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 2001

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. The differences in focus of the models have had great effects on the style and uses of the programming languages that are based on them. One of the most important differences is where the model fits 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). Additionally, the stored-program computer realizes the interchangeability of program and data concretely. Every model stakes out a position in how "program" and "data" are treated, and this position greatly affects the model's design.

The λ calculus is a simple model of computation in large part because it lies at one end of this continuum. It has only functions. There are no inherent values in this model— rather, functions can be treated as values. The lambda calculus is the model of computation that is the basis for the functional programming languages. The style of programming in these languages is determined by the position of the this model at one end of the continuum. This placement also has a significant effect on the techniques that are used to prove things about the languages and the programs written in them.

In the middle of the continuum lie the random access machines (RAM) model. RAM programs explicitly 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.

Our initial goal was to create a model of computation at the other end of the spectrum where there is only data. Here there was no corresponding fundamental model at the level of λ-calculus or Turing or RAM machines. 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 came from neither an already existing model of computation (like Functional programming) nor purely out of software practice (like Structured programming). Instead, 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-centric 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.

We base our model on objects which are *nest-able tuples* i.e. tuples whose fields can themselves be tuples.

What distinguishes our model is its concept of *incomplete objects* and manner of performing computation on them without any program. An incomplete object has holes called *ports* in it. A port can be a field or part of a larger term. The concept 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 be filled themselves.

These insights lead to a model of computation that is Turing-complete and confluent, but that has no control flow, and is thereby completely data-flow. In addition to its novelty, our model has three points of interest. First, we demonstrate some promise as the base of a model for object-oriented programming, by outlining basic programming mechanisms and building a type system that includes subtyping. Additionally, our model can express *self* in a natural way. Second, our model is allied with context calculi, and in some sense has nothing other than contexts to compute with. Third, as a data-flow language, it provides a new angle on how data flows into contexts.

The remainder of the paper is organized as follows, Section 2 recounts prior work in the areas of object oriented models of computation, contexts, and data-flow computation. Section 3 gives a complete description of the model. Section 4 gives some simple examples of how to represent common object types. Section 5 shows the relation between our model and other, better known models of computation. Section 6 gives a brief argument for our model being considered Object Oriented. Section 7 adds simple types to the model. Section 8 describes alternative models, giving motivation to the current version of the model. Section 9 describes the relation between our work and the context literature. Section 10 gives future work and conclusions. More formal descriptions and detailed proofs are in the appendices.

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 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 arithmetic, 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 [AC96] 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. Finally, Castagna et. al. created a variation on the λ calculus, called $\lambda\&$, introduced in [CGL95], and extended in [Cas97], which models objects not as records which include their own methods, as the above do, but as records that external overloaded functions (which might be thought of as sets of functions) may act upon. Instead of methods being grouped with their objects, they are grouped by method name, as in languages like CLOS. In this way they explicitly study method overloading and multiple dispatch. 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

We defined an object as a tuple. We denote a tuple with fields (say) A, B, C by $[A, B, C]$. The *arity* of a tuple is the number of fields. The *empty tuple*, for example, is denoted by [ ] and has arity 0.

Two natural operations on tuples are *append*, which joins two tuples together, denoted by $[A, B, C]@\,[D, E] = [A, B, C, D, E]$, and a family of *field extraction* operators, which retrieve a particular field from an object, exemplified by $\pi_1([A, B, C, D, E]) = A$ and $\pi_4([A, B, C, D, E]) = D$ (we number from 1). $\pi_i(T)$ is an error if T is a tuple with arity less than i. If T has arity at least i and the ith field is empty (i.e. it contains a port, described next), then $\pi_i(T)$ is not an error but rather *inhibited*. If and when that field is filled with an object A, then $\pi_i(T)$ can evaluate to A. Henceforth we refer to expressions such as $\pi_3([A, B, C, D])$ as *object terms* to distinguish them from *base objects* such as $[A, B, C, D]$ itself, which have $[\ldots]$ outermost. Collectively everything except a stand-alone port is called an *object*.

We also have two different types of holes, a *port* $\ominus$ and a *locked port* $\oslash$, and a means of filling the holes, the *plug* operator, Υ. The difference between a port and a locked port is what happens when they are filled, or *plugged*. The basic idea is that $A\ \Upsilon\ B_1\ B_2\ \ldots\ B_m$ stands for the object obtained by filling the ports of A

by the objects $B_1, \ldots, B_m$. 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. When a locked port $\oslash$ is plugged by an object B_i the result is not B_i but an unlocked port $\ominus$. If there are more than m ports available then filling re-commences at the beginning of the argument list, while in case of fewer than m the leftover arguments are discarded. All available ports in the original A are filled after a plugging operation has taken place. Allowing explicit control over how many cycles take place, and un-filled holes to remain after a plug operation lead to problems in the type theory, as seen in Section 8.1.

The main detail is determining which ports in an object are open to receive arguments. We first stipulate that any port inside more than one level of $[\ldots]$ is "shielded" and hence not *pluggable*. This agrees with the idea that only the top-level fields of an object can be accessed with one application of the "." operator in a language such as C++, Java, or Ada. Second, if $A \curlyvee B_1 \ \ldots \ B_m$ is on the left hand side of another plug operation, then ports in A are shielded from receiving arguments from from that plug. All ports in A must be filled immediately by $B_1, \ldots, B_m$—while ports in $B_1, \ldots, B_m$ themselves are not thus shielded, so that these arguments may evolve before being plugged into A. This restriction makes it easier to pre-determine which arguments will go to which ports in A.

