, we can't create an object that can take either k or $k+\ell$ arguments. The upshot is that currying is not something that happens "for free" in UPSILON, as it does in the λ-calculus, but is rather an operation that needs to be applied.

5 Turing Completeness

In keeping with goals set forth in the Introduction, we show that UPSILON is adept at simulating both "machines" and "calculi." Thus we have tried to prove in two different ways that UPSILON is a universal model of computation. Unfortunately, the second, simulating the appropriate combinators has proven more difficult than expected, see Sections 5.2 and 10 for more details.

5.1 RAM Simulation

We take machines with some fixed number n of *counters* as representative of random-access machines. Minsky [Min67] showed that two-counter RAMs are universal, but we have no hardship in allowing $n > 2$. A RAM program is a finite sequence of *instructions*, numbered consecutively from 1. We assume that the last line is an **end** statement that appears nowhere else—if two programs P, Q are concatenated, the **end** statement of P can be regarded as "continue" instead. Thus we have four kinds of statement, using **Ni** for the line number and **Rj** for the register number:

Ni INC Rj line i increments register j;

Ni DEC Rj line i decrements register j;

Ni *C* where C is either *continue* or *end*: line i does nothing;

Ni Rj JMP Nk If register j is 0 then line i jumps to line k; otherwise it just progresses to the next statement.

Using this we will create an interpreter object that takes a representation of a RAM program and executes it.

Proof: For the representation:

- The state will be represented as an object with n fields, each a simple integer, as described in Section 4.2, each representing a register.
- We will have the additional syntactic sugar of $\texttt{Id} = \pi_1([\ominus]\ \Upsilon\ \ominus)$ to represent the identity function. As opposed to $\pi_1[\ominus]$ by itself, `Id` can be plugged even when it is a field of a base object. We write $\texttt{Id}(\ominus)$ to emphasize that it has one top level port.
- A line of the program will be an object [`next-line, next-state, end`], where `next-line` is a method that takes an object containing the whole program and the current state, and returns the next line to be executed. Further, `next-state` takes the current state and returns the new state after the line is executed, and `end` is a boolean describing whether the line is the end statement.

  In more detail:

  Ni INC Rj $[\pi_{i+1}(\pi_1\ominus), [\pi_1(\ominus), \pi_2\ominus, \ldots, \texttt{Succ}(\pi_j\ominus), \pi_{j+1}\ominus \ldots \pi_n\ominus]\ \ \Upsilon^n\ \ \ominus, \texttt{False}]$. The next-line method just returns the next line, and the next-state method does nothing except increment the appropriate register.

  Ni DEC Rj $[\pi_{i+1}(\pi_1(\ominus)), [\pi_1\ominus, \pi_2\ominus \ldots, \texttt{Pred}(\pi_j\ominus), \pi_{j+1}\ominus \ldots \pi_n\ominus]\ \ \Upsilon^n\ \ \ominus, \texttt{False}]$ The next-line method just returns the next line, and the next-state method does nothing except decrement the appropriate register.

  Ni continue $[\pi_{i+1}(\pi_1(\ominus)), \texttt{Id}(\ominus), \texttt{False}]$. The next-line method just returns the next line, and the next-state method does nothing.

  Ni end $[\pi_i(\pi_1\ominus), \texttt{Id}(\ominus), \texttt{True}]$. The next-line and next-state methods will never get called, but it might as well loop indefinitely and not change the state if it ever does.

  Ni Rj JMP Nk

  $$[\texttt{if (zero?}(\pi_j(\pi_2\ominus)))\ \texttt{then}\ \pi_k(\pi_1\ominus)\ \text{else}\ \pi_{i+1}(\pi_1(\ominus)), \texttt{Id}(\ominus), \texttt{False}]$$

  If the appropriate register is 0 then the next-line method of this will jump to the appropriate line, otherwise it will just go on to the next line. The next-state method does nothing.

- A program is an object with several lines in it. There must be at least one `end` statement, otherwise the program will never halt.
- The input to the interpreter has the form [interpreter, program, current_line, state]. The interpreter is $(\texttt{if}\ \pi_3(\pi_3\ominus)\ \texttt{then}\ \pi_1(\pi_4\ominus)\ \text{else}\ \pi_1(\texttt{Update}))\ \Upsilon^9\ \ominus$, where `Update` is the term

$$
\begin{array}{l}
[\ominus_a \Upsilon (\\
\qquad [\pi_1(\ominus)_b,\ \pi_2\ominus_c,\ \ominus_d \Upsilon ([\pi_2\ominus_e, \pi_4\ominus_e]\ \Upsilon^2\ \ominus_e),\ \ominus_f\ \Upsilon\ \pi_4\ominus_g] \\
\qquad \Upsilon \\
\qquad \ominus_b\ \ominus_c\ \pi_1(\pi_3\ominus)_d\ \ominus_e\ \pi_2(\pi_3\ominus)_f\ \ominus_g \\
\qquad) \\
] \\
\Upsilon \\
(\pi_1(\ominus))_a\ \ominus_b\ \ominus_c\ \ominus_d\ \ominus_e\ \ominus_f\ \ominus_g
\end{array}
$$

Here $\pi_3\ominus$ is the current line, so that $\pi_3(\pi_3\ominus)$ tells whether the current line is the end. If it is, the `Then` branch is finally taken, and returns the first register of the state—which is $\pi_4\ominus$.

Otherwise, we want to send a new argument back to the interpreter. The first port ($\ominus_a$) will hold the interpreter itself. Its argument comes via the $(\pi_1\ominus)_a$ at the end. Now we need to construct the argument. The first two arguments never change, so we just take the same first and second fields—note that $\ominus_b$ and $\ominus_c$ get passed through two levels of Υ before reaching their final destinations. The third field is the current line. We will need to find the next-line method of the current line and pass in the program and the state. The component $[\pi_2\ominus_e, \pi_4\ominus_e]\ \Upsilon^2\ \ominus_e$ will become an object with these two fields. The body of the method to find the next line is provided by $(\pi_1(\pi_3\ominus))_d$—note how this gets plugged into the first port in $\ominus_d\ \Upsilon\ ([\pi_2\ominus_e, \pi_4\ominus_e]\ \Upsilon^2\ \ominus_e)$.

Likewise, the last field is the current state. We need to find the next-state method of the current line and pass it $\pi_4\ominus_g$. The next-state method is $(\pi_2(\pi_3\ominus))_f$ and this is what gets passed into the first port of $\ominus_f\ \Upsilon\ \pi_4\ominus_g$.

We needed to add some [...] around some terms in order to control which ports are pluggable, so we need to get rid of them with the π_1 at the front.

■

5.2 Simulation of Combinators and Lambda-Calculus

Turing completeness implies that UPSILON can simulate the lambda-calculus, but does not necessarily yield a "natural" simulation. Because UPSILON has no variables, one should expect even more readily to simulate combinators, which are closed lambda-terms. The currying tools from Section 4.3 enable us to do the latter straightaway, with unary Υ formally playing the role of application in an *applicative structure* as defined in [Bar84]. In Section 4.3 we curried $K_0 = \pi_1[\ominus, \ominus]$ to obtain an UPSILON term K that satisfies the defining property $Kxy = x$ of the K-combinator. To generate all combinators by application—in UPSILON by plugging—we need only translate the S combinator $Sxyz = xz(yz)$. We define

$$S_0 = ((\ominus\ \Upsilon\ \pi_2[\ominus, \ominus])\ \Upsilon\ (\ominus\ \Upsilon\ \ominus)\ \ominus\ (\ominus\ \Upsilon\ \ominus))\ \Upsilon\ \pi_1[\ominus, \ominus]\ \ominus\ \ominus\ \ominus\ \ominus \tag{4}$$

and $S = \texttt{Curry}_k(S_0\ \Upsilon\ \ominus\ \ominus\ \ominus)$ from Corollary 4.2. Then for all objects x, y, z,

$$((S\ \Upsilon\ x)\ \Upsilon\ y)\ \Upsilon\ z = S_0\ \Upsilon\ x\ y\ z = (x\ \Upsilon\ z)\ \Upsilon\ (y\ \Upsilon\ z),$$

where "=" again means having the same normal form and $(x\ \Upsilon\ z)\ \Upsilon\ (y\ \Upsilon\ z)$ may (of course) reduce further.

