INFORMATIQUE THÉORIQUE ET APPLICATIONS

ROY L. CROLE

Encoding fix in object calculi

Informatique théorique et applications, tome 34, n° 1 (2000), p. 15-38

<http://www.numdam.org/item?id=ITA_2000__34_1_15_0>

Theoretical Informatics and Applications
Theoret. Informatics Appl. **34** (2000) 15–38

ENCODING FIX IN OBJECT CALCULI

ROY L. CROLE[1]

Abstract. We show that the *FIX* type theory introduced by Crole and Pitts [3] can be encoded in variants of Abadi and Cardelli's object calculi. More precisely, we show that the *FIX* type theory presented with judgements of both equality and operational reduction can be translated into object calculi, and the translation proved sound. The translations we give can be seen as using object calculi as a metalanguge within which *FIX* can be represented; an analogy can be drawn with Martin Löf's Theory of Arities and Expressions. As well as providing a description of certain interesting recursive objects in terms of rather simpler expressions found in the *FIX* type theory, the translations will be of interest to those involved with the automation of operational semantics.

AMS Subject Classification. 68Q55.

1. MOTIVATION

The results and ideas presented in this paper concern type theories equipped with either equational or operational semantics. We shall assume that readers are familiar with type theory, and more specifically with type theories possessing judgements of both equality, and operational reduction. A good general reference for (dependent) type theory is the book [10]. For operational reduction, see for example [2, 4, 6, 8, 11–13].

In fact the results presented here are founded on the *FIX* type theory introduced by Crole and Pitts in [3], and the object type theories (calculi) introduced by Abadi and Cardelli in [1]. If the reader is not familiar with the details of these type theories, we would ask that they consult the key references which we have just given. However, we will give some informal explanations of the theories in *loc cit* which should help readers through the key ideas of the current paper. We shall also repeat *some* of the technical details in *loc cit* when it is sensible to do

[1] Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester LE1 7RH, U.K.; e-mail: Roy.Crole@mcs.le.ac.uk

so. Before moving on to explain the results of this paper, let us review informally the *FIX* and object type theories.

The *FIX* type theory was (in part) designed to be used as a metalanguage for programming language semantics. It contains natural numbers, products and sums, higher order functions, computation types [9] to interpret lazy and eager evaluation strategies, and finally a *fixpoint* type which provides (in the presence of computation types) a framework in which recursive programming constructs can be interpreted. Previous work of the author, and others, used the *FIX* type theory (sometimes embedded in a logic) to give semantics to programming languages.

Abadi and Cardelli introduced object type theories (calculi) to provide a foundation for the theory of object oriented languages at the level of an operational semantics. In their book [1] one finds discussions of very many systems, but broadly each system provides a theory of objects with either equational semantics, single step operational semantics, or natural semantics. Abadi and Cardelli have shown that object calculi are highly expressive, leading to the principle that "everything is an object". Various people (see for example [5]) have provided evidence for this, in the form of examples, and simple translations or encodings. One of the key properties of object calculi is that their operational semantics are "highly recursive", and as a simple consequence of this Abadi and Cardelli have also shown that such calculi model recursion operators.

These observations lead to the questions

1. *Can a rich type theory such as FIX be encoded in object calculi?*
2. *Can this be done in a way which unifies those examples given by Abadi and Cardelli (and others)*?
3. *What uses can we find for the encoding?*

The aim of this paper is to provide answers to these questions. The short answer to both (1) and (2) is "yes". We answer (1) by giving a translation of *FIX* into object calculi, and proving the translation sound. In fact we shall consider both an equational version of *FIX*, and an operational version, and specify sound translations into suitable equational and operational object type theories. We shall show that the definitions of the encoding given here can be explained in a canonical way. It is this canonical translation which provides us with at least one answer to (3). In recent work of Ambler and the author, the theorem prover Isabelle has been used to represent programming languages and to verify properties of the languages. We have used the simply typed lambda calculus as a metalanguage in which to represent object level programming languages. In fact the translation we describe in this paper can also be viewed as using *OBJ* as a metalanguage. This gives us a new system for representing object level programming languages. The fact that our translation is canonical, as explained in detail in Section 3, provides a useful guiding principle when formulating automated machine translations in theorem provers such as Isabelle.

The paper proceeds as follows. In Section 2 we give a short but technically detailed review of *FIX*, and an object type theory which we call *OBJ*. Both of these systems are equational theories. In Section 3 we give a translation of *FIX*

M	$::=$	$\langle M, M\rangle$	*pairs*
	$\mid$	$\mathsf{Split}(M, v.v.M)$	*splitting*
	$\mid$	$\mathsf{Inl}(M)$	*left copair insertion*
	$\mid$	$\mathsf{Inr}(M)$	*right copair insertion*
	$\mid$	$\mathsf{Case}(M, v.M, v.M)$	*cases*
	$\mid$	$\lambda(v.M)$	*functions*
	$\mid$	$M\ M$	*function application*
	$\mid$	$\mathsf{Val}(M)$	*value computation types*
	$\mid$	$\mathsf{Let}(M, v.M)$	*sequencing computation types*
	$\mid$	ω	*fixpoint index*
	$\mid$	$s(M)$	*fixpoint successor*
	$\mid$	$(v.M)^M$	*fixpoint iterator*

FIGURE 1. Term Grammar for *FIX*.

into *OBJ*. We explain how the translation has been derived in a canonical way. We give a proof that the translation is sound, that is, we show that we have given an encoding of *FIX* in *OBJ*. The section ends with a discussion of *FIX* recursion and its translation into *OBJ*. In Section 4 we introduce the systems FIX_{op} and OBJ_{op}. These are analogues of *FIX* and *OBJ* which have an operational semantics. This is essential for the kind of work mentioned above, namely the representation of object level programming languages. In Section 5 we give a translation of FIX_{op} into OBJ_{op}, and prove it sound. We then conclude the paper.