Additionally, since $\ominus$ (or $\oslash$) denotes the *absence* of an object, rather than an object itself, $\ominus \curlyvee B$ is inhibited from evaluating. This represents the case where the invoking object has not yet been supplied. It is possible to plug the port in $\ominus \curlyvee B$ from the outside in a compound term such as $(\ominus \curlyvee B) \curlyvee A$, which may reduce to $A \curlyvee B$ if B itself has no pluggable ports. Likewise, if any of B_i in the above is $\ominus$ (or $\oslash$), the $\curlyvee$ is inhibited because the invoker does not yet know where to obtain the corresponding argument if it is needed.

The following illustrations hold for any objects A, B, and C:

$$\begin{aligned}
[\ominus] \curlyvee B &= [B] \\
[\ominus, \ominus] \curlyvee B &= [B, B] \\
[\ominus, \ominus, \ominus, \ominus] \curlyvee B\ C\ D &= [B, C, D, B] \\
[\ominus, [\ominus, \ominus], \ominus] \curlyvee B\ C\ D &= [B, [\ominus, \ominus], C] \\
[\ominus, [\ominus, \oslash], \oslash] \curlyvee B\ C\ D &= [B, [\ominus, \oslash], \ominus].
\end{aligned}$$

The final detail in our model is that $A \curlyvee B_1 \ \ldots \ B_n$ may reduce only when the invoker A is already reduced, but does not require any of $B_1 \ \ldots \ B_n$ to be reduced. [CGL95] has a very similar requirement, that the target of a method invocation must be fully reduced before it can be sent to the message. They require this so that the final type of the target is known at method invocation time to allow for late binding. We require it for confluence, as can be seen in the following example:

$$([\ominus, \ominus] \curlyvee \pi_1(\ominus)) \curlyvee B\ C$$

This could be reduced by expanding the right-hand $\curlyvee$ to get $[\ominus, \ominus] \curlyvee \pi_1(B)$ (losing the C because there is only one visible port) and finally $[\pi_1(B), \pi_1(B)]$, instead of the correct reduction to $[\pi_1(\ominus), \pi_1(\ominus)] \curlyvee B\ C$, and thence to $[\pi_1(B), \pi_1(C)]$.

This does leave us free to pursue lazy-evaluation strategies with regard to arguments, just not with regard to target objects. Under the analogy between $A \curlyvee B$ in UPSILON and $(\lambda x.A)B$ (which is not perfect—see Section 5.2 on currying below), this restriction corresponds to *weak reduction* **REFERENCE PLEASE?** in the λ-calculus. Here is an example that illustrates much of the above behavior:

$$\begin{aligned}
\pi_1(\pi_2((\ominus \curlyvee [\ominus]\ [[A]] \oslash) \curlyvee [\ominus, \ominus, \ominus]\ [B])) \curlyvee C &= \pi_1(\pi_2([\ominus, \ominus, \ominus] \curlyvee [[B]]\ [[A]] \ominus)) \curlyvee C \\
&= \pi_1(\pi_2([\ominus, \ominus, \ominus] \curlyvee [[B]]\ [[A]]\ C)) \\
&= \pi_1(\pi_2([[[B]], [[A]], C])) = \pi_1([[A]]) = [A].
\end{aligned}$$

Here the $\ominus$ to the left of the first $\curlyvee$ representing a missing invoker is filled by $[\ominus, \ominus, \ominus]$ on expanding the middle $\curlyvee$. The third $\curlyvee$ could not expand because its invoker was expandable. The *valence* of an object its

number of pluggable ports, so the valence of the invoker of the middle Υ is 3, regardless of what A is, owing to the "double-shielding" in $[[A]]$. Since there are two arguments, the first argument $[\ominus, \ominus, \ominus]$ is replicated—and unlocks the locked port $\oslash$, leaving $\ominus$ in its place—while the second argument $[B]$ is not (or is "thrown away"). Note also that the argument $[B]$ acquires double-shielding by being plugged into $[\ominus]$. Next, the first Υ was still inhibited, by having an absent third argument, but the outer Υ becomes expandable. The ports in the $[\ominus, \ominus, \ominus]$ to its left are not pluggable, by the rule against plugging into an invoker from outside, and since those in $[[A]]$ and $[[B]]$ are shielded, C goes only into the port which was unlocked. Finally the leading Υ becomes expandable, and since the result is a base object whose first field is nonempty, the outer $\pi_1\pi_2$ is expandable to leave $[A]$.

With this all said, the resulting model is called UPSILON, standing for "Universal Programming System with Incomplete, Lazy Object Notation."

3.1 Computation in UPSILON

Every UPSILON object is a program, and computation proceeds if the object can *reduce*. A *redex* is defined in a fairly standard fashion:

1. $A@B$ is a redex if both A and B are base objects.
2. $\pi_i(A)$ is a redex if A is a base object with at least i-many fields, and the i-th field of A holds an object, not a port. (An error results if there are fewer than i fields in A.)
3. $A \; \Upsilon \; B_1 \; \ldots \; B_n$ is a redex if A is in normal form and neither A nor any B_i is a port.

An object is *reduced*, or *in normal form*, if there are no redexes inside of it. A redex *reduces* in the obvious ways, given the semantics of the previous section. These definitions conform to the intuitive descriptions given in the previous section, and *reductions* are defined (formally in Appendix A.2) as expected. The definitions are also well-founded, with no circularity from "A is in normal form."