Now in combinatory logic, SKK always equals the identity function. In UPSILON, SKK is written $(S\ \Upsilon\ K)\ \Upsilon\ K$, and this reduces to

$$(((\ominus\ \Upsilon\ \ominus\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon\ [[S_0]]\ [[K]]\ [[K]]\ \ominus$$

in normal form. Now on presenting any argument object z, this reduces to

$$
\begin{array}{lll}
(\ominus\ \Upsilon\ \ominus\ \ominus\ \ominus)\ \Upsilon\ S_0\ K\ K\ z & & \\
= & S_0\ \Upsilon\ K\ K\ z & \\
= & (K\ \Upsilon\ z)\ \Upsilon^(\ K\ \Upsilon\ z) & \text{(now use (2) on the first } (K\ \Upsilon\ z)) \\
= & (((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon\ [[K_0]]\ [[z]]\ \ominus)\ \Upsilon\ (K\ \Upsilon\ z) & \\
= & K_0\ \Upsilon\ z\ (K\ \Upsilon\ z) = z &
\end{array}
$$

as required of a program for the identity. Note that UPSILON is nowhere close to being *extensional*—which would mandate that all terms that compute the identity have the same normal form—since (RAM) program equivalence is undecidable. Note also that it was important for K to be produced by currying as well, as SK_0K_0z in UPSILON reduces to $z \Upsilon z$, not z. This does make SK_0K_0 in UPSILON an expression for the ω-combinator, but a simpler one is

$$\omega = (\ominus \Upsilon \ominus) \Upsilon \ominus.$$

Note that the outer Υ makes ω formally have valence 1; the object $\omega_0 = \ominus \Upsilon \ominus$ has valence 2, even though $\omega_0 \Upsilon z = \omega \Upsilon z = z \Upsilon z$ for all objects z. And, of course, the reduction of $\omega \Upsilon \omega$ never terminates.

The UPSILON representations of combinator expressions are proportional in size to the originals, and the time to reduce them is likewise linear. However, the size expansion from lambda terms of size n to combinators must be $\Omega(n \log n)$ [Sta86]. Moreover Barendregt ([Bar94], p258), citing an earlier version of [Sta86], remarks that Statman's matching $O(n \log n)$ construction has constants so high that $O(n^2)$ translations such as Turner's [Tur79] and improvements of it are more attractive in practice. A question for current attention is whether more-efficient simulations of the λ-calculus are possible into UPSILON. Our current strategies for translating de Bruijn's "nameless dummies" method of representing and operating on λ-terms [deB72]) into UPSILON also have quadratic blowup in worst case—they involve repeating the tree structure of applications for each lambda-abstraction in the term. We speculate whether UPSILON provides a truly richer or more succinct set of combinators, as hinted by the simple form of ω above, rich and succinct enough to achieve a linear-size, linear-time simulation.

We have shown that UPSILON is equally adept at simulating an imperative model (RAM programs) and a functional model (combinators). The simulations are natural and intuitive in both cases. This enhances the prospects for UPSILON serving as an informative simple model for languages that compute naturally.

5.3 Self

Because UPSILON is Turing compete and it has a composition function, we see that it is an Acceptable Programming System (APS), as described in [Smi94]. A fundamental consequence of this is that via the recursion theorem of [Kle38], there is an effective procedure that will turn any object O into an O' that is otherwise equivalent to O, except that it contains an extra field with a description of O'. In other words, a notion of "self" emerges from the mathematical properties of our model. The recursion theorem is also proved directly in Appendix B.

The recursion theorem is flexible enough that through its use we can have any of a number of functionalities for self. We can use it to get access to the object that has self at the top level, or any object which has self in a sub-object. More interestingly, we can use the recursion theorem to allow objects which haven't even been created yet to have access to themselves. Since the recursion theorem is constructive, we can incorporate it into any program that we wish to use to interpret an UPSILON program.

The problem remains to determine when the recursion theorem should be actually applied to expand "self" into an actual object. In π_i`self`, the self must expand, otherwise the π_i can't evaluate. Likewise, in `self` Υ ... the self must expand. Otherwise, if an instance of self passes through a level of [ ] then it must be evaluated otherwise afterwards it may refer to a completely unrelated object. E.g. in [[...] Υ `self`] the self obviously should refer to the outer object. Likewise in [π_1[`self` ...]] the self should refer to the inner object. Other than these four cases, "self" can remain exactly that, a string that only gets expanded when it is needed. This allows us to have [..., `self`]@[..., `self`] where, after the @ is evaluated, the two selfs will refer to the same object. This should allow us to model inheritance in a relatively seamless way when we add types to the model.

5.4 Better Integers

We could use integers like we did it in Section 4.2, However, now that we have access to self, there is an easier way to do `Pred`. The only difference is that now, `Zero`'s second field can actually be `self`. So `Zero` = [`True`, `self`, [`False`, $\ominus$, $\pi_3\ominus$] Υ^2 $\ominus$], and our new syntactic sugar for predecessor, which looks like $\pi_2(\ominus)$, is much simpler than the previous version and keeps the predecessor of 0 to be 0 because of the self.

6 Showing that UPSILON is object-oriented

Before we can show that UPSILON truly *is* object-oriented, we must first explain what we mean by "object-oriented". We break this up into two parts, syntax, and style.

An object-oriented syntax must, obviously, be built on top of objects. Objects are arbitrarily nested structures (records or tuples) in which some of the fields may represent methods, or computations that can be invoked later. Additionally, the methods must have some access to the object in which they reside. This is the normal use of `self`. UPSILON does meet all of these requirements. A method is a field which is a term.

The object-oriented style is mainly that of objects communicating with each other through method invocations. Our booleans and better integers exemplify this very well. An integer is an object with three methods. It knows whether or not it is `Zero`, its predecessor, and how to find its successor. No external functions are needed to invoke these operations, you merely need to access the methods that are already there. With a little more effort, we could add any other arithmetic methods that we wished. Likewise, booleans are objects with an if-then-else method, very much like in Smalltalk [GR89].

The other major (many would same the primary) components of an object-oriented system are inheritance and subsumption. We can model inheritance and extension crudely through the @ operator, however, we do not yet have a concept of subtype or subsumption. Adding subtypes is a future addition to the types as described in the next section.

7 Types

The difficulty of devising a type theory for UPSILON is in determining how to type ports, and incomplete objects. First, in the typed λ-calculus, each parameter has a type associated with it in the λ-binding. We have no such binding operator, so we need another way to describe what type a port is expecting. We will do this with a superscript so, $\ominus^\tau$ is a port expecting to be filled with a value of type τ, and $\oslash^\tau$ is a locked port expecting to be filled TWICE. The first time, it can be filled with anything, the second time, it must be filled with a value of type τ. For incomplete objects, we found that a generalized arrow-notation works well, with the left hand side of the arrow being a list of the types of the pluggable ports in the object, and the right hand side being the type of object that results when those ports are all filled.

With that in mind, the syntax for types is as follows:

$$\tau :== \overline{\tau} | \overline{\overline{\tau}} | \langle \tau, \ldots, \tau \rangle | \{\tau, \ldots, \tau\} \rightarrow \tau$$

- $\overline{\tau}$ is the type of a port which is expecting a value of type τ. ie. $\ominus^\tau$.
- $\overline{\overline{\tau}}$ represents the type of a locked port of type τ. ie $\oslash^\tau$.
- $\langle \tau, \ldots, \tau \rangle$ represents a base object (or something that reduces to a base object) with fields of appropriate types.
- $\{\tau, \ldots, \tau\} \rightarrow \tau$ represents a term that has m ports, of types on the left, and when they are filled, we get the τ on the right.
- We will also allow the meta-syntactic type $\top$ to allow a locked port to be unlocked by any object, regardless of type. This is an unnecessary construct, but it proves useful.

7.1 Type Rules

First it will be useful to define some helper functions. Map is the standard map function, applying a function to every value in a list and returning a list of the results. Actually, we will abuse notation slightly and say for example: map(LHS$'$, τ_i) to mean that we want to apply the function LHS to each of the τ_i's and return a list of the results. LHS is the left hand side of the type, RHS is the right hand side, and LHS$'$ is the left hand side, minus ports that aren't visible because the term is already inside a base object. RHS$'$ is likewise the RHS of a term that is already inside a base object. More formally:

- if $\tau = \langle \tau_1, \ldots, \tau_m \rangle$:

  - $\mathrm{RHS}(\tau) = \langle \mathrm{RHS}'(\tau_1), \ldots, \mathrm{RHS}'(\tau_1) \rangle$ The result of plugging a base object is a base object with all of its fields plugged, with the knowledge that those fields are within a base object.
  - $\mathrm{RHS}'(\tau) = \tau$
    The result of plugging a base object that is already within a base object is itself
  - $\mathrm{LHS}(\tau) = \mathrm{map}(\mathrm{LHS}', \tau_i)$
    The visible ports in a base object are the visible ports in the fields, keeping in mind that they are within a base object.
  - $\mathrm{LHS}'(\tau) = \{\}$
    A base object within another base object has no visible ports.

- if $\tau = \{\sigma_1, \ldots, \sigma_m\} \rightarrow \sigma$

  - $\mathrm{RHS} = \mathrm{RHS}'(\tau) = \sigma$
    The result of plugging a term is the... result of plugging the term. [NOTE: Need a better way of saying this]
  - $\mathrm{LHS} = \mathrm{LHS}'(\tau) = \{\sigma_1, \ldots, \sigma_m\}$
    The visible ports in a term are the ... visible ports in the term. [NOTE: need a better way of saying this too]

- if $\tau = \overline{\sigma}$

  - $\mathrm{RHS} = \mathrm{RHS}'(\tau) = \sigma$
    The result of plugging a port is the type that is expected by the port.
  - $\mathrm{LHS} = \mathrm{LHS}'(\tau) = \{\sigma\}$
    The visible port in a port is the port itself. There is no risk of accidentally evaluating $\ominus \Upsilon A$ because that is a separate type rule.