2. THE SYSTEMS *FIX* AND *OBJ*

We outline the type theories used in Section 3. For general background on type theory, see [10]. The equational theory *FIX* was introduced by Pitts and the author; please see [3] for further details. This theory is simply typed, with **types** given by the grammar

$$\sigma ::= K_{FIX} \mid \mathit{fix} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \Rightarrow \sigma \mid T\sigma$$

where K_{FIX} is an arbitrary ground type (such as natural numbers). The **terms** M are given in Figure 1 where v ranges over a fixed set of **variables**. As outlined in Section 1 the system provides a type theory suitable for programming language semantics: ground types, products (pair types), coproducts (sum types), higher order functions, computation types, and the fixpoint type. Without going into technical detail here, the fixpoint type allows recursion to be interpreted. In fact any term whose type is of the form $T\sigma \Rightarrow T\sigma$ (informally a term which transforms computations into computations) must have a fixpoint which can be represented directly within the type theory. Note that while we have presented a formal term

grammar, we shall often use a suggestive notation to indicate the types of terms, such as P for a pair (term of type $\sigma \times \sigma'$), C for a copair (term of type $\sigma + \sigma'$), F for a function (term of type $\sigma \Rightarrow \sigma'$) and so on.

The theory *OBJ* contains rules for deriving judgements of the following forms

$\Gamma \vdash M : \sigma$	*type assignment*
$\Gamma \vdash M = M' : \sigma$	*equality*

where Γ is an environment assigning types to terms, often denoted by $[x_1 \colon \sigma_1, \ldots, x_n \colon \sigma_n]$. Thus, formally, Γ is a (finite) partial function from the set of variables to the set of types. The rules for deriving type assignments and equalities are all standard, and may be found in [10] together with [3]. Note that a consequence of deriving an equality judgement is that the two terms involved are well typed; for example if $\Gamma \vdash M = M' : \sigma$ is a valid judgement, then $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M' : \sigma$ are valid type assignments.

Although we ask that the reader be familiar with the *FIX* theory, we remind the reader of the ideas behind the simple domain theoretic model of *FIX*, which will help to give meaning to those parts of *FIX* which are associated with computation types [9] and the fixpoint type. We model types as (bottomless) ωcpos, and type assignments as continuous functions. Let σ and σ' be modelled by the ωcpos D and D'. Also, let $M \colon \sigma$ be modelled by $d \in D$. $T\sigma$ is modelled by the lifted domain $D_\perp$. The term $\mathsf{Val}(M) \in D_\perp$ is modelled as the inclusion $\iota \colon D \to D_\perp$ of d in $D_\perp$. The term $\mathsf{Let}(E, x.E')$ is modelled as the Kleisli composition of the functions modelling E and $x \colon \sigma \vdash E' : T\sigma'$. *fix* is modelled as the vertical natural numbers with infinity, $\mathbb{N}^\infty$. Let $x \colon T\sigma \vdash F : \sigma$ be modelled by the function $f \colon D_\perp \to D$; let $\vdash E : T\mathit{fix}$ modelled by $e \in \mathbb{N}^\infty_\perp$; and let $\vdash N : \mathit{fix}$ be modelled by $n \in \mathbb{N}^\infty$. Then ω is modelled by $\infty \in \mathbb{N}^\infty$; the term $(x.F)^N$ can be thought of as the Nth recursive unfolding of F and is modelled by $(\iota \circ f)^n(\perp)$; $s(E)$ is modelled by $e+1$ if $e \neq \infty$ and ∞ otherwise; and finally $(x.F)^\omega$ is modelled by $\bigvee_{i \in \mathbb{N}} (\iota \circ f)^i(\perp)$, which is the least fixpoint of $\iota \circ f$.

Let us now move to a brief description of *OBJ*. This type theory is essentially Abadi and Cardelli's $Ob \cup \Delta_\forall \cup \Delta_\mu$. This system is their basic calculus of objects, enriched with $\forall$-types (see [1], p. 170) and recursive foldings and unfoldings (see [1], p. 114). Thus *OBJ* has **types**

σ	$::=$	V	*variables*
	$\mid$	K_{OBJ}	*ground type*
	$\mid$	$[l_i \colon \sigma_i \ ^{i \in I}]$	*object type*
	$\mid$	$\forall(V.\sigma)$	*universal type*
	$\mid$	$\mu(V.\sigma)$	*recursive type*

where V ranges over a set of type variables, the symbol l is an element of a fixed set of labels and I ranges over finite subsets of $\mathbb{N}$. Such labels give names to the fields and methods (sometimes referred to as attributes and operations) of objects.

The **terms** are given by the grammar

M	$::=$	v	*variables*
	$\mid$	$[l_i = \varsigma(v_i)M_i{}^{i\in I}]$	*objects*
	$\mid$	$M \,.\, l$	*method invocation*
	$\mid$	$M \,.\, l \Leftarrow \varsigma(v)M$	*method update*
	$\mid$	$\Lambda(V.M)$	*type variable abstraction*
	$\mid$	M_σ	*type variable application*
	$\mid$	$\mathsf{Fld}(M)$	*folding for recursive types*
	$\mid$	$\mathsf{UFld}(M)$	*unfolding for recursive types.*

The **results** are given by the grammar

$$\begin{array}{rcl} R & ::= & [l_i = \varsigma(v_i)M_i{}^{i\in I}] \\ & \mid & \Lambda(V.M) \\ & \mid & \mathsf{Fld}(M). \end{array}$$

One can think of the results as final values returned by operational reductions; they are used below.