There are two basic kinds of reduced objects (not counting ports as objects): base objects whose fields are all reduced, and proper terms containing operations that are inhibited because some field or argument or invoker is a port. Even with the restriction that the left hand side of a plug operation must be in normal form for the operator to reduce, an object may have more than one redex. The order of expanding them does not matter:

Theorem 3.1. *Reduction in* UPSILON *has the* Church-Rosser property *and is therefore* confluent*, meaning that if an object A reduces to an object B in normal form, then all terminating reduction sequences yield B. [A more complete version of this result, and a proof are given in Appendix A]*

Since UPSILON has no names, we do not need to say that the object B is "unique up to α-equivalence" as in the λ-calculus; it is simply and syntactically unique.

We say more about the motivations for our rules and discuss variants that still preserve confluence later on.

4 Examples

Now we give examples of computation in UPSILON and essential programming primitives. The modeling of Booleans is familiar from the λ-calculus, but our later examples and simulation of RAMs show off some advantages of UPSILON for expressing object oriented concepts.

4.1 Booleans

Whereas Abadi and Cardelli [AC96] assume booleans as a basic type, we build them from more-primitive components, combining aspects of the λ-calculus and *Smalltalk*.

Define $\texttt{True} = [\pi_1([\ominus, \ominus])]$ and $\texttt{False} = [\pi_2([\ominus, \ominus])]$. We interpret `True` as a unary base object whose sole field holds a "then" method, and `False` as a base object holding only an "else" method. 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.

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

This envisions three arguments A, B, C, with A Boolean. Then `If-Then-Else` Υ A B C reduces to $\pi_1(A)$ Υ 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 held by A 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 `If-Then-Else` Υ A B C, and employ natural names for other fixed UPSILON objects and terms, such as the following Boolean operations which are partial completions of `If-Then-Else`. We also use standard function notation for outermost arguments, e.g. writing $\texttt{Or}(A, B)$ for `Or` Υ A B.

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

Note that `Not` is the first time that we actually use the feature that we can plug more objects than are needed. The final port in `Not` is purely for control. If it weren't there, then the outermost 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 the original is the most elegant version.

Also, note that the ports within the occurrences of `True` and `False` in these operators are not pluggable because they are "shielded" by two levels of $[\ldots]$, so that when (e.g.) $\texttt{Or}(A, B)$ 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 expanded redexes marked by an arrow.

$$\begin{aligned}
((\ominus \Upsilon \ominus \texttt{False}) \Upsilon \pi_1(\ominus) \ominus) \overset{\downarrow}{\Upsilon}\ \texttt{True}\ \texttt{False} &\rightsquigarrow (\ominus \Upsilon \ominus \texttt{False}) \overset{\downarrow}{\Upsilon}\ \pi_1(\texttt{True})\ \texttt{False} \\
\rightsquigarrow \pi_1(\overset{\downarrow}{\texttt{True}})\ \Upsilon\ \texttt{False}\ \texttt{False} \rightsquigarrow (\overset{\downarrow}{\pi_1}\ ([\pi_1[\ominus, \ominus]]))\ \Upsilon\ \texttt{False}\ \texttt{False} &\rightsquigarrow (\pi_1([\ominus, \ominus]))\ \Upsilon\ \texttt{False}\ \texttt{False} \\
&\rightsquigarrow \pi_1([\texttt{False}, \texttt{False}]) \rightsquigarrow \texttt{False}.
\end{aligned}$$

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 interpreted as an object that knows whether or not it is 0, knows its predecessor, and knows how to find its successor. The first two can be constant fields, while the third must be a method. The third field holds

$$\texttt{Succ} = [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon \ominus.$$

Here an object n plugged into the rightmost port of `Succ` will be replicated twice in the base object—we could write Υ^2 as a reminder of that. The first copy expresses that n is the predecessor of $\texttt{Succ}(n)$, while the π_3 applied to the second copy moves `Succ` itself into the third field of the result, which thus becomes the integer $n + 1$. For example,

$$\texttt{Zero} \quad = \quad [\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon \ominus],$$

$$
\begin{aligned}
\texttt{One} &= [\texttt{False}, \texttt{Zero}, [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus] \\
&= [\texttt{False}, [\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus], [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus], \\
\texttt{Two} &= \texttt{False}, \texttt{One}, [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus] \\
&= [\texttt{False}, \\
&\quad [\texttt{False}, [\texttt{True}, [\,], [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus], [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus], \\
&\quad [\texttt{False}, \ominus, \pi_3(\ominus)]\ \Upsilon\ \ominus]\,.
\end{aligned}
$$

The term $\texttt{Zero?} = \pi_1(\ominus)$ functions as a test for a number being zero. While the successor method requires several reductions to evaluate, it is still a constant-time operation, and shows how an operation usually regarded as atomic follows from smaller, more simplistic, operations.

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*

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

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 $\texttt{Zero} = [\texttt{True}, \texttt{Zero}, \texttt{Pred}]$ is circular. In Section 5.6 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 accesses 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.

From this we are able to do arithmetic just like in [AC96].

5 Turing Completeness

5.1 RAM Simulation

We take machines with some fixed number k of *counters* as representative of random-access machines. Minsky [Min67] showed that two-counter RAMs are universal, but we have no hardship in allowing $k > 2$. A RAM program is a finite sequence of *instructions*, numbered consecutively from 1. Using N_i for line numbers and R_j for register numbers, we may take the following as the RAM instructions to consider:

N_i **continue**	:	line i does nothing,
N_i **INC** R_j	:	line i increments register j,
N_i **DEC** R_j	:	line i decrements register j,
$N_i\ R_j$ **JMP** N_ℓ	:	if $R_j = 0$ then jump to line ℓ; else continue.
N_i **end**	:	computation halts.