- if $\tau = \overline{\overline{\sigma}}$

  - $\mathrm{RHS} = \mathrm{RHS}'(\tau) = \overline{\sigma}$
    The result of plugging a locked port is a port.
  - $\mathrm{LHS} = \mathrm{LHS}'(\tau) = \{*\}$
    The visible port in a locked port is the port itself, but it can be unlocked by any object, regardless of type.

1. TyPort

   $$\frac{}{\ominus^{\tau} : \overline{\tau}}$$

   This shows the type of a port expecting a value of type τ.

2. TyLPort

   $$\frac{}{\oslash^{\tau} : \overline{\overline{\tau}}}$$

   This shows the type of a locked port expecting a value of type τ.

3. TyBase

$$\frac{A_i : \tau_i}{[A_1, \ldots, A_n] : \langle \tau_1, \ldots, \tau_n \rangle}$$

This shows the type of a base object, with fields of types τ_i.

4. TyPi

$$\frac{\begin{array}{c} A : \langle \tau_1, \ldots, \tau_n \rangle \\ \tau_i \neq \overline{\sigma} \end{array}}{\pi_i(A) : \tau_i}$$

The type of a projection term that can actually evaluate is the type of the field that will be projected out.

5. TyPiPort

$$\frac{A : \overline{\tau} A' : \tau \Rightarrow \pi_i(A') : \sigma}{\pi_i(A) : \{\tau\} \rightarrow \sigma}$$

If the projection is paused because the object to be projected from is a port, then it must be plugged so that it can be reduced. It may be that τ is not a base object type either, however, once A is plugged, a value of type τ will result, and the result of the whole term will be whatever a projection from a value of type τ results in. Note that this does take into account the possibility that $A : \overline{\overline{\tau}}$.

6. TyPiPause

$$\frac{\begin{array}{c} A : \langle \tau_1, \ldots, \tau_n \rangle \\ \tau_i = \overline{\sigma} \\ \sigma = \overline{\overline{\sigma}'} \end{array}}{\pi_i(A) : \mathrm{map}(\mathrm{LHS}', \tau_j) \rightarrow \sigma}$$

If the projection is paused because the field to be projected is a port, then it can't be evaluated until it is plugged. So, all of the ports of the fields must be plugged and then the σ field is projected out.

7. TyPiLPause

$$\frac{\begin{array}{c} A : \langle \tau_1, \ldots, \tau_n \rangle \\ \tau_i = \overline{\overline{\sigma}} \end{array}}{\pi_i(A) : \mathrm{map}(\mathrm{LHS}', \tau_j) \rightarrow (\mathrm{map}(\mathrm{LHS}', (\mathrm{RHS}'(\tau_k))) \rightarrow \sigma)}$$

If the projection is paused because the field to be projected is a locked port, then it can't be evaluated until it is plugged twice. So, first all of the ports of the fields must be plugged creating an object of type $\langle \mathrm{RHS}'(\tau_1), \ldots, \mathrm{RHS}'(\tau_n) \rangle$. This must then be plugged in full to allow the σ term to be projected out.

8. TyPiTPause

$$\frac{\begin{array}{c} A : \{\tau_1, \ldots, \tau_m\} \rightarrow \tau \\ A' : \tau \Rightarrow \pi_i(A') : \sigma \end{array}}{\pi_i(A) : \{\tau_1, \ldots, \tau_m\} \rightarrow \sigma}$$

If the projection is paused because the object to be projected from is not yet a base object (and can't be reduced to one), then it must be plugged so that it can be reduced. It may be that τ is not a base object type either, however, once A is plugged, a value of type τ will result, and the result of the whole term will be whatever a projection from a value of type τ results in.

9. TyAppend

$$\frac{\begin{array}{c} A : \langle \tau_1, \ldots, \tau_n \rangle \\ B : \langle \sigma_1, \ldots, \sigma_m \rangle \end{array}}{A@B : \langle \tau_1, \ldots, \tau_n, \sigma_1, \ldots, \sigma_m \rangle}$$

If the term is an append term, and it can evaluate, then the type of the term is the two base types concatenated.

10. TyAppendPause

$$\frac{\begin{array}{c} A : \tau \\ B : \sigma \\ \text{One of } \tau \text{ and } \sigma \text{ is not a base type} \\ (A' : \mathrm{RHS}(\tau) \wedge B' : \mathrm{RHS}(\sigma)) \Rightarrow A'@B' : \nu \end{array}}{A@B : \ \mathrm{LHS}(\tau) \ @ \ \mathrm{LHS}(\sigma) \to \nu}$$

If the term is an append term, and it can't evaluate, one of the two sub-terms must be plugged.

11. TyPlug

$$\frac{\begin{array}{c} A : \{\tau_1, \ldots, \tau_m\} \to \tau \\ A_i : \tau_i \\ \text{None of the } \tau\text{'s are over-lined.} \end{array}}{A \ \Upsilon \ A_1, \ldots, A_m : \tau}$$

If the term is a plug term, and it can evaluate, then the type of the term is the RHS of the type of A.

12. TyPlugTarg

$$\frac{\begin{array}{c} A : \overline{\{\tau_1, \ldots, \tau_m\} \to \tau} \\ \text{NOT: } A : \overline{\overline{\{\tau_1, \ldots, \tau_m\} \to \tau}} \\ A_i : \sigma_i \\ \text{None of the } \sigma_i \text{ are double over-lined} \\ \mathrm{RHS}(\sigma_i) = \tau_i \end{array}}{A \ \Upsilon \ A_1, \ldots, A_m : \{\{\tau_1, \ldots, \tau_m\} \to \tau\}@\mathrm{map}(\mathrm{LHS}, \sigma_i) \to \tau}$$

If the plug can't evaluate because A is a port, and there are no locked ports, then the term must be plugged. The visible ports are A and those in the A_is. Once those are all plugged, the term can evaluate, and give a τ. Note that we do NOT require that the $A_i : tau_i$ because there is a plug to take place that may make the arguments into the correct types even if they aren't the correct types yet.

13. TyPlugLTarg

$$\frac{\begin{array}{c} A : \overline{\overline{\{\tau_1, \ldots, \tau_m\} \to \tau}} \\ A_i : \sigma_i \\ \mathrm{RHS}(\sigma_i) = \tau_i \end{array}}{\begin{array}{c} A \ \Upsilon \ A_1, \ldots, A_m : \{\{\tau_1, \ldots, \tau_m\} \to \tau\}@\mathrm{map}(\mathrm{LHS}, \sigma_i) \to (\{\{\tau_1, \ldots, \tau_m\} \to \tau\}@ \\ \mathrm{map}(\mathrm{LHS}, (\mathrm{RHS}(\tau_i))) \to \sigma) \end{array}}$$

If the plug can't evaluate because A is a locked port, then everything must be plugged twice. So, first all of the ports of A and the arguments must be plugged creating an object where $A = \ominus$, and everything else is the RHS of what was there previously. This must then be plugged to allow the plug to actually evaluate.

14. TyPlugTargLSource

$$\frac{\begin{array}{c} A : \overline{\{\tau_1, \ldots, \tau_m\} \rightarrow \tau} \\ \text{NOT: } A : \overline{\overline{\{\tau_1, \ldots, \tau_m\} \rightarrow \tau}} \\ A_i : \sigma_i \\ \text{Some of the } \sigma_i \text{ are double over-lined} \\ \text{RHS}(\sigma_i) = \tau_i \end{array}}{A \Upsilon A_1, \ldots, A_m : \{\{\tau_1, \ldots, \tau_m\} \rightarrow \tau\}@\text{map}(\text{LHS}, \sigma_i) \rightarrow (\text{map}(\text{LHS}, (\text{RHS}(\tau_i))) \rightarrow \sigma)}$$

If the A is a port, but one of the arguments is a locked port, then the term must still be plugged twice in order to evaluated. However, only the first plug can plug into A.

15. TyPlugSource

$$\frac{\begin{array}{c} A : \{\tau_1, \ldots, \tau_m\} \rightarrow \tau \\ A_i : \sigma_i \\ \text{RHS}(\sigma_i) = \tau_i \\ \text{At least one of the } \sigma_i\text{'s are over-lined, but not doubly so.} \end{array}}{A \Upsilon A_1, \ldots, A_m := \text{map}(\text{LHS}, \sigma_i) \rightarrow \tau}$$

If A is not a port, but some of the arguments are, but none of them are locked, then the term must be plugged once in order to evaluate.

16. TyPlugLSource

$$\frac{\begin{array}{c} A : \{\tau_1, \ldots, \tau_m\} \rightarrow \tau \\ A_i : \sigma_i \\ \text{RHS}(\sigma_i) = \tau_i \\ \text{At least one of the } \sigma_i\text{'s are doubly over-lined.} \end{array}}{A \Upsilon A_1, \ldots, A_m := \text{map}(\text{LHS}, \sigma_i) \rightarrow (\text{map}(\text{LHS}, (\text{RHS}(\tau_i))) \rightarrow \sigma)}$$

If A is not a port, but some of the arguments are locked ports, then the term must be plugged twice in order to evaluate.