The theory *OBJ* contains rules for deriving judgements of the forms

$\Theta \vdash M : \sigma$	*type assignment*
$\Theta \vdash M = M' : \sigma$	*equality*

whose intended meaning is the same as the analogous constructs from *FIX*. Here, $\Theta \stackrel{\text{def}}{=} \Delta \mid \Gamma$ is an environment assigning types to variables in Γ, and listing free type variables in Δ. Thus a typical Δ is $[X_1, \ldots, X_n]$ with the X_i all type variables. As usual, in any typing judgement

$$[X_1, \ldots, X_n] \mid [x_1 : \sigma_1, \ldots, x_n : \sigma_n] \vdash M : \sigma$$

the free type variables occuring in the σ_i, σ and M must appear in $[X_1, \ldots, X_n]$, and the free term variables occuring in M must appear in $[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$. We omit Δ or Γ if either is empty. The equational theory for *OBJ* consists of the usual equations of Abadi and Cardelli's core object calculus, the usual equations for $\forall$-types, and the usual fold/unfold equations which specify that fold and unfold constructors are mutual inverses.

3. An encoding for equality judgements

3.1. Explaining a translation of *FIX* into *OBJ*

Now we address the first new topic of this paper, namely giving an encoding of *FIX* into *OBJ*. Thus we need to specify a translation of *FIX* into *OBJ* and prove it sound. The encoding is canonical in the sense that it is based on an adaptation of the following idea. It is well understood that the simply typed lambda calculus with constants, $\mathcal{L}$, can be used as a metalanguage into which other type theories

	$\mathcal{T} \stackrel{\text{def}}{=} FIX$
Constr	$\langle M, M' \rangle$
Destr	$\mathsf{Split}(P, x.x'.N)$
Eq	$\mathsf{Split}(\langle M, M' \rangle, x.x'.N) = N\{x, x' \leftarrow M, M'\}$
	$\mathcal{L} \stackrel{\text{def}}{=} E ::= x \mid k \mid E\,E \mid v.E$
Constr	$k_{\mathsf{Pair}}\,[\![M]\!]\,[\![M']\!]$
Destr	$k_{\mathsf{Split}}\,[\![P]\!]\,(x.x'.[\![N]\!])$
Eq	$k_{\mathsf{Split}}\,(k_{\mathsf{Pair}}\,[\![M]\!]\,[\![M']\!])\,(x.x'.[\![N]\!]) = [\![N]\!]\{x, x' \leftarrow [\![M]\!], [\![M']\!]\}$

FIGURE 2. Explaining arities and expressions with product types.

$\mathcal{T}$ can be translated [10] in a canonical fashion. This paper proceeds in a similar manner, but we shall explain how to use OBJ as a metalanguage into which $\mathcal{T}$ can be translated in a canonical way. In order to better understand these ideas, we shall first recall how to use $\mathcal{L}$ as a metalanguage for general type theories $\mathcal{T}$, by giving a translation function $[\![-]\!]\colon \mathcal{T} \to \mathcal{L}$. This idea, sometimes known as Martin Löf's theory of arities and expressions, is explained fully in [10].

1. Variable binders in $\mathcal{T}$ are modelled by abstraction in $\mathcal{L}$. More precisely, if $\mathcal{T}$ has a binding construction of the form $\vec{v}.T$, then it will be canonically represented in $\mathcal{L}$ by $\lambda\vec{v}.[\![T]\!]$. Note that we often drop the λ and simply write $\vec{v}.[\![T]\!]$ overloading the notation.
2. Constructor and destructor terms in $\mathcal{T}$ are represented in $\mathcal{L}$ by constants. In either case, if a term in $\mathcal{T}$ is built out of n sub-terms $\vec{v}.T_i$, where the $\vec{v}$ may be empty if there are no bound variables in the subterm, then it will be represented in $\mathcal{L}$ by $k(\vec{v}.[\![T_1]\!], \ldots, \vec{v}.[\![T_n]\!])$ where k is a constant in $\mathcal{L}$ and the notation is the usual shorthand[1] denoting nested function applications in $\mathcal{L}$.
3. If an equation $T = T'$ holds in $\mathcal{T}$, then we assert that $[\![T]\!] = [\![T']\!]$ holds (in the theory representing $\mathcal{T}$) in $\mathcal{L}$. Thus equalities in $\mathcal{T}$ are represented by (non-logical) equalities in $\mathcal{L}$.

We give an example of this methodology when $\mathcal{T}$ is FIX. The FIX product constructor terms take the form $\langle M, M' \rangle$; the destructor terms take the form $\mathsf{Split}(P, x.y.N)$. They satisfy the equation

$$\mathsf{Split}(\langle M, M' \rangle, x.y.N) = N\{x, y \leftarrow M, M'\} \qquad (\dagger)$$

where we have omitted environments and types which would clutter our explanation of the central ideas. The pair constructor is built from two terms, M and M', and the split destructor is built from a pair P and a term $x.y.N$ in which any occurrences of the variables x and y in N are bound in the destructor term. We require two $\mathcal{L}$ constants to represent the constructor and destructor terms respectively;

[1]For example, $k(T_1, T_2, T_3)$ denotes $((kT_1)T_2)T_3$.

we call these k_{Pair} and k_{Split}. The translations of the constructor and destructor terms are shown in Figure 2. Finally note that we assert that the translation of the equands appearing in † are equal in $\mathcal{L}$ as an axiom.

We shall now explain how we can replace $\mathcal{L}$ by OBJ. We now regard OBJ as a metalanguage, and describe a translation $[\![-]\!]\colon \mathcal{T} \to OBJ$. First, we need a little more notation. We write $\uparrow \stackrel{\text{def}}{=} \varsigma(z)z\,.\,l$. Then for any object of the form $O \stackrel{\text{def}}{=} [\ldots, l = \varsigma(z)z\,.\,l, \ldots]$ we have $O\,.\,l = O\,.\,l = \ldots$. Thus we can regard O as a "looping object". In fact the closed term $\uparrow$ can be instantiated at any type, and we shall use it as a generic object which inhabits all types. One might liken it to the recursive term $\mathsf{rec}x.x$ for "undefinedness". Often, we will wish to create an object in which the names of labels are known, but the attribute body is not. We can use fields of the form $l = \uparrow$ to fulfil this role. We shall now explain the general formulation of $[\![-]\!]$ on terms of non-*fix* or non-computation type.