A RAM program must have an **end** statement as its last line and nowhere else—if two RAM programs are concatenated then the **end** statement of the first becomes a **continue**.

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

- The state is represented by an object with n fields, each a simple integer as described above and representing a register.
- We use $\texttt{Id} = \pi_1([\ominus]\ \Upsilon\ \ominus)$ 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.

*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`.

- A line of the program is 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. Furthermore, `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.

For our translation into UPSILON, we employ subscripts and superscripts that are not formally part of the language but are intended solely to aid the reader. For any term T and $n \geq 1$, T^k stands for n consecutive copies of T in an argument list. The superscript in Υ^k, however, is an assertion that the argument objects will be copied exactly k times—and unless otherwise noted, the notation $A\ \Upsilon^k\ B_1\ \ldots\ B_m$ asserts that the receiving object A has exactly mk pluggable ports. Finally, subscripting a port by an object or index is done to help the reader trace which argument is destined for which port when an Υ is expanded. The instruction translations are:

$$\begin{aligned}
N_i\ \textbf{INC}\ R_j &= [\pi_{i+1}(\pi_1(\ominus)), [\pi_1(\ominus), \pi_2(\ominus), \ldots, \texttt{Succ}(\pi_j(\ominus)), \pi_{j+1}(\ominus) \ldots \pi_k(\ominus)]\ \Upsilon^k\ \ominus, \texttt{False}] \\
N_i\ \textbf{DEC}\ R_j &= [\pi_{i+1}(\pi_1(\ominus)), [\pi_1(\ominus), \pi_2(\ominus) \ldots, \texttt{Pred}(\pi_j(\ominus)), \pi_{j+1}(\ominus) \ldots \pi_k(\ominus)]\ \Upsilon^k\ \ominus, \texttt{False}] \\
N_i\ \textbf{continue} &= [\pi_{i+1}(\pi_1(\ominus)), \texttt{Id}(\ominus), \texttt{False}] \\
N_i\ \textbf{end} &= [\pi_i(\pi_1(\ominus)), \texttt{Id}(\ominus), \texttt{True}] \\
N_i\ R_j\ \textbf{JMP}\ N_\ell &= [\texttt{if}\ (\texttt{zero?}(\pi_j(\pi_2(\ominus))))\ \texttt{then}\ \pi_k(\pi_1(\ominus))\ \text{else}\ \pi_{i+1}(\pi_1(\ominus)), \texttt{Id}(\ominus), \texttt{False}].
\end{aligned}$$

In the first two, the next-line method simply returns the next line, and the next-state method does nothing except increment or decrement the appropriate register. In **continue**, the next-line method returns the next line and the next-state method does nothing—while in **end** neither method will ever get called, and there is no loss of generality in the latter looping indefinitely and not changing the state if it ever were called. Finally for **JMP**, if the fingered register holds 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.

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

$$\begin{aligned}
&[\ominus_a\ \Upsilon\ (\ [\pi_1(\ominus_b),\ \pi_2(\ominus_c),\ \ominus_d\ \Upsilon\ ([\pi_2(\ominus_e), \pi_4(\ominus_e)]\ \Upsilon^2\ \ominus_e),\ \ominus_f\ \Upsilon\ \pi_4(\ominus_g)] \\
&\qquad \Upsilon \\
&\qquad \ominus_b\ \ominus_c\ \pi_1(\pi_3(\ominus_d))\ \ominus_e\ \pi_2(\pi_3(\ominus_f))\ \ominus_g \\
&\qquad)\ ] \\
&\Upsilon\ (\pi_1(\ominus))_a\ \ominus_b\ \ominus_c\ \ominus_d\ \ominus_e\ \ominus_f\ \ominus_g.
\end{aligned}$$

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 Composition and Currying

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

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

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 $C_0(f,g,x) = f(g(x))$, it does *not* satisfy $C_0(f,g)(x) = f(g(x))$, as required by (1). Abstractly, the difficulty is that in UPSILON it does not hold for all objects h, x, y that $h\ \Upsilon\ x\ y = (h\ \Upsilon\ x)\ \Upsilon\ y$. Thinking here of h as C_0 and x as standing for f and g together, we can regard the problem with C_0 as a "failure of currying." Locked ports were introduced to fix this problem within UPSILON itself.

A naive use of locked ports, i.e.

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

Also doesn't work because when f and g are plugged into C_1 we get $f\ \Upsilon\ (g\ \Upsilon\ \ominus)$. This can reduce producing unexpected results.

Lemma 5.1 (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 $T' = \texttt{Curry}\ \Upsilon\ T$, where

$$\texttt{Curry} = (((\ominus_T\ \Upsilon\ \ominus^k\ \ominus^\ell)\ \Upsilon\ \pi_1(\pi_1(\ominus_T))\ \pi_1(\pi_1(\ominus))^k \ominus^\ell)\ \Upsilon$$
$$\ominus_T\ [[\ominus]\ \Upsilon\ \ominus]^k\ \oslash^\ell)\ \Upsilon\ [[\ominus]\ \Upsilon\ \ominus]_T\ \oslash^k.$$

Here as before the superscripted k and ℓ stand for sublists of k-many (respectively, ℓ-many) copies of the superscripted sub-object, while the subscript T helps us keep track of which ports are expecting T. Here $[[\ominus]\ \Upsilon\ \ominus]$ 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 ready. Then $\pi_1(\pi_1(\ominus))$ 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, giving what we need. □

Crucially, the locked ports allow us to *wait* for the k arguments after we have plugged in T. Below, in Section 8.2, we prove the interesting fact that without $\oslash$'s there is *no* way to do composition.