7.2 Properties of the Type Theory

Theorem 7.1 (Unique Types). *In the typed version of* UPSILON, *if $O : \tau$ and $O : \tau'$ are both derivable, then $\tau \equiv \tau'$.*

Proof: This follows by a trivial induction over the derivation of $O : \tau$. ∎

Theorem 7.2 (Subject Reduction). *Let O be any object, and assume $O \overset{\rightarrow}{\Upsilon} O'$ (as defined in Appendix A.2). If $O : \tau$ then $O' : \tau$.*

Proof: This follows by a trivial induction over the derivation of $O \overset{\rightarrow}{\Upsilon} O'$. ∎

7.3 Differences Between the Typed and Untyped Models

As would be expected from the Strong Normalization result, the typed UPSILON is a much weaker model than the untyped UPSILON. It is not possible to have infinite calculations. Additionally, it is impossible to represent recursive types within the current type theory because they would be infinitely long. Neither the integers, nor even the simple integers described above are typable because of the predecessor and successor fields, both of which return integers. Also, the RAM simulator is untypable because it takes itself as one of its arguments which would also create a recursive type. The solution to this problem is of course to add recursive μ types. This is not surprising as every object-oriented system has some means of achieving recursive types, whether it be Abadi and Cardelli's ς operator or the more standard μ.

Booleans are a little more interesting. Our implementation of booleans is typable, however, it is in fact a family of types. For every type, τ, there would have to be a different type, Bool(τ), where the arguments and results of the If-Then-Else method are of type τ. The best solution to this problem is polymorphism.

These two solutions have not yet been investigated.

8 Alternative versions of the model

We have already seen why two of the distinctive rules of UPSILON are required, namely that you can't plug into a term until it is reduced, and that an Υ with a naked port on either side of it can't reduce. However, there are other restrictions that need to be motivated as well. We will motivate these by describing alternate models in which various parts of the model are changed, and show where things go wrong.

8.1 Partial Fill

One of the more surprising restrictions is that when an Υ term reduces, it must fill every pluggable port in the LHS. This restriction greatly simplifies the type theory in Section 7.1.

Assume that we have a modified version of UPSILON, call it UPSILON$'$ where we relax the restriction that every visible port be filled by a plug, and we require the superscripts on the Υ operator to force cycling of arguments. If not all ports are filled, the ports are filled from left to right. Also note that the object may be able to reduce before all top level ports are filled, as in $\pi_1([\ominus, \ominus])$. If we fill in only the first port, it can reduce. First realize that in our proofs of confluence, in Appendix A and in our proof of Turing completeness in Section 5 we always fill every top level port. So, those proofs go through with no changes in UPSILON$'$. Consider the type rules in this model.

The type of a term will still need to have the types of all of the visible ports, but it will also need to have the type of object that is returned if only the first m ports are filled for every m. So, each arrow type will look like: $\{\tau_1, \ldots, \tau_n\} \to \{\sigma_1, \ldots, \sigma_n\}$. I.e. a type will look like a tree with each node potentially having large fan-in and large fan-out. In this model, the lengths of the types become unmanageable.

Theorem 8.1. *In* UPSILON$'$ *the length of a type can be exponential in the lengths of the types of its sub-terms.*

Proof: We are going to compute a good-case lower bound on length of a type. To this end, we will assume that if you fill up all of the top level ports in a sub-term, it will not reduce to something else with ports. This assumption only makes the length of the type shorter, and it makes it much easier to compute the lower bound on the length. It is also highly unlikely in actual practice.

Consider the term:

$$A = \pi_n([A_1, A_2, \ldots, A_{n-1}, \ominus^\tau])$$

with:

$$A_i : a_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,m_i}\} \to \{b_{i,1}, b_{i,2}, \ldots, b_{i,m_i}\}$$

Where the $b_{i,j}$ look the same as the top level type, except they have an additional subscript, etc.

The left-hand side of the type of A will be the size of the number of ports visible in A. Consider the right-hand side.

Renumber the fields right to left, so the $\ominus^\tau$ is field 0, etc, so we actually have:

$$A = \pi_n([A_{n-1}, A_{n-2}, \ldots, A_1, \ominus^\tau])$$

So, what will the rightmost fields of the type look like?

$$A : \{\ldots, \tau\} \to \{\ldots, \tau\}$$

Because if you fill all of the ports, you can evaluate the π_n, and the τ comes out. So, the final field contributes a single τ to the final tree.

Now let us consider the next field to the left. The type of A_1 will contribute m_1 ports to the overall type. However, if we fill all of the ports in A except the final one, then we don't have just whatever A_1 returns when it is all filled, we also have the final τ. So, every right-hand side in the entire sub-tree under a_1 will have an additional branch, in case at THAT point, the τ is filled. This doubles the size of the tree. So, now the type has size: $2|a_1| + 1$. where a_1 is the size of the type that A_1 contributed.

Now, let us look at a_2. Now, instead of needing to add a τ to every node in the a_2 tree, we need to add the entire a_1 tree, augmented with the final τ. So, the a_1 contribution ends up being $|a_2| * (2|a_1| + 1)$. And the final size end up being: $|a_2| * (2|a_1| + 1) + (2|a_1| + 1)$. We can now clearly see the pattern. The size of the j^{th} type will be:

$$f_j = |a_{n-1}| * (f_{n-1} + 1) + \sum_{i=1}^{j-1} f_i$$

This is clearly exponential.

■

Notice that this is exponential NOT in the size of A but in the sizes of the a_i. I.e. it's exponential in the sizes of the types of the sub-terms. In the lambda-calculus, it is linear. I.e. the only type rule that makes larger types is $(\Gamma, x : \tau \vdash M : \sigma) \Rightarrow (\lambda x.M : \tau \to \sigma)$. And the size of $\tau \to \sigma$ is just $|\tau| + |\sigma| + 1$. When we look at this in terms of the size of types with respect to terms, in the lambda-calculus, this becomes polynomial, and in ours, it becomes super-exponential. Also, remember that this is sort of a "good-case" lower bound. We are NOT dealing with any ports that a term leaves over when all of its top level ports are filled and it is reduced. This will add MORE ports to every tree, making things bigger by the size of a_{1,m_1} for every port thereafter, etc.

8.2 No locked ports

The other unique concept of UPSILON is the idea of a locked port. The only sections where they are used is in Currying in Section 4.3, and in $s_{1,1}$ in the proof of the recursion theorem in Section B. Since it is NOT used in the proof of Turing completeness in Section 5, it might seem that these could be implemented without this tool. However, this is not possible. Let us consider UPSILON$''$, a version of UPSILON where plugging must fill every top level port, but there are no locked ports.

Theorem 8.2. UPSILON$''$ *is Turing complete, but not an APS*

First we will need to prove a lemma.

Lemma 8.3. *In* UPSILON$''$, *in the term $T' = T^A$, where T' is reduced, so it can be plugged, no term A', a sub-term of A, can occur anywhere within an Υ term.*

Proof: Let T, T', A and A' be defined as above. Assume that A' is a sub-term of T' and it is within an Υ term. The only way that it could have gotten there is to have been plugged there by some outer Υ term that has reduced. Therefore, at that time, all top level ports were filled. So there were no naked ports, and nothing to keep the Υ term that A' is in from reducing, which means that T' is not normal.

■

Now, we are ready to prove the theorem.

Proof: First, since the proof of Turing completeness in Section 5 doesn't use locked ports at all and still fills every top-level port, it is clear that UPSILON$''$ is still Turing complete. It remains to prove that it is not an APS. We use the notion of reduction ($\rightsquigarrow$) as defined in Appendix A.2

Because of Lemma 8.3, there is no term S in which

$$S\ \Upsilon\ i\ x \rightsquigarrow ((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon^{[}\ [i]]\ [[x]]\ \ominus$$

which is what we really need for $s_{1,1}$.

The only other possible way to get $s_{1,1}$ is to have $S \Upsilon\ i\ x = T$ not actually stall. So, it will have to wrap something around x to keep it from interacting with things, call this $P()$, and have some sort of place holder for y, call it Q. So, $T = S \Upsilon\ ix = i \Upsilon\ P(x)Q$ With three guarantees:

1. For all $B, P(x) \Upsilon\ B \rightsquigarrow x$
2. For all $B, Q \Upsilon\ B \rightsquigarrow B$
3. P and Q are guaranteed to not cause any damage inside i or x.

So, then what would happen in T is $P(x)$ and Q would plug into T but since they are inert, there would be no unexpected interactions until B came along and exposed everything, then it all evaluates as normal.

However, this is not possible. Let $i = \ominus \Upsilon\ \pi_1(\ominus)$ Then we end up with: $P(x) \Upsilon\ \pi_1(Q)$ Now, consider guarantee 1, this means that we end up with x. When we plug y into this we get $x \Upsilon\ y$ not $x \Upsilon\ \pi_1(y)$ which is what we expected.