1. Variable binders in $\mathcal{T}$ are all modelled by the OBJ self binder. More precisely, if $\mathcal{T}$ has a binding construction of the form $\vec{v}.T$, then it will be canonically represented in OBJ by $[\![v_1 \ldots v_K.T]\!] \stackrel{\text{def}}{=} \varsigma(z)[\![T]\!]\{v_k \leftarrow z\,.\,l_k \;\; {}^{1\le k\le K}\}$. The choice of actual name of the labels l_k will be explained in 3 below.
2. A constructor term in $\mathcal{T}$ will be represented by an object some of whose fields correspond to its sub-terms. The constructor term has a corresponding destructor term. This destructor term will itself contain sub-terms which specify the possible final results when the destructor consumes a constructor. Fields in the object will be used to indicate these final result terms. The remaining field is used to indicate the actual final result.
   In more detail, if a constructor term in $\mathcal{T}$ is built out of n sub-terms $\vec{v}.T_i$, and its corresponding destructor returns one of m possible final result terms, then it will be represented in OBJ by an object of the form

   $$[l_1 = [\![\vec{v}.T_1]\!], \ldots, l_n = [\![\vec{v}.T_n]\!], \mathsf{cnxt}_1 = \uparrow, \ldots, \mathsf{cnxt}_m = \uparrow, \mathsf{res} = \varsigma(z)R]$$

   where the term R is used to select a final result from the cnxt fields. For example, in the case of translations of *FIX* terms with non-*fix* or non-computation type, if cnxt_j labels the final result, R will be $z\,.\,\mathsf{cnxt}_j$.
3. A typical destructor term will be built out of a single subterm T, which represents a constructor term to be consumed, together with a sequence of subterms $v_1 \ldots v_K.N_i$ which specify the possible final results from computing the destructor. If the term has a non-*fix* or non-computation type then these data will typically be translated into an object of the form

   $$[\![T]\!] \circ [\![T_1]\!] \circ [\![T_2]\!] \circ \ldots \circ [\![T_m]\!] \stackrel{\text{def}}{=} ([\![T]\!]\,.\,\mathsf{cnxt}_1 \Leftarrow \varsigma(z)[\![T_1]\!]\{v_k \leftarrow z\,.\,l_k \;\; {}^{1\le k\le K}\}$$
   $$\vdots$$
   $$.\,\mathsf{cnxt}_m \Leftarrow \varsigma(z)[\![T_m]\!]\{v_k \leftarrow z\,.\,l_k \;\; {}^{1\le k\le K}\}).\mathsf{res}.$$

4. If an equation $T = T'$ holds in $\mathcal{T}$, then the choice of fields and methods suggested above will ensure that $[\![T]\!] = [\![T']\!]$ holds in OBJ.

Let us apply these ideas when $\mathcal{T}$ is *FIX* to develop a canonical translation of *FIX* products into *OBJ*.

1. Variable bindings will appear below.
2. Consider the constructor term $\langle M, M' \rangle$. This term has two sub-terms M and M' and thus its translation has two fields whose labels we call fst and snd. The corresponding destructor term is $\mathsf{Split}(\langle M, M' \rangle, x.y.N)$ which can return just one possible final result term based on N. The translation will have just one field of the form $\mathsf{cnxt} = \uparrow$. Finally, adding in a method to return the result value held in the cnxt field, we see that we obtain the object

$$[\mathsf{fst} = [\![M]\!], \mathsf{snd} = [\![M']\!], \mathsf{cnxt} = \uparrow, \mathsf{res} = \varsigma(z) z \,.\, \mathsf{cnxt}].$$

   This is the object $pair_X^{[\![M]\!];[\![M']\!]}$ given in Figure 4 and is used for the translation of pairs.
3. Consider the destructor term $\mathsf{Split}(P, x.y.N)$. This has a main (sub-)term P which represents a pair constructor and just one other subterm $x.y.N$. Thus this term is translated to an object term of the form

$$[\![P]\!] \circ [\![N]\!] = ([\![P]\!] \,.\, \mathsf{cnxt} \Leftarrow \varsigma(z)[\![N]\!]\{x, y \leftarrow z \,.\, \mathsf{fst}, z \,.\, \mathsf{snd}\}) \,.\, \mathsf{res}.$$

   The object $[\![P]\!]$ contains a field cnxt which should hold the computation of the final result of (the translation of) the split term. This final result is obtained from $[\![N]\!]$, whose bound variables must be updated to hold the first and second components of (the translation of) the pair P. These are selected using fst and snd. Finally, the invocation of res returns the final result.
4. By calculating in *OBJ*, using just logical equalities we can see that

$$[\![\langle M, M' \rangle]\!] \circ [\![N]\!] = [\![N]\!]\{x, y \leftarrow [\![M]\!], [\![M']\!]\}.$$

   In fact, as we shall see later, $[\![N\{x, y \leftarrow M, M'\}]\!] = [\![N]\!]\{x, y \leftarrow [\![M]\!], [\![M']\!]\}$ and thus the encoding is sound—recall † on page 20. We summarise these ideas in Figure 3. Note that the constructor translation involves a type abstraction. This is an extra piece of book-keeping to ensure the correctness of types. The type variable X will be instantiated with the translation of the type of the split term when the constructor is actually consumed by a destructor—hence the term $[\![P]\!]_{[\![\tau]\!]}$ in Figure 3.