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 $K\ \Upsilon\ X = (((\ominus\ \Upsilon\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus)\ \Upsilon\ [[K_0]]\ [[X]]\ \ominus$.

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:

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

Thus while K_0 satisfied $K_0(x,y) = x$, K satisfies $K(x)(y) = x$.

Objects of any valence k may be curried to take one argument at a time:

Corollary 5.2. *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 $x_1, \ldots, x_k$,*

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

Applying currying to the term C_0 defined above leads to the following composition operator.

$$\texttt{Compose} = (((\ominus_f \curlyvee (\ominus_g \curlyvee \ominus_y)) \curlyvee \pi_1(\pi_1(\ominus_f)) \pi_1(\pi_1(\ominus_g)) \ominus_y) \curlyvee [[\ominus] \curlyvee \ominus_f][[\ominus] \curlyvee \ominus_g]\oslash_y)$$

Here again the subscripts say which value each port is expecting. The idea is to encapsulate f and g while unlocking the port for y. When y is plugged in, the outermost $\curlyvee$ can evaluate, unwrapping f and g and plugging them into the final term, which behaves as C_0 from that point on. This gives us $\texttt{Compose}(f, g)(y) = C_0(f, g, y) = f(g(y))$, so `Compose` satisfies (1). This also generalizes to f and g having valence greater than 1.

5.3 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 more readily to simulate *combinators*, which are closed lambda-terms. The above currying tools enable us to do the latter straightaway. 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

$$S_0 = ((\ominus \curlyvee \pi_2([\ominus, \ominus])) \curlyvee (\ominus \curlyvee \ominus) \ominus (\ominus \curlyvee \ominus)) \curlyvee \pi_1([\ominus, \ominus]) \ominus \ominus \ominus \ominus \tag{2}$$

and $S = \texttt{Curry}_k(S_0 \curlyvee \ominus \ominus \ominus)$ from Corollary 5.2. Then for all objects x, y, z,

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

where $(x \curlyvee z) \curlyvee (y \curlyvee z)$ may (of course) reduce further.

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

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

in normal form. Now on presenting any argument object z, this reduces (as required) to

$$\begin{aligned}
(\ominus \curlyvee \ominus \ominus \ominus) \curlyvee S_0\ K\ K\ z &= S_0 \curlyvee K\ K\ z \\
&= (K \curlyvee z) \curlyvee (K \curlyvee z) \\
&= (((\ominus \curlyvee \ominus \ominus) \curlyvee \pi_1(\pi_1(\ominus)) \pi_1(\pi_1(\ominus)) \ominus) \curlyvee [[K_0]]\ [[z]] \ominus) \curlyvee (K \curlyvee z) \\
&= K_0 \curlyvee z\ (K \curlyvee z) = z.
\end{aligned}$$

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 $z \curlyvee z$, not z. This does make SK_0K_0 in UPSILON an expression for the ω-combinator, but a simpler one is

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

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

5.4 Complexity of these simulations

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.

We note here that these simulations are also feasible in standard terms of computational complexity. Within UPSILON itself, there are two complexity measures that stand out:

- The number of reductions made, i.e., of redexes expanded.

- The sum, over all reductions made, of the sizes of the objects involved in the redex.

The latter is a "fair" measure of serial time compared to the former because as objects are plugged into the right-hand arguments of a Υ the size of these arguments may grow—this is particularly relevant when arguments are duplicated.

Above we defined integers in unary notation, and have simulated a form of RAM machine that is known to be exponentially slower than realistic models. However, much as Church numerals can be improved upon for the lambda calculus, we can adapt standard representations of numbers and strings into UPSILON, for example letting a base object with n Boolean fields encode a binary string of length n. With these fixed, we obtain standard notions of UPSILON programs computing a function on strings or numbers and of solving decision problems, and thus associate complexity classes of languages or functions to bounds on the two UPSILON complexity measures. These turn out to be the familiar important complexity classes, including:

Theorem 5.3. *(a) The class of languages/functions computed by* UPSILON *programs running in polynomial fair cost is the same as the class P of languages/functions computed in polynomial time by Turing machines.*

(b) The class of languages/functions computed by polynomial-size UPSILON *programs with a poly-logarithmic number of reductions equals the class NC of languages/functions computed by circuits of polynomial size and poly-logarithmic depth.*

Part (a) already follows from our combinator simulation above, while (b) uses base objects of Booleans to simulate vector operations in a manner similar to [Reg94]. The simulation in (b) is *non-uniform*, insofar as it gives separate UPSILON terms T_n, one for each length n of binary input strings, that solves the problem on inputs of that length.

5.5 Self

Because UPSILON is Turing-compete and has a composition function, it is an *acceptable programming system* (APS) as defined by Rogers [Rog58]. A fundamental consequence 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 subtypes to the model.