Now its possible that $P()$ and Q can be more intelligent than stated above, that instead of guarantee 1, we say that P and Q are guaranteed to act in some intelligent way to make it work out. For example, in this example, the correct thing for it to do might be to replace all of the ports in x with $\pi_1(\ominus)$ thus when B gets plugged into this now x', the projection takes place, just one step later. However, how can $P()$ know this? What if we changed the π_1 in i to π_2? $P()$ can't have this information, so it fails.

So, from Lemma 8.3, it is impossible to stall the initial plug that plugs x into i, ie, we can't protect x outside of i, and from the above it is impossible to safeguard x within i, ie, we can't protect x inside of i. If we don't protect x, then several different things can happen. 1) i can have some unexpected interaction with x, such as plugging x into itself when x is expecting to have y plugged into it. 2) When y *is* plugged in, it will end up directly in x instead of in the ports within i where it is expected. Therefore there is no way to effectively compute $s_{1,1}$ within UPSILON$''$.

■

Because of the results of this section, both the complete fill restriction and the existence of locked ports are technically well motivated.

9 Contexts

Most of the research into contexts has been done within the λ-calculus, where the interplay between holes and variable binding can be studied. Usually, contexts are reduced to a different form of variable binding operation, and thus raised to the level of the well-understood λ-calculus. We have taken the opposite approach. We have eliminated everything from the model *except* the contexts. We have eliminated variable binding completely, and left context filling as the only method of information transfer. We think that in looking at contexts in this completely new way we may be able to see things about contexts that we couldn't see before.

One of the main areas of interest in context research is that of variable capture. One might think that since UPSILON has no variables, we can't even describe the variable capture problem. However, by combining our foci of contexts and object-oriented programming we can do just that.

Consider a pre-method (a method body before it has been inserted into an object). Since a method may have access to the `self` object, so may a pre-method. However, what does `self` bind to? We can consider `self` as an unbound variable in this context. We can see this more clearly if we extend the meta-syntax to `yourself`, which behaves just like `self`, except that when it is plugged into a base object, it turns into `self` rather than expanding, so that

$$[\ominus] \Upsilon\ \texttt{yourself} \rightsquigarrow [\texttt{self}].$$

This lets us describe pre-methods as terms which contain `yourself` at top level. Now, the exact object that `yourself` refers to can change depending on precisely which object the pre-method is plugged into, and thus we can have variable capture issues, as in normal context discussions.

Additionally, since contexts are the main foundation of our model, we can represent more complex contexts than are normally studied. In most studies of contexts, each term only has one hole, which sometimes is allowed to occur in multiple places within the term. In contrast, we allow not only arbitrary number and copies of holes (the same data plugging multiple holes), but our contexts are much more complex. Each context in UPSILON is actually a structured hierarchy of contexts with some holes being visible and others not. None of these things are studied in the literature we've seen, presumably because they are too complex to be represented in the standard syntax, however within our model, they are automatically present.

10 Future Work

The preliminary results outlined above show the viability of UPSILON as a model of computation. Since the model has an object-oriented flavor, we may be able to build onto it features and derive from it explanations that address distinctive components of OOP and programming languages: inheritance, types, dynamic dispatch, side-effects, verification, efficiency of object encoding, analysis of execution, and more. The first step in this is to add recursion, inheritance, and subsumption to our type theory. The former is so that we can re-acquire internal references to self for the typed version.

We also intend to develop complexity measures for UPSILON with which to analyze the cost of sub-object extraction and inheritance operations. We also intend to emulate the successful approach of Jones [Jon99], who was able to capture some well known complexity classes by restricting the application of certain control structures. The hope here is to characterize in complexity-theoretic terms the effect of some practically-oriented restrictions on object-oriented programming. For example, how is program complexity related to the structure and orientation of the Υ operators in one of our objects? The same question can also be asked about $\ominus$.

Our confluence theorem was mathematically required to justify our new model. From the standpoint of confluence, what we needed to show was that when there was a choice of reduction to performed at any step, it did not matter which choice was made. Furthermore, it was easy to identify cases where reduction order could matter, and those were all cases in which the *target* of a plugging operation was not reduced. For cases when the order does not matter, the operations may also be performed in parallel. Indeed, we suspect that UPSILON is usefully closer to modeling parallel complexity theory than other calculi. Immerman [Imm87, Imm89] and others including [CH82] have demonstrated that low-level parallel complexity classes correspond to first-order formal systems, and that possession of a total ordering matters in the analysis. To implement complexity analysis, we need to introduce some input and output conventions to UPSILON, and not much else.

Additionally, the same properties that allowed us to induce `self` also lead to several more powerful recursion theorems [Smi94]. For example, there are *infinitary* recursion theorems that allow the construction of infinite sequences of programs, each of which knows its own index in the sequence and how to generate the sequence. These effectively yield a collection of programs, each of which has "pointers" to a finite number of other programs in the collection. Hence, we have the capability to construct an object with explicit pointers to all of the objects from which it inherits methods, etc. Unraveling the chain of mathematics leading to these results in the UPSILON has the potential to suggest efficient solution strategies to the well known "yo-yo" problem. Since we already have an organic proof of the simple recursion theorem within the model, the proofs of the other, syntactically more potent, recursion theorems will follow according to their original proofs. We only need to work out the details of the strategies.

Also, at this point, it should be clear that UPSILON is primarily a data-flow language. Our requirement that none of the arguments to a Υ operator are naked ports when it reduces is exactly the restriction that an operation can't fire in data-flow languages until all of its incoming links have data on them. We intend to investigate this relationship further.

Equally clearly, UPSILON is a context calculus. We intend to use this to investigate the relationships between contexts and data-flow as a basis for computation.

The type theory also calls for further investigation. We must prove a strong normalization result. Given this, it would also be nice to determine explicitly which objects are representable within our simple type theory, along lines of [Sch76].

References

[AC96a] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[AC96b] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AD79] W.B. Ackerman and J.B. Dennis. Val – a value-oriented algorithmic language: Preliminary reference manual. Technical Report TR-218, MIT Laboratory for Comptuer Science, June 1979.

[Ada68] D.A. Adams. A computation model with dataflow sequencing. Technical Report CS 117, Computer Science Department, Stanford University, 1968.

[AGP78] Arvind, K.P. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report 114a, University of California, Irvine, December 1978.

[Bar84] Henk Barendregt. *The lambda calculus, its syntax and semantics.* North Holland Publishing Co., Amsterdam, 1984.

[Bar94] H. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 320–363. Elsevier and MIT Press, 1994.

[BBDCL97] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping constraints for incomplete objects (extended abstract). In *TAPSOFT*, pages 465–477, 1997.

[BBL96] Viviana Bono, Michele Bugliesi, and Luigi Liquori. A lambda calculus of incomplete objects. In *MFCS*, pages 218–229, 1996.

[Bru94] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[CH82] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[deB72] N.G. deBruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[FHM94] Kathleen Fisher, Funio Honsell, and John Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1:3–37, 1994.

[Fri58] R. Friedberg. Three theorems on recursive enumeration. *Journal of Symbolic Logic*, 23:309–316, 1958.

[GR89] Adele Goldberg and David Robson. *Smalltalk 80: The Language.* Addison-Wesley, Reading, MA, 1989.

[HO98] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Studies, Kyoto Univ, 1998.

[Imm87] N. Immerman. Languages which capture complexity classes. *SIAM Journal on Computing*, 16:760–778, 1987.

[Imm89] N. Immerman. Expressibility and parallel complexity. *SIAM Journal on Computing*, 18:625–638, 1989.

[Jon99] N. Jones. Logspace and ptime characterized by programming languages. *Theoretical Computer Science*, 1999.

[Kle38] S. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[LC96] Luigi Liquori and Giuseppe Castagna. A typed lambda calculus of objects. In *ASIAN 1996*, pages 129–141, 1996.

[LF96] Shinn-Der Lee and Daniel Friedman. Enriching the lambda calculus with contexts: Toward a theory of incremental program construction. In *International Confecence on Functional Programming Languages 1996*, pages 239–250, May 1996.

[Min67] Marvin Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (parts i and ii). *Information and Computation*, 100(1):1–77, September 1992.

[PT94] Benjamin Pierce and David Turner. Simple type-theoretic foundations for object-oriented programmin g. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Rod69] J.E. Rodriguez. A graph model for parallel computation. Technical Report TR-64, MIT, September 1969.

[Rog58] H. Rogers Jr. Gödel numberings of partial recursive functions. *Journal of Symbolic Logic*, 23:331–341, 1958.

[San98] David Sands. Computing with contexts. In A. D. Gordon, A. M. Pitts, and C. L. Talcott, editors, *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 1998.

[Sch76] Helmut Schwichtenberg. Definierbare functionen im λ-kalkül mit typen. *Arch. Math. Logik*, 17:113–114, 1976.

[Smi94] C. Smith. *A Recursive Introduction to the Theory of Computation.* Springer, 1994.