Let us now explain our translation of the type *fix* and its terms. It will be useful to refer to a categorical model of *FIX* as given by a FIX-category. See [3] for details. The type $T\sigma$ indicates a computation of type σ. This is translated to the object type $[l^T \colon [\![\sigma]\!]]$ which records that any computation with result type $[\![\sigma]\!]$ can be extracted by invoking the l^T method. With this, we can define $[\![fix]\!] \stackrel{\text{def}}{=} \mu(X.[l^T \colon X])$. The translation is the expected one, using standard fold/unfold type recursion. The constructor s is modelled as the structure map (isomorphism) $Tfix \to fix$ of the initial T-algebra in a *FIX*-category. Given

	OBJ
Constr	$\xi \stackrel{\text{def}}{=} \Lambda(X.[\mathsf{fst}=[\![M]\!], \mathsf{snd}=[\![M']\!], \mathsf{cnxt}=\uparrow, \mathsf{res}=\varsigma(z\colon PROD_X^{[\![\sigma]\!];[\![\sigma']\!]})z\,.\,\mathsf{cnxt}])$
Destr	$([\![P]\!]_{[\![\tau]\!]}\,.\,\mathsf{cnxt} \Leftarrow \varsigma(z)[\![N]\!]\{x, x' \leftarrow z\,.\,\mathsf{fst}, z\,.\,\mathsf{snd}\})\,.\,\mathsf{res}$
Eq	$(\xi_{[\![\tau]\!]}\,.\,\mathsf{cnxt} \Leftarrow \varsigma(z)[\![N]\!]\{x, x' \leftarrow z\,.\,\mathsf{fst}, z\,.\,\mathsf{snd}\})\,.\,\mathsf{res} = [\![N]\!]\{x, x' \leftarrow [\![M]\!], [\![M']\!]\}$

FIGURE 3. Explaining the encoding principle at product types.

this, we can expect to translate the term $s(E)$ as $\mathsf{Fld}([\![E]\!])$. The term ω represents a global element of *fix* in any categorical model. In particular, it is used to "enumerate" any "infinite" recursive unfolding of a term modelled by a morphism $TA \to TA$ (recall the *FIX*-category example in Sect. 2). It satisfies an equation of the form $\omega = s(\mathsf{Val}(\omega))$. By considering this equational property, one soon decides that $\mathsf{Fld}([l^T = \varsigma(z\colon [l^T\colon [\![fix]\!]])\mathsf{Fld}(z)])$ is a suitable candidate for the translation. Finally, we consider the term $(x.F)^N$ and also the equality

$$\Gamma \vdash (x.F)^{s(E)} = F\{x \leftarrow \mathsf{Let}(E, n.\mathsf{Val}((x.F)^n))\} : \sigma \qquad (*).$$

The object that $(x.F)^N$ is translated to should have attributes to represent $x.F$ and N. Let us name the label for $[\![N]\!]$ by pow. Note that $x.F$ is a computation, hence its attribute will be labelled by l^T. The translation should also have a "final result" attribute, which will perform recursive unfoldings, say with label rec. This leads to a translation type $IT^\sigma \stackrel{\text{def}}{=} [l^T\colon [l^T\colon \sigma], \mathsf{pow}\colon [\![fix]\!], \mathsf{rec}\colon \sigma]$. Let us think about the attribute bodies in the translation of $(x.F)^N$. First, pow simply records the translation of the term N. Next, the method rec is used to return recursive unfoldings. Thus we can set its body to be $\mathsf{rec} = \varsigma(z'\colon IT^\sigma)[\![F]\!]\{x \leftarrow z'\,.\,l^T\}$. This object body selects the result of the computation $x.[\![F]\!]$ by invoking l^T, and passes this (recursively) into $x.[\![F]\!]$ by binding the result to x. Finally, we think about the remaining attribute l^T. This will hold a computation (say c) which matches the right hand side of $*$, as this will implicitly ensure the soundness of $*$ in the translation. We obtain c by asking how the right hand side of $*$ is evaluated. The term $s([\![E]\!])$ (given by $z\,.\,\mathsf{pow}$) is unfolded to $[\![E]\!]$ and then this computation is evaluated; this corresponds to $\mathsf{UFld}(z.\mathsf{pow}).l^T$. This value is the new value taken by pow, and hence we should have an update of pow. The final result given by c should be extracted by invoking rec. Thus we have $c = (z\,.\,\mathsf{pow} \Leftarrow \mathsf{UFld}(z\,.\,\mathsf{pow})\,.\,l^T)\,.\,\mathsf{rec}$. Putting this all together gives rise to the translation of $(x.F)^N$ in Figure 5.

3.2. THE TRANSLATION

Let us formally define a translation function $[\![-]\!]\colon FIX \to OBJ$ on the types and terms of *FIX*. This translation function was developed using the basic ideas

- $IT^{\sigma} \stackrel{\text{def}}{=} [l^T\colon [l^T\colon \sigma], \mathsf{pow}\colon fix_{OBJ}, \mathsf{rec}\colon \sigma]$ where $fix_{OBJ} \stackrel{\text{def}}{=} \mu(X.[l^T\colon X])$
- $PROD_X^{\sigma;\sigma'} \stackrel{\text{def}}{=} [\mathsf{fst}\colon \sigma, \mathsf{snd}\colon \sigma', \mathsf{cnxt}\colon X, \mathsf{res}\colon X]$
- $COPR_X^{\sigma;\sigma'} \stackrel{\text{def}}{=} [\mathsf{inl}\colon \sigma, \mathsf{inr}\colon \sigma', \mathsf{cnxtl}\colon X, \mathsf{cnxtr}\colon X, \mathsf{res}\colon X]$
- $it_v^{N;F} \stackrel{\text{def}}{=} [l^T = \varsigma(z\colon IT^{\sigma})\xi, \mathsf{pow} = N, \mathsf{rec} = \varsigma(z'\colon IT^{\sigma})F\{v \leftarrow z'.l^T\}]$
  where $\xi \stackrel{\text{def}}{=} [l^T = (z.\mathsf{pow} \Leftarrow \mathsf{UFld}(z.\mathsf{pow}).l^T).\mathsf{rec}]$
- $pair_X^{M;M'} \stackrel{\text{def}}{=} [\mathsf{fst} = M, \mathsf{snd} = M', \mathsf{cnxt} = \uparrow, \mathsf{res} = \varsigma(z\colon PROD_X^{\sigma;\sigma'})z.\mathsf{cnxt}]$
- $inl_X^M \stackrel{\text{def}}{=} [\mathsf{inl} = M, \mathsf{inr} = \uparrow, \mathsf{cnxtl} = \uparrow, \mathsf{cnxtr} = \uparrow, \mathsf{res} = \varsigma(z\colon COPR_X^{\sigma;\sigma'})z.\mathsf{cnxtl}]$
- $inr_X^M \stackrel{\text{def}}{=} [\mathsf{inl} = \uparrow, \mathsf{inr} = M, \mathsf{cnxtl} = \uparrow, \mathsf{cnxtr} = \uparrow, \mathsf{res} = \varsigma(z\colon COPR_X^{\sigma;\sigma'})z.\mathsf{cnxtr}]$
- $fun_v^M \stackrel{\text{def}}{=} [\mathsf{arg} = \uparrow, \mathsf{val} = \varsigma(z)M\{v \leftarrow z.\mathsf{arg}\}]$
- $M \bullet N \stackrel{\text{def}}{=} (M.\mathsf{arg} \Leftarrow N).\mathsf{val}$