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

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

- if $\tau = \overline{\sigma}$
  - $\text{RHS} = \text{RHS}'(\tau) = \sigma$
    The result of plugging a port is the type that is expected by the port.
  - $\text{LHS} = \text{LHS}'(\tau) = \{\sigma\}$
    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 $\tau = \overline{\overline{\sigma}}$
  - $\text{RHS} = \text{RHS}'(\tau) = \overline{\sigma}$
    The result of plugging a locked port is a port.
  - $\text{LHS} = \text{LHS}'(\tau) = \{*\}$
    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

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

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

$$\frac{A : \langle \tau_1, \ldots, \tau_n \rangle \quad B : \langle \sigma_1, \ldots, \sigma_m \rangle}{A@B : \langle \tau_1, \ldots, \tau_n, \sigma_1, \ldots, \sigma_m \rangle}$$

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

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

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

11. TyPlug

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

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

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

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

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

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

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

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

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

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

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

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 $O \overset{\rightharpoonup}{\sim}_1 O'$ (as defined in Appendix A.2). If $O : \tau$ then $O' : \tau$.*

Proof: This follows by a trivial induction over the derivation of $O \overset{\rightharpoonup}{\sim}_1 O'$. ■

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: $\{\tau_1, \ldots, \tau_n\} \rightarrow \{\sigma_1, \ldots, \sigma_n\}$. 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:

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

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?

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

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 5.2, 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 RAM simulator 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

$$S \curlyvee i\ x \rightsquigarrow ((\ominus \curlyvee \ominus \ominus) \curlyvee \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus) \curlyvee [[i]]\ [[x]]\ \ominus$$

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

The only other possible way to get $s_{1,1}$ is to have $S \curlyvee i\ x = T$ 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, $T = S \curlyvee ix = i \curlyvee P(x)Q$ With three guarantees:

1. For all $B, P(x) \curlyvee B \rightsquigarrow x$
2. For all $B, Q \curlyvee B \rightsquigarrow B$
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 $i = \ominus \curlyvee \pi_1(\ominus)$ Then we end up with: $P(x) \curlyvee \pi_1(Q)$ Now, consider guarantee 1, this means that we end up with x. When we plug y into this we get $x \curlyvee y$ not $x \curlyvee \pi_1(y)$ 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, subtypes, 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

[AC96] 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.

[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(4):127–206, 1994.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming A Unified Foundation*. Birkhäuser, Boston, 1997.

[CGL95] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[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.

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

[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.

[Reg94] K. Regan. A new parallel vector model, with exact characterizations of NC^k. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computation*, volume 778 of *Lecture Notes in Computer Science*, pages 289–300. Springer-Verlag, 1994.

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

[Rog58] H. Rogers. Gödel-numberings of the 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.

[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.

A Proof of Confluence, Theorem A.1

To make the proof easier to present and follow, we define two "conservative extensions" of the calculus in which arguments in a list *RHS* are plugged one-at-a-time rather than all at once. First, we define $A^{B_1,\ldots,B_n}$ to be the object created when one plugs the elements of the list $B_1, \ldots, B_n$ into A. 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. Next, 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 Appendix 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 (Appendix 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 $A \overset{\rightharpoonup}{}_1 B$, 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) $O \overset{\rightharpoonup}{}_1 O$

(b)
$$\left.\begin{array}{l} O_1 \overset{\rightharpoonup}{}_1 O_1' \\ O_2 \overset{\rightharpoonup}{}_1 O_2' \\ \quad\vdots \\ O_n \overset{\rightharpoonup}{}_1 O_n' \end{array}\right\} \Longrightarrow [O_1, O_2, \ldots, O_n] \overset{\rightharpoonup}{}_1 [O_1', O_2', \ldots, O_n']$$

(c) $\left.\begin{array}{l} O_1 \overset{\rightharpoonup}{}_1 O_1' \\ O_2 \overset{\rightharpoonup}{}_1 O_2' \end{array}\right\} \Longrightarrow O_1@O_2 \overset{\rightharpoonup}{}_1 O_1'@O_2'$

(d) $[O_1, O_2, \ldots, O_n]@[O_{n+1}, O_{n+2}, \ldots, O_m] \overset{\rightharpoonup}{}_1 [O_1, O_2, \ldots, O_n, O_{n+1}, O_{n+2}, \ldots, O_m]$

(e) $O \overset{\rightharpoonup}{}_1 O' \Longrightarrow \pi_i(O) \overset{\rightharpoonup}{}_1 \pi_i(O')$

(f) $\pi_i([\underbrace{\ldots}_{i-1}, O_i, \ldots]) \overset{\rightharpoonup}{}_1 O_i$, provided $O_i \neq \odot$.

(g) $\left.\begin{array}{c} O_0 \overset{\rightharpoonup}{}_1 O_0' \\ O_1 \overset{\rightharpoonup}{}_1 O_1' \\ O_2 \overset{\rightharpoonup}{}_1 O_2' \\ \vdots \\ O_n \overset{\rightharpoonup}{}_1 O_n' \end{array}\right\} \Longrightarrow O_0 \; \Upsilon^i \; O_1 O_2 \ldots O_n \overset{\rightharpoonup}{}_1 O_0' \; \Upsilon^i \; O_1' O_2' \ldots O_n'$

(h) Provided L is reduced, $\lceil \text{valance}(L) \div n \rceil = i$, $L, O_1, \ldots, O_n \neq \odot$, and $n \geq 1$, $L \; \Upsilon \; O_1 O_2 \ldots O_n \overset{\rightharpoonup}{}_1 (L^{\overline{O_1}}) \; \Upsilon \; O_2 \ldots O_n \; (O_1 O_2 \ldots O_n)^{i-1}$.