[Sta86] R. Statman. On translating lambda terms into combinators: the basis problem. In *Proceedings of the 2nd IEEE Conference on Logic in Computer Science*, pages 378–382, 1986.

[Tal93] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112:99–143, 1993.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceeding of the London Mathematical Society*, 42:230–265, 1936.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31, 1979.

A Proof of Confluence, Theorem 3.1

To make the proof easier to present and follow, we define a "conservative extension" of the calculus in which arguments in a list *RHS* are plugged one-at-a-time rather than all at once. Note that UPSILON does *not* satisfy the identity $A^{B,C} \equiv (A^B)^C$, because A^B may have as its first pluggable port one that "comes from" B. We introduce notation for terms in which certain sub-terms are marked "unpluggable" via an over-line. Then we obtain the identity $A^{B,C} \equiv (A^{\overline{B}})^C$. We also employ the meta-syntactic symbol $\odot$ to stand for $\ominus$ or $\oslash$ to signify cases in which it doesn't matter which kind a given port is.

The *extension* is that for any object O, $\overline{O}$ is an "extended object." The other rules for object formation are unchanged, allowing extended objects on their right-hand sides. "Conservative" means that every object A (*sans* over-lines) has the same normal form in the calculus with over-lines.

We also introduce some lettering conventions to clarify the cases used in the lemmas and proofs.

O_i Any object, whether original, extended, over-lined, a proper term, a base object, even $\ominus$ itself.

Q_i Any object with no pluggable ports—such as a base object with no pluggable ports, or a term with no ports that are not in over-lined sub-terms.

A Any object that has a pluggable port. In this section, it will always be the leftmost such object.

L An object that is not a port.

A.1 Plugging

The advantage of our extension is that now we can give a formal inductive definition of E^B, based on the structure of an extended object E. The proofs can then be based on this structure, and the notation for verifying the required identities becomes more manageable than what one would get by basing the proof on the original calculus.

We put off the requirement that all ports of an object be filled by a plug until we define the reduction of Υ in Section A.2. So, E^B has a valance exactly 1 smaller than E.

$\ominus^B \equiv B$

$\oslash^B \equiv \ominus$ (Note that these first two rules don't risk us plugging a naked port on the LHS with terms from the RHS of the same Υ because we take care of that in the reductions section below (Section A.2).

$[Q_1, \ldots, Q_n, A, O_1, \ldots O_m]^B \equiv [Q_1, \ldots, Q_n, A^B, O_1, \ldots O_m]$. (Note that by our lettering convention, A cannot be a base object, since A is the first field value with a pluggable port.)

$(A@O_1)^B \equiv A^B@O_1$.

$(Q_1@A)^B \equiv Q_1@A^B$.

$(\pi_i A)^B \equiv \pi_i(A^B)$.

$(\odot\ \Upsilon^i\ O_1 \ldots O_n)^B \equiv (\odot^B\ \Upsilon^i\ O_1 \ldots O_n)$.

$L\ \Upsilon^i\ Q_1 \ldots Q_n, A, O_1, \ldots O_m)^B \equiv (L\ \Upsilon^i\ Q_1 \ldots Q_n, A^B, O_1, \ldots, O_m)$.

A.2 Reductions

We define a relation $A \overset{\rightarrow}{\Upsilon} B$, meaning "$A$ reduces in at most one step to B," on objects A and B. Note cases where the definition allows many redexes to be expanded in parallel in "one step." This is a notable difference from the λ-calculus, and we look for parallel behavior in UPSILON to be better than that in the λ-calculus in many instances. The rules without double arrows $\Longrightarrow$ are base cases. As above, $O_1, \ldots, O_n$ stand for objects in the extended calculus. Our rules add a case $L\ \Upsilon^{RHS}$ where the list RHS is as a convenient technical device.

Definition A.1.

(a) $O \overset{\rightarrow}{\Upsilon} O$

(b) $$\left.\begin{array}{c} O_1 \overset{\rightarrow}{\Upsilon} O_1' \\ O_2 \overset{\rightarrow}{\Upsilon} O_2' \\ \vdots \\ O_n \overset{\rightarrow}{\Upsilon} O_n' \end{array}\right\} \Longrightarrow [O_1, O_2, \ldots, O_n] \overset{\rightarrow}{\Upsilon} [O_1', O_2', \ldots, O_n']$$

(c) $\left.\begin{array}{l} O_1 \underset{1}{\rightsquigarrow} O_1' \\ O_2 \underset{1}{\rightsquigarrow} O_2' \end{array}\right\} \Longrightarrow O_1 @ O_2 \underset{1}{\rightsquigarrow} O_1' @ O_2'$

(d) $[O_1, O_2, \ldots, O_n] @ [O_{n+1}, O_{n+2}, \ldots, O_m] \underset{1}{\rightsquigarrow} [O_1, O_2, \ldots, O_n, O_{n+1}, O_{n+2}, \ldots, O_m]$

(e) $O \underset{1}{\rightsquigarrow} O' \Longrightarrow \pi_i(O) \underset{1}{\rightsquigarrow} \pi_i(O')$

(f) $\pi_i[\underbrace{\ldots}_{i-1}, O_i, \ldots] \underset{1}{\rightsquigarrow} O_i$, provided $O_i \neq \odot$.

(g) $\left.\begin{array}{c} O_0 \underset{1}{\rightsquigarrow} O_0' \\ O_1 \underset{1}{\rightsquigarrow} O_1' \\ O_2 \underset{1}{\rightsquigarrow} O_2' \\ \vdots \\ O_n \underset{1}{\rightsquigarrow} O_n' \end{array}\right\} \Longrightarrow O_0 \Upsilon^i O_1 O_2 \ldots O_n \underset{1}{\rightsquigarrow} O_0' \Upsilon^i O_1' O_2' \ldots O_n'$

(h) Provided L is reduced, $\lceil \text{valance}(L) \div n \rceil = i$, $L, O_1, \ldots, O_n \neq \odot$, and $n \geq 1$, $L \Upsilon O_1 O_2 \ldots O_n \underset{1}{\rightsquigarrow} (L^{\overline{O_1}}) \Upsilon O_2 \ldots O_n \ (O_1 O_2 \ldots O_n)^{i-1}$.

(i) Provided valance$(L) = 0$, $L \Upsilon O_1, \ldots, O_n \underset{1}{\rightsquigarrow}$ **deline**(L), where **deline**(L) means L with all over-lines removed from sub-terms.

Rules (h) and (i) need further explication. Assuming that the superscripts to Υ tell us the valance of the LHS, Rule (i) incorporates

$$\begin{array}{rcl} L \Upsilon O_1 O_2 \ldots O_n & \underset{1}{\rightsquigarrow} & (L^{\overline{O_1}}) \Upsilon O_2 \ldots O_n, \\ L \Upsilon^i O_1 & \underset{1}{\rightsquigarrow} & (L^{\overline{O_1}}) \Upsilon^{i-1} O_1, \quad \text{and} \\ L \Upsilon O_1 & \underset{1}{\rightsquigarrow} & (L^{\overline{O_1}}) \Upsilon^0 O_1 \end{array}$$

which in turn sets up the terminal case (i). The fine point is why we can stipulate that the terminal case removes *all* over-lines from L. The reason is that any sub-term with over-lines in it is part of a term whose highest operator is an Υ that forms a *redex*, since it came from a term $L_0 \Upsilon RHS$ in which the Υ was expandable. By induction one can prove that all objects that can arise by reduction in the extended calculus starting from a non-extended object have over-lines only if they are reducible. Since the Υ cannot be expandable if L_0 is reducible, it follows that L_0 must have no over-lines. Moreover—and crucially—it is not possible to introduce other over-lines into those terms $L \Upsilon RHS'$ that arise in the course of expanding $L_0 \Upsilon RHS$ one argument at a time. This is covered both by the prohibition on plugging into a non-reduced term and (for L in particular) by the rule against plugging into L in $L \Upsilon RHS'$ when L isn't a port. Thus the only over-lines in L are those introduced by the serial treatment of $L_0 \Upsilon RHS$, and it is proper to remove them when the end of the list RHS is reached.

Thus all the extension does is provide a means of book-keeping the one-at-a-time progress of the expansion, and this is all we want.

A.3 Confluence Theorem

Lemma A.1. $O \underset{1}{\rightsquigarrow} O' \rightarrow A^O \underset{1}{\rightsquigarrow} A^{O'}$

Proof: We prove this by induction on the structure of A.

Case 1 $A = \ominus$ In this case, $A^O = O$ and $A^{O'} = O'$. Since we already know that $O \underset{1}{\rightsquigarrow} A'$, we know that $A^O \underset{1}{\rightsquigarrow} A^{O'}$.

Case 2 $A = \oslash$ In this case, $A^O = A^{O'} = \ominus$.