FIGURE 4. Auxiliary definitions for translating *FIX* into *OBJ*.

which we have just presented. The function $[\![-]\!]$ maps type assignments to type assignments:

$$\begin{array}{ccc} \Gamma \vdash M : \sigma & \xmapsto{[\![-]\!]} & [\,] \mid [\![\Gamma]\!] \vdash [\![M]\!] : [\![\sigma]\!] \\ \uparrow & & \uparrow \\ \textit{judgement in FIX} & & \textit{judgement in OBJ} \end{array}$$

Note that a consequence of the encoding is that $[\![\sigma]\!]$ and $[\![M]\!]$ have no free type variables, and thus have an environment of the form $[\,] \mid [\![\Gamma]\!]$.

The definition of $\sigma \mapsto [\![\sigma]\!]$ appears in Figure 5. Note that given the definition of $[\![\sigma]\!]$ for any type σ, we define the translation on contexts by

$$[\![[x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n]]\!] \stackrel{\text{def}}{=} [x_1\colon [\![\sigma_1]\!] \ldots, x_n\colon [\![\sigma_n]\!]].$$

Given these definitions, the specification of the assignment $M \mapsto [\![M]\!]$ also appears in Figure 5; note that given $\Gamma \vdash M : \sigma$, the definition of $[\![M]\!]$ depends on $[\![\Gamma]\!]$ and $[\![\sigma]\!]$, and that there are auxiliary definitions of types and terms in Figure 4. The "$\circ$" notation is defined in Lemma 3.3. We have already met this notation informally in Section 3.1. Finally, $M\{v \leftarrow N\}$ denotes substitution for free variables (whose formal definition we omit).

- $[\![K_{FIX}]\!] \stackrel{\text{def}}{=} K_{OBJ}$;
- $[\![\mathit{fix}]\!] \stackrel{\text{def}}{=} \mathit{fix}_{OBJ}$
- $[\![\sigma \times \sigma']\!] \stackrel{\text{def}}{=} \forall(X.PROD_X^{[\![\sigma]\!];[\![\sigma']\!]})$
- $[\![\sigma + \sigma']\!] \stackrel{\text{def}}{=} \forall(X.COPR_X^{[\![\sigma]\!];[\![\sigma']\!]})$
- $[\![\sigma \Rightarrow \tau]\!] \stackrel{\text{def}}{=} [\mathsf{arg}\colon [\![\sigma]\!], \mathsf{val}\colon [\![\tau]\!]]$
- $[\![T\sigma]\!] \stackrel{\text{def}}{=} [l^T\colon [\![\sigma]\!]]$

- $[\![\Gamma \vdash v : \sigma]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash v : [\![\sigma]\!]$;
- $[\![\Gamma \vdash \langle M, M'\rangle : \sigma \times \sigma']\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \Lambda(X.pair_X^{[\![M]\!];[\![M']\!]}) : [\![\sigma \times \sigma']\!]$
- $[\![\Gamma \vdash \mathsf{Split}(P, x.y.N) : \tau]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash [\![P]\!]_{[\![\tau]\!]} \circ [\![N]\!] : [\![\tau]\!]$
- $[\![\Gamma \vdash \mathsf{Inl}(M) : \sigma + \sigma']\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \Lambda(X.inl_X^{[\![M]\!]}) : [\![\sigma + \sigma']\!]$
- $[\![\Gamma \vdash \mathsf{Inr}(M) : \sigma + \sigma']\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \Lambda(X.inr_X^{[\![M]\!]}) : [\![\sigma + \sigma']\!]$
- $[\![\Gamma \vdash \mathsf{Case}(C, x.N, y.N') : \tau]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash [\![C]\!]_{[\![\tau]\!]} \circ [\![N]\!] \circ [\![N']\!] : [\![\tau]\!]$
- $[\![\Gamma \vdash \lambda(v.M) : \sigma \Rightarrow \sigma']\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \mathit{fun}_v^{[\![M]\!]} : [\![\sigma \Rightarrow \sigma']\!]$
- $[\![\Gamma \vdash F\,A : \sigma']\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash [\![F]\!] \bullet [\![A]\!] : [\![\sigma']\!]$
- $[\![\Gamma \vdash \mathsf{Val}(M) : T\sigma]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash [l^T = [\![M]\!]] : [\![T\sigma]\!]$
- $[\![\Gamma \vdash \mathsf{Let}(E, v.F) : T\sigma']\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash [\![F]\!]\{v \leftarrow [\![E]\!] \,.\, l^T\} : [\![T\sigma']\!]$
- $[\![\Gamma \vdash \omega : \mathit{fix}]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \mathsf{Fld}([l^T = \varsigma(z\colon [l^T : \mathit{fix}_{OBJ}])\mathsf{Fld}(z)]) : [\![\mathit{fix}]\!]$
- $[\![\Gamma \vdash s(E) : \mathit{fix}]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \mathsf{Fld}([\![E]\!]) : [\![\mathit{fix}]\!]$
- $[\![\Gamma \vdash (v.F)^N : \sigma]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \mathit{it}_v^{[\![N]\!];[\![F]\!]} \,.\, \mathsf{rec} : [\![\sigma]\!]$

FIGURE 5. Translating *FIX* into *OBJ*.