(i) Provided $\text{valance}(L) = 0$, $L \; \Upsilon \; O_1, \ldots, O_n \overset{\rightharpoonup}{}_1 \mathbf{deline}(L)$, where $\mathbf{deline}(L)$ 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

$$\begin{aligned} L \; \Upsilon \; O_1 O_2 \ldots O_n &\overset{\rightharpoonup}{}_1 (L^{\overline{O_1}}) \; \Upsilon \; O_2 \ldots O_n, \\ L \; \Upsilon^i \; O_1 &\overset{\rightharpoonup}{}_1 (L^{\overline{O_1}}) \; \Upsilon^{i-1} \; O_1, \quad \text{and} \\ L \; \Upsilon \; O_1 &\overset{\rightharpoonup}{}_1 (L^{\overline{O_1}}) \; \Upsilon^0 \; O_1 \end{aligned}$$

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

Now we establish that the resulting calculus has the important properties of confluence and uniqueness of reduced forms.

Theorem A.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 A.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.

First we need to prove some lemmas.

Lemma A.3. $O \overset{\rightharpoonup}{}_1 O' \rightarrow A^O \overset{\rightharpoonup}{}_1 A^{O'}$

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 $O \overset{\rightharpoonup}{}_1 A'$, we know that $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

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: $A_1^O \overset{\rightharpoonup}{}_1 A_1^{O'}$.
And therefore, $[Q_1, \ldots Q_n, A_1^O, O_1, \ldots O_m] \overset{\rightharpoonup}{}_1 [Q_1, \ldots Q_n, A_1^{O'}, O_1, \ldots O_m]$. Hence, $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

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: $A_1^O \overset{\rightharpoonup}{}_1 A_1^{O'}$, Thus we know that $(A_1^O @ O_1) \overset{\rightharpoonup}{}_1 (A_1^{O'} @ O_1)$. Therefore, $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

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: $A_1^O \overset{\rightharpoonup}{}_1 A_1^{O'}$, Thus we know that $(Q_1 @ A_1^O) \overset{\rightharpoonup}{}_1 (Q_1 @ A_1^{O'})$. So, $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

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: $A_1^O \overset{\rightharpoonup}{}_1 A_1^{O'}$, Thus we know that $(\pi_i(A_1^O)) \overset{\rightharpoonup}{}_1 (\pi_i(A_1^{O'}))$. Hence, $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

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 $(O\ \Upsilon^i\ O_1 \ldots O_n) \overset{\rightharpoonup}{}_1 (O'\ \Upsilon^i\ O_1 \ldots O_n)$. Thus, $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

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: $A_1^O \overset{\rightharpoonup}{}_1 A_1^{O'}$, Thus we know that
$(O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1^O O_1 \ldots O_n) \overset{\rightharpoonup}{}_1 (O_0\ \Upsilon^i\ Q_1 \ldots Q_m A_1^{O'} O_1 \ldots O_n)$. Therefore, $A^O \overset{\rightharpoonup}{}_1 A^{O'}$.

■

Lemma A.4. $\rightharpoonup_1$ *satisfies the diamond property. i.e.*

$$\forall O.O \rightharpoonup_1 O_1, \text{ and } O \rightharpoonup_1 O_2 \rightarrow \exists O_3.O_1 \rightharpoonup_1 O_3 \text{ and } O_2 \rightharpoonup_1 O_3$$

Proof: The proof is by induction on $O \rightharpoonup_1 O_1$. For this proof, π_i and R_i represent arbitrary objects.

Case 1 $O \rightharpoonup_1 O_1 = O \rightharpoonup_1 O$ So, if $O \rightharpoonup_1 O_2$ We can make $O_3 = O_2$.

Case 2 $O \rightharpoonup_1 O_1 = [P_1, \ldots P_n] \rightharpoonup_1 [P'_1, \ldots, P'_n]$ where $\forall i.P_i \rightharpoonup_1 P'_i$. 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: $O \rightharpoonup_1 O_2 = [P_1, \ldots P_n] \rightharpoonup_1 [P''_1, \ldots, P''_n]$ with
$\forall i.P_i \rightharpoonup_1 P'_i$ and $P_i \rightharpoonup_1 P''_i$. By the Induction Hypothesis, we know that
$\exists P'''_i.P'_i \rightharpoonup_1 P'''_i$ and $P''_i \rightharpoonup_1 P'''_i$. Thus
$[P'_1, \ldots P'_n] \rightharpoonup_1 [P'''_1, \ldots, P'''_n]$ and $[P''_1, \ldots P''_n] \rightharpoonup_1 [P'''_1, \ldots, P'''_n]$. So, $O_3 = [P'''_1, \ldots, P'''_n]$.

Case 3 $O \rightharpoonup_1 O_1 = P_1@P_2 \rightharpoonup_1 P'_1@P'_2$ With $P_1 \rightharpoonup_1 P'_1, P_2 \rightharpoonup_1 P'_2$

Case 3.1 $O \rightharpoonup_1 O_2 = P_1@P_2 \rightharpoonup_1 P''_1@P''_2$, with $P_1 \rightharpoonup_1 P''_1$, and $P_2 \rightharpoonup_1 P''_2$. So, just like in Case 2, by the Induction Hypothesis there must be P'''_1 and P'''_2 such that $P'_1 \rightharpoonup_1 P'''_1, P''_1 \rightharpoonup_1 P'''_1$ and $P'_2 \rightharpoonup_1 P'''_2, P''_2 \rightharpoonup_1 P'''_2$. 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 $O \rightharpoonup_1 O_2$ is actually
$[P_1, P_2, \ldots, P_n]@[R_1, R_2, \ldots, R_m] \rightharpoonup_1 [P_1, P_2 \ldots P_n, R_1, R_2, \ldots, R_m]$. In this case O_1 must have been
$[P'_1, P'_2, \ldots, P'_n]@[R'_1, R'_2, \ldots, R'_m]$, Since, as in Case 2, that's the only way we can reduce base objects.