Case 3 $A = [Q_1, \ldots Q_n, A_1, O_1, \ldots O_m]$ We know that A_1 must be a term or a port, because otherwise it would have been over-lined, and thus an unpluggable Q_i.
In this case, $A^O = [Q_1, \ldots Q_n, A_1, O_1, \ldots O_m]^O \equiv [Q_1, \ldots Q_n, A_1^O, O_1, \ldots O_m]$.
At the same time, $A^{O'} = [Q_1, \ldots Q_n, A_1, O_1, \ldots O_m]^{O'} \equiv [Q_1, \ldots Q_n, A_1^{O'}, O_1, \ldots O_m]$.
From the Induction hypothesis, we can say that: $A_1^O \rightsquigarrow_1 A_1^{O'}$.
And therefore, $[Q_1, \ldots Q_n, A_1^O, O_1, \ldots O_m] \rightsquigarrow_1 [Q_1, \ldots Q_n, A_1^{O'}, O_1, \ldots O_m]$. Hence, $A^O \rightsquigarrow_1 A^{O'}$.

Case 4 $A = (A_1 @ O_1)$ In this case, $A^O = (A_1 @ O_1)^O \equiv (A_1^O @ O_1)$ At the same time, $A^{O'} = (A_1 @ O_1)^{O'} \equiv (A_1^{O'} @ O_1)$. From the Induction Hypothesis we can say: $A_1^O \rightsquigarrow_1 A_1^{O'}$, Thus we know that $(A_1^O @ O_1) \rightsquigarrow_1 (A_1^{O'} @ O_1)$. Therefore, $A^O \rightsquigarrow_1 A^{O'}$.

Case 5 $A = (Q_1 @ A_1)$ In this case, $A^O = (Q_1 @ A_1)^O \equiv (Q_1 @ A_1^O)$ At the same time, $A^{O'} = (Q_1 @ A_1)^{O'} \equiv (Q_1 @ A_1^{O'})$. From the Induction Hypothesis we can say: $A_1^O \rightsquigarrow_1 A_1^{O'}$, Thus we know that $(Q_1 @ A_1^O) \rightsquigarrow_1 (Q_1 @ A_1^{O'})$. So, $A^O \rightsquigarrow_1 A^{O'}$.

Case 6 $A = (\pi_i A_1)$ In this case, $A^O = (\pi_i A_1)^O \equiv (\pi_i A_1^O)$ At the same time, $A^{O'} = (\pi_i A_1)^{O'} \equiv (\pi_i T^{O'})$. From the Induction Hypothesis we can say: $A_1^O \rightsquigarrow_1 A_1^{O'}$, Thus we know that $(\pi_i A_1^O) \rightsquigarrow_1 (\pi_i A_1^{O'})$. Hence, $A^O \rightsquigarrow_1 A^{O'}$.

Case 7 $A = (\ominus\ \Upsilon^i\ O_1 \ldots O_n)$
In this case, $A^O = (\ominus\ \Upsilon^i\ O_1 \ldots O_n)^O \equiv (O\ \Upsilon^i\ O_1 \ldots O_n)$
At the same time, $A^{O'} = (\ominus\ \Upsilon^i\ O_1 \ldots O_n)^{O'} \equiv (O'\ \Upsilon^i\ O_1 \ldots O_n)$.
And we know that $(O\ \Upsilon^i\ O_1 \ldots O_n) \rightsquigarrow_1 (O'\ \Upsilon^i\ O_1 \ldots O_n)$. Thus, $A^O \rightsquigarrow_1 A^{O'}$.

Case 8 $A = (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1 O_1 \ldots O_n)$. In this case,
$A^O = (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1 O_1 \ldots O_n)^O \equiv (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1^O O_1 \ldots O_n)$. At the same time,
$A^{O'} = (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1 O_1 \ldots O_n)^{O'} \equiv (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1^{O'} O_1 \ldots O_n)$. From the Induction Hypothesis we can say: $A_1^O \rightsquigarrow_1 A_1^{O'}$, Thus we know that
$(O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1^O O_1 \ldots O_n) \rightsquigarrow_1 (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1^{O'} O_1 \ldots O_n)$. Therefore, $A^O \rightsquigarrow_1 A^{O'}$.

■

Lemma A.2. $\rightsquigarrow_1$ *satisfies the diamond property. i.e.*

$$\forall O.O \rightsquigarrow_1 O_1,\ \textit{and}\ O \rightsquigarrow_1 O_2 \rightarrow \exists O_3.O_1 \rightsquigarrow_1 O_3\ \textit{and}\ O_2 \rightsquigarrow_1 O_3$$

Proof: The proof is by induction on $O \rightsquigarrow_1 O_1$. For this proof, π_i and R_i represent arbitrary objects.

Case 1 $O \rightsquigarrow_1 O_1 = O \rightsquigarrow_1 O$ So, if $O \rightsquigarrow_1 O_2$ We can make $O_3 = O_2$.

Case 2 $O \rightsquigarrow_1 O_1 = [P_1, \ldots P_n] \rightsquigarrow_1 [P_1', \ldots, P_n']$ where $\forall i.\pi_i \rightsquigarrow_1 P_i'$. Since O is a base object, this is the only type of reduction we can do on it. O_2 must be similar, although the P_i' may be different. So, we have: $O \rightsquigarrow_1 O_2 = [P_1, \ldots P_n] \rightsquigarrow_1 [P_1'', \ldots, P_n'']$ with
$\forall i.\pi_i \rightsquigarrow_1 P_i'$ and $\pi_i \rightsquigarrow_1 \pi_i''$. By the Induction Hypothesis, we know that
$\exists P_i'''.P_i' \rightsquigarrow_1 P_i'''$ and $P_i'' \rightsquigarrow_1 P_i'''$. Thus
$[P_1', \ldots P_n'] \rightsquigarrow_1 [P_1''', \ldots, P_n''']$ and $[P_1'', \ldots P_n''] \rightsquigarrow_1 [P_1''', \ldots, P_n''']$. So, $O_3 = [P_1''', \ldots, P_n''']$.

Case 3 $O \rightsquigarrow_1 O_1 = P_1 @ P_2 \rightsquigarrow_1 P_1' @ P_2'$ With $P_1 \rightsquigarrow_1 P_1', P_2 \rightsquigarrow_1 P_2'$

Case 3.1 $O \rightsquigarrow_1 O_2 = P_1 @ P_2 \rightsquigarrow_1 P_1'' @ P_2''$, with $P_1 \rightsquigarrow_1 P_1''$, and $P_2 \rightsquigarrow_1 P_2''$. So, just like in Case 2, by the Induction Hypothesis there must be P_1''' and P_2''' such that $P_1' \rightsquigarrow_1 P_1''', P_1'' \rightsquigarrow_1 P_1'''$ and $P_2' \rightsquigarrow_1 P_2''', P_2'' \rightsquigarrow_1 P_2'''$. Therefore, we can make $O_3 = P_1''' @ P_2'''$.

Case 3.2 The other alternative is that O is actually
$[P_1, P_2, \ldots, P_n] @ [R_1, R_2, \ldots, R_m]$, and that $O \rightsquigarrow_1 O_2$ is actually
$[P_1, P_2, \ldots, P_n] @ [R_1, R_2, \ldots, R_m] \rightsquigarrow_1 [P_1, P_2 \ldots P_n, R_1, R_2, \ldots, R_m]$. In this case O_1 must have been

$[P'_1, P'_2, \dots, P'_n]@[R'_1, R'_2, \dots, R'_m]$, Since, as in Case 2, that's the only way we can reduce base objects.
So, we have:
$[P_1, P_2, \dots, P_n]@[R_1, R_2, \dots, R_m] \overset{\leadsto}{\scriptscriptstyle 1} [P'_1, P'_2, \dots, P'_n]@[R'_1, R'_2, \dots, R'_m]$ and
$[P_1, P_2, \dots, P_n]@[R_1, R_2, \dots, R_m] \overset{\leadsto}{\scriptscriptstyle 1} [P_1, P_2 \dots P_n, R_1, R_2, \dots, R_m]$.
We can make $O_3 = [P'_1, P'_2 \dots P'_n, R'_1, R'_2, \dots, R'_m]$.

Case 4 $O \overset{\leadsto}{\scriptscriptstyle 1} O_1 = \pi_i(P) \overset{\leadsto}{\scriptscriptstyle 1} \pi_i(P')$ with $P \overset{\leadsto}{\scriptscriptstyle 1} P'$.

Case 4.1 $O \overset{\leadsto}{\scriptscriptstyle 1} O_2 = \pi_i(P) \overset{\leadsto}{\scriptscriptstyle 1} \pi_i(P'')$ with $P \overset{\leadsto}{\scriptscriptstyle 1} P''$. Again by the inductive hypothesis, there is some P''' such that $P' \overset{\leadsto}{\scriptscriptstyle 1} P'''$ and $P'' \overset{\leadsto}{\scriptscriptstyle 1} P'''$. So, $O_3 = \pi_i(P''')$.

Case 4.2 The other alternative is that O is actually a base object, and $O \overset{\leadsto}{\scriptscriptstyle 1} O_2 = \pi_i[P_1 \dots, P_i, \dots, P_n] \overset{\leadsto}{\scriptscriptstyle 1} P_i$. In this case, we can make $O_3 = P'_i$.