3.3. PROVING SOUNDNESS

We wish to prove that the translation specified in Figure 5 soundly models *FIX* equalities in *OBJ*. The soundness of the translation depends on three key lemmas. Lemma 3.1 is a version of a standard result that translations respect substitutions. Lemmas 3.2 and 3.3 provide the key properties which ensure that the translations we have given for terms of non-*fix* or non-computation type will soundly model equalities.

Lemma 3.1. *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma, v\colon \sigma \vdash N : \tau$ *in FIX, then for* $[\![-]\!]\colon FIX \to OBJ$ *we have*

$$[\![N]\!]\{v \leftarrow [\![M]\!]\} \equiv [\![N\{v \leftarrow M\}]\!]$$

where $\equiv$ *denotes* ***syntactic identity.***

Proof. This is proved easily by induction on the structure of N. We omit the details. □

Lemma 3.2. *If* $\Theta, x\colon \sigma \vdash M : \tau$ *and* $\Theta \vdash N : \sigma$ *in OBJ, then*

$$\Theta \vdash fun_x^M \bullet N = M\{x \leftarrow N\} : \tau.$$

Proof. A simple calculation in *OBJ*. See [1] for details. □

Lemma 3.3. *All judgements in this lemma are in OBJ. Suppose that* $\Theta \vdash M_i : \sigma_i$ *where* $1 \leq i \leq n$, *and* $\Theta, x_k\colon \sigma_k{}^{k \in S_j} \vdash N_j : \tau$ *where* $1 \leq j \leq m$ *and* $S_j \subset \{1, \ldots, n\}$ *and* $L_j \stackrel{\text{def}}{=} \{ l_k \mid k \in S_j \}$. *Let*

$$GEN_X^{\sigma_1;\ldots;\sigma_n} \stackrel{\text{def}}{=} [l_i\colon \sigma_i{}^{i \in n}, \mathsf{cnxt}_{L_j} : X{}^{j \in m}, \mathsf{res}\colon X]$$

and also

$$gen_{X;j}^{M_1;\ldots;M_n} \stackrel{\text{def}}{=} [l_i = M_i{}^{i \in n}, \mathsf{cnxt}_{L_j} = \uparrow{}^{j \in m}, \mathsf{res} = \varsigma(z\colon GEN_X^{\sigma_1;\ldots;\sigma_n})z \,.\, \mathsf{cnxt}_{L_j}].$$

Note that

$$X, \Theta \vdash gen_{X;j}^{M_1;\ldots;M_n} : GEN_X^{\sigma_1;\ldots;\sigma_n}.$$

We shall also write

$$O \circ N_1 \circ N_2 \circ \ldots \circ N_m \stackrel{\text{def}}{=} (O \quad \begin{array}{l} \,.\, \mathsf{cnxt}_{L_1} \Leftarrow \varsigma(z) N_1\{x_k \leftarrow z \,.\, l_k{}^{k \in S_1}\} \\ \qquad\qquad \vdots \\ \,.\, \mathsf{cnxt}_{L_m} \Leftarrow \varsigma(z) N_m\{x_k \leftarrow z \,.\, l_k{}^{k \in S_m}\}) \,.\, \mathsf{res}. \end{array}$$

Then it follows that

$$\Theta \vdash (\Lambda(X.gen_{X;j}^{M_1;\ldots;M_n}))_\tau \circ N_1 \circ N_2 \circ \ldots \circ N_m = N_j\{x_k \leftarrow M_k{}^{k \in S_j}\} : \tau.$$

Proof. The proof is a simple calculation of equalities in *OBJ*. For clarity, we omit the typing environments.

$$\begin{aligned} &(\Lambda(X.gen_{X;j}^{M_1;\ldots;M_n}))_\tau \circ N_1 \circ N_2 \circ \ldots \circ N_m \\ &\quad = \quad (gen_{\tau;j}^{M_1;\ldots;M_n} \ldots \,.\, \mathsf{cnxt}_{L_j} \Leftarrow \varsigma(z) N_j\{x_k \leftarrow z \,.\, l_k{}^{k \in S_j}\} \ldots) \,.\, \mathsf{res} \\ &\quad = \quad \xi \,.\, \mathsf{res} \\ &\quad = \quad \xi \,.\, \mathsf{cnxt}_{L_j} \\ &\quad = \quad N_j\{x_k \leftarrow \xi \,.\, l_k{}^{k \in S_j}\} \\ &\quad = \quad N_j\{x_k \leftarrow M_k{}^{k \in S_j}\} \end{aligned}$$

where

$$\xi \stackrel{\text{def}}{=} [l_i = M_i{}^{i \in n}, \mathsf{cnxt}_{L_j} = \varsigma(z) N_j\{x_k \leftarrow z \,.\, l_k{}^{k \in S_j}\}{}^{j \in m}, \mathsf{res} = \varsigma(z\colon GEN_X^{\sigma_1;\ldots;\sigma_n})z \,.\, \mathsf{cnxt}_{L_j}].$$

□

We can now state the first soundness result. Theorem 3.4 states that *OBJ* encodes *FIX*.