So, we have:
$[P_1, P_2, \ldots, P_n]@[R_1, R_2, \ldots, R_m] \rightharpoonup_1 [P'_1, P'_2, \ldots, P'_n]@[R'_1, R'_2, \ldots, R'_m]$ and
$[P_1, P_2, \ldots, P_n]@[R_1, R_2, \ldots, R_m] \rightharpoonup_1 [P_1, P_2 \ldots P_n, R_1, R_2, \ldots, R_m]$.
We can make $O_3 = [P'_1, P'_2 \ldots P'_n, R'_1, R'_2, \ldots, R'_m]$.

Case 4 $O \rightharpoonup_1 O_1 = \pi_i(P) \rightharpoonup_1 \pi_i(P')$ with $P \rightharpoonup_1 P'$.

Case 4.1 $O \rightharpoonup_1 O_2 = \pi_i(P) \rightharpoonup_1 \pi_i(P'')$ with $P \rightharpoonup_1 P''$. Again by the inductive hypothesis, there is some P''' such that $P' \rightharpoonup_1 P'''$ and $P'' \rightharpoonup_1 P'''$. So, $O_3 = \pi_i(P''')$.

Case 4.2 The other alternative is that O is actually a base object, and
$O \rightharpoonup_1 O_2 = \pi_i([P_1 \ldots, P_i, \ldots, P_n]) \rightharpoonup_1 P_i$. In this case, we can make $O_3 = P'_i$.

Case 5 $O \rightharpoonup_1 O_1 = P_0 \ \Upsilon^i \ P_1 \ldots P_n \rightharpoonup_1 P'_0 \ \Upsilon^i \ P'_1 \ldots P'_n$ with $\forall j.P_j \rightharpoonup_1 P'_j$.

Case 5.1 $O \rightharpoonup_1 O_2 = P_0 \ \Upsilon^i \ P_1 \ldots P_n \rightharpoonup_1 P''_0 \ \Upsilon^i \ P''_1 \ldots P''_n$ with $\forall j.P_j \rightharpoonup_1 P''_j$. Again, by the Induction Hypothesis, we know that $\forall j.\exists P'''_j.P'_j \rightharpoonup_1 P'''_j$ and $P''_j \rightharpoonup_1 P'''_j$. So, we can make $O_3 = P'''_0 \ \Upsilon^i \ P'''_1 \ldots P'''_n$.

Case 5.2 The other option is that $O \rightharpoonup_1 O_2 = P_0 \ \Upsilon^i \ P_1 \ldots P_n \rightharpoonup_1 P_0 \ \overbrace{\ }^{i}$ $\overbrace{\ }^{P_1 \ldots P_n}$, 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 $O_3 = P_0 \ \overbrace{\ }^{i}$ $\overbrace{P'_1 P'_1 \ldots P'_1}$ $\overbrace{P'_2 P'_2 \ldots P'_2}^{i}$ ..... $\overbrace{P'_n P'_n \ldots P'_n}^{i}$. From the previous lemma and the Induction Hypothesis, we know that both O_1 and O_2 reduce to this.

■

Let $\rightsquigarrow$ be the transitive closure of $\overset{\sim}{\to}_1$ and $\approx$ the equality relation that it induces.

Proof: Plugging in the above Lemma A.4 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 to complete the proof of the theorem. ■

B Recursion Theorem

Our internal proof of the recursion theorem follows that of [Kle38] and also [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 5.2, 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 $e = s \Upsilon vv$.

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

Now, we plug e with some argument x:

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

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:

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

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

$$\begin{aligned}
e &= v\ \Upsilon\ vx = (((\ominus\ \Upsilon\ \ominus\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus\ \ominus)\ \Upsilon\ [[j]]\ [[i]]\ \ominus\ \ominus)\ \overset{\downarrow}{\Upsilon}\ v\ x \\
&\rightsquigarrow (((\ominus\ \Upsilon\ \ominus\ \ominus\ \ominus)\ \Upsilon\ \pi_1(\pi_1(\ominus))\ \pi_1(\pi_1(\ominus))\ \ominus\ \ominus)\ \overset{\downarrow}{\Upsilon}\ [[j]]\ [[i]]\ v\ x) \\
&\rightsquigarrow ((\ominus\ \Upsilon\ \ominus\ \ominus\ \ominus)\ \Upsilon\ \overset{\downarrow}{\pi_1}\ (\overset{\downarrow}{\pi_1}\ ([[j]]))\ \overset{\downarrow}{\pi_1}\ (\overset{\downarrow}{\pi_1}\ ([[i]]))\ v\ x) \\
&\rightsquigarrow ((\ominus\ \Upsilon\ \ominus\ \ominus\ \ominus)\ \overset{\downarrow}{Plug}\ j\ i\ v\ x) \\
&\rightsquigarrow (j\ \Upsilon\ i\ v\ x)
\end{aligned}$$

Now, we need to expand j:

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

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.

■