Case 5 $O \overset{\leadsto}{\scriptscriptstyle 1} O_1 = P_0\ \Upsilon^i\ P_1 \dots P_n \overset{\leadsto}{\scriptscriptstyle 1} P'_0\ \Upsilon^i\ P'_1 \dots P'_n$ with $\forall j. P_j \overset{\leadsto}{\scriptscriptstyle 1} P'_j$.

Case 5.1 $O \overset{\leadsto}{\scriptscriptstyle 1} O_2 = P_0\ \Upsilon^i\ P_1 \dots P_n \overset{\leadsto}{\scriptscriptstyle 1} P''_0\ \Upsilon^i\ P''_1 \dots P''_n$ with $\forall j. P_j \overset{\leadsto}{\scriptscriptstyle 1} P''_j$. Again, by the Induction Hypothesis, we know that $\forall j. \exists P'''_j . P'_j \overset{\leadsto}{\scriptscriptstyle 1} P'''_j$ and $P''_j \overset{\leadsto}{\scriptscriptstyle 1} P'''_j$. So, we can make $O_3 = P'''_0\ \Upsilon^i\ P'''_1 \dots P'''_n$.

Case 5.2 The other option is that $O \overset{\leadsto}{\scriptscriptstyle 1} O_2 = P_0\ \Upsilon^i\ P_1 \dots P_n \overset{\leadsto}{\scriptscriptstyle 1} P_0 \overbrace{\underbrace{\qquad}_{i}}^{P_1 \dots P_n}$, P_0 is actually stable (ie it can't reduce anymore) and none of the P_i are ports. Notice that in this case $P'_0 = P_0$.
So, we can make $O_3 = P_0 \underbrace{P'_1 P'_1 \dots P'_1}_{i} \underbrace{P'_2 P'_2 \dots P'_2}_{i} \dots\dots \underbrace{P'_n P'_n \dots P'_n}_{i}$. From the previous lemma and the Induction Hypothesis, we know that both O_1 and O_2 reduce to this.

∎

Let $\leadsto$ be the transitive closure of $\overset{\leadsto}{\scriptscriptstyle 1}$ and $\approx$ the equality relation that it induces.

Theorem A.3 (Confluence Property (Church-Rosser Theorem)). *1. $\leadsto$ is confluent.*

2. $M \approx N \Rightarrow \exists Z[M \leadsto Z$ and $N \leadsto Z]$

Proof: Plugging in the above Lemma A.2 in place of Lemma 3.2.6 of Barendregt [Bar84], and the definitions above for his Lemma 3.2.7, makes the confluence proof in [Bar84] carry over without incident. ∎

B Recursion Theorem

Our internal proof of the recursion theorem follows that of [Smi94]. Say we are looking at an object i with two holes and we are trying to create a version with a copy of itself inside filling the first port.

Theorem B.1 (Recursion Theorem). *For every object i which takes a single argument, which is a two field object, there is another object e such that for all argument objects x, $e\ \Upsilon\ x = i\ \Upsilon\ [e, x]$.*

Proof: First we need a $s_{1,1}$ object (which we shorten to s). The s object is basically a storing method. It takes an object with two holes (i) and an argument to fill one of the holes (x). It fills the first hole, storing the argument as a constant value in the original object, creating an object with only one hole for the other argument (y). This is closely related to the $\texttt{Curry}_2$ object, defined in Section 4.3, however we need to simplify it a bit:

$$s_{1,1} = (((\ominus_i \Upsilon \ominus_x \ominus_y) \Upsilon \pi_1(\pi_1(\ominus_i)) \pi_1(\pi_1(\ominus_x)) \ominus_y) \Upsilon$$
$$[[\ominus] \Upsilon \ominus_i] [[\ominus] \Upsilon \ominus_x] \oslash_y)$$

Next, we let j be an object with three holes. The second gets plugged twice into s. Both this and the third argument, then get boxed into an object and plugged into the first argument. So, j looks like:

$$j = (\ominus_1 \Upsilon [\ominus_{s(2,2)}, \ominus_3]) \Upsilon \ominus_1((s \Upsilon \ominus_2\ominus_2) \Upsilon^2 \ominus_2)\ominus_3$$

Now, we let $v = s \Upsilon ji$, and let $e = s \Upsilon vv$.

$$\begin{array}{rcl}
e = s \Upsilon vv & = & (((\ominus \Upsilon \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus) \Upsilon [[\ominus] \Upsilon \ominus] [[\ominus] \Upsilon \ominus] \oslash) \overset{\downarrow}{\Upsilon} vv \\
& \rightsquigarrow & (((\ominus \Upsilon \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus) \Upsilon [[\ominus] \overset{\downarrow}{\Upsilon} v] [[\ominus] \overset{\downarrow}{\Upsilon} v] \ominus) \\
& \rightsquigarrow & (((\ominus \Upsilon \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus) \Upsilon [[v]] [[v]] \ominus)
\end{array}$$

Now, we plug e with some argument x:

$$\begin{array}{rcl}
e \Upsilon x & = & (((\ominus \Upsilon \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus) \Upsilon [[v]] [[v]] \ominus) \overset{\downarrow}{\Upsilon} x \\
& \rightsquigarrow & (((\ominus \Upsilon \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus) \overset{\downarrow}{\Upsilon} [[v]] [[v]] x) \\
& \rightsquigarrow & ((\ominus \Upsilon \ominus \ominus) \Upsilon \overset{\downarrow}{\pi_1} (\overset{\downarrow}{\pi_1} ([[v]]))\ \overset{\downarrow}{\pi_1} (\overset{\downarrow}{\pi_1} ([[v]])) x) \\
& \rightsquigarrow & ((\ominus \Upsilon \ominus \ominus) \overset{\downarrow}{\Upsilon} v\ v\ x) \\
& \rightsquigarrow & v \Upsilon v\ x
\end{array}$$

Now we need to expand out the first v in order to continue. Notice that since j has three holes instead of two, we must use $s_{1,2}$ which stores the first argument, but leaves the other two to be specified later, rather than $s_{1,1}$ as used elsewhere:

$$\begin{array}{rcl}
v & = & s_{1,2} \Upsilon ji = (((\ominus \Upsilon \ominus \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus \ominus) \Upsilon [[\ominus] \Upsilon \ominus] [[\ominus] \Upsilon \ominus] \oslash \oslash) \overset{\downarrow}{\Upsilon} j\ i \\
& \rightsquigarrow & (((\ominus \Upsilon \ominus \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus \ominus) \Upsilon [[\ominus] \overset{\downarrow}{\Upsilon} j] [[\ominus] \overset{\downarrow}{\Upsilon} i] \ominus \ominus) \\
& \rightsquigarrow & (((\ominus \Upsilon \ominus \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus \ominus) \Upsilon [[j]] [[i]] \ominus \ominus)
\end{array}$$

Now, we look back at $e = v \Upsilon vx$

$$\begin{array}{rcl}
e & = & v \Upsilon vx = (((\ominus \Upsilon \ominus \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus \ominus) \Upsilon [[j]] [[i]] \ominus \ominus) \overset{\downarrow}{\Upsilon} v\ x \\
& \rightsquigarrow & (((\ominus \Upsilon \ominus \ominus \ominus) \Upsilon \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus \ominus) \overset{\downarrow}{\Upsilon} [[j]] [[i]]\ v\ x) \\
& \rightsquigarrow & ((\ominus \Upsilon \ominus \ominus \ominus) \Upsilon \overset{\downarrow}{\pi_1} (\overset{\downarrow}{\pi_1} ([[j]]))\ \overset{\downarrow}{\pi_1} (\overset{\downarrow}{\pi_1} ([[i]]))\ v\ x) \\
& \rightsquigarrow & ((\ominus \Upsilon \ominus \ominus \ominus)\ P\overset{\downarrow}{l}ug\ j\ i\ v\ x) \\
& \rightsquigarrow & (j \Upsilon i\ v\ x)
\end{array}$$

Now, we need to expand j:

$$\begin{array}{rcl}
e & = & j \Upsilon i\ v\ x = ((\ominus \Upsilon [\ominus, \ominus]) \Upsilon \ominus((s \Upsilon \ominus\ominus) \Upsilon^2 \ominus)\ominus) \overset{\downarrow}{\Upsilon} ivx
\end{array}$$

$$
\begin{aligned}
&\rightsquigarrow (\ominus\ \Upsilon\ [\ominus,\ominus])\ \Upsilon\ i((s\ \Upsilon\ \ominus\ominus)\ \overset{\downarrow}{\Upsilon}{}^{2}\ v)x \\
&\rightsquigarrow (\ominus\ \Upsilon\ [\ominus,\ominus])\ \Upsilon\ i(s\ \Upsilon\ vv)x
\end{aligned}
$$

But $(s\ \Upsilon\ vv) = e$ so,:

$$
\begin{aligned}
e &\rightsquigarrow (\ominus\ \Upsilon\ [\ominus,\ominus])\ \overset{\downarrow}{\Upsilon}\ iex \\
&\rightsquigarrow i\ \Upsilon\ [e,x]
\end{aligned}
$$

Which is exactly what we expected it to be.

■