Theorem 3.4. *We have given an encoding of FIX into OBJ. More precisely, if* $\Gamma \vdash M : \sigma$ *in FIX, then* $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\sigma]\!]$ *in OBJ, and if* $\Gamma \vdash M = M' : \sigma$ *in FIX, then* $[\![\Gamma]\!] \vdash [\![M]\!] = [\![M']\!] : [\![\sigma]\!]$ *in OBJ.*

Proof. The proof proceeds by induction on derivations. We illustrate the proof on case and iteration terms. The typing rule for case terms is

$$\frac{\Gamma \vdash C : \sigma + \sigma' \quad \Gamma, x\colon \sigma \vdash N : \tau \quad \Gamma, x'\colon \sigma' \vdash N' : \tau}{\Gamma \vdash \mathsf{Case}(C, x.N, x'.N') : \tau}.$$

By induction, we have

$$[\![\Gamma]\!] \vdash [\![C]\!] : \forall(X.COPR_X^{[\![\sigma]\!];[\![\sigma']\!]})$$

$$[\![\Gamma]\!], x\colon [\![\sigma]\!] \vdash [\![N]\!] : [\![\tau]\!]$$

$$[\![\Gamma]\!], x'\colon [\![\sigma']\!] \vdash [\![N']\!] : [\![\tau]\!]$$

and hence we can derive

$$[\![\Gamma]\!] \vdash [\![C]\!]_{[\![\tau]\!]} : COPR_{[\![\tau]\!]}^{[\![\sigma]\!];[\![\sigma']\!]}$$

$$[\![\Gamma]\!], z\colon COPR_{[\![\tau]\!]}^{[\![\sigma]\!];[\![\sigma']\!]}, x\colon [\![\sigma]\!] \vdash [\![N]\!] : [\![\tau]\!] \qquad \text{(by weakening)}$$

$$[\![\Gamma]\!], z\colon COPR_{[\![\tau]\!]}^{[\![\sigma]\!];[\![\sigma']\!]}, x'\colon [\![\sigma']\!] \vdash [\![N']\!] : [\![\tau]\!] \qquad \text{(by weakening)}$$

$$[\![\Gamma]\!], z\colon COPR_{[\![\tau]\!]}^{[\![\sigma]\!];[\![\sigma']\!]} \vdash z\,.\,\mathsf{inl} : [\![\sigma]\!].$$

From this we have

$$\Gamma, z\colon COPR_{[\![\tau]\!]}^{[\![\sigma]\!];[\![\sigma']\!]} \vdash [\![N]\!]\{x \leftarrow z\,.\,\mathsf{inl}\} : [\![\tau]\!]$$

with a similar result for N'. Thus

$$[\![\Gamma]\!] \vdash [\![C]\!]_{[\![\tau]\!]}\,.\,\mathsf{cnxtl} \Leftarrow \varsigma(z)[\![N]\!]\{x \leftarrow z\,.\,\mathsf{inl}\}\,.\,\mathsf{cnxtr} \Leftarrow \varsigma(z)[\![N']\!]\{x' \leftarrow z\,.\,\mathsf{inr}\} : COPR_{[\![\tau]\!]}^{[\![\sigma]\!];[\![\sigma']\!]}.$$

Finally, note that if we invoke the above object at the label res, we get precisely

$$[\![\Gamma]\!] \vdash [\![C]\!]_{[\![\tau]\!]} \circ [\![N]\!] \circ [\![N']\!] : [\![\tau]\!]$$

as required.

The typing rule for iteration terms is

$$\frac{\Gamma, x\colon T\sigma \vdash F : \sigma \qquad \Gamma \vdash N : fix}{\Gamma \vdash (x.F)^N : \sigma}.$$

By induction we have $[\![\Gamma]\!], x\colon [l^T : [\![\sigma]\!]] \vdash [\![F]\!] : [\![\sigma]\!]$ and $[\![\Gamma]\!] \vdash [\![N]\!] : fix_{OBJ}$. Then we have $[\![\Gamma]\!] \vdash it_x^{[\![N]\!];[\![F]\!]} : IT^{[\![\sigma]\!]}$ by typing rules in OBJ, and thus $[\![\Gamma]\!] \vdash it_x^{[\![N]\!];[\![F]\!]}.\mathsf{rec} : [\![\sigma]\!]$.

The soundness of equalities not involving the fixpoint type is a direct consequence of Lemma 3.2, which deals with functions, and Lemma 3.3 which deals with the other terms, together with Lemma 3.1.

For example, soundness of

$$\Gamma \vdash \mathsf{Case}(\mathsf{Inl}(M), x.N, x'.N') = N\{x \leftarrow M\} : \tau$$

requires

$$\begin{aligned} [\![\Gamma]\!] \vdash [\![\mathsf{Case}(\mathsf{Inl}(M), x.N, x'.N')]\!] &\stackrel{\text{def}}{=} \Lambda(X.inl_X^{[\![M]\!]})_{[\![\tau]\!]} \circ [\![N]\!] \circ [\![N']\!] \\ &=_{3.3} [\![N]\!]\{x \leftarrow [\![M]\!]\} : [\![\tau]\!] \\ &\equiv_{3.1} [\![N\{x \leftarrow M\}]\!]. \end{aligned}$$

As for the fixpoint terms, we prove soundness for

$$\Gamma \vdash (x.F)^{s(E)} = F\{x \leftarrow \mathsf{Let}(E, n.\mathsf{Val}((x.F)^n))\} : \sigma.$$

We have, using Lemma 3.1 and setting $O \stackrel{\text{def}}{=} it_x^{\mathsf{Fld}([\![E]\!]);[\![F]\!]}$,

$$\begin{aligned} [\![\Gamma]\!] \vdash [\![(x.F)^{s(E)}]\!] &\stackrel{\text{def}}{=} O.\mathsf{rec} \\ &= [\![F]\!]\{x \leftarrow O.l^T\} \\ &= [\![F]\!]\{x \leftarrow [l^T = (O.\mathsf{pow} \Leftarrow \mathsf{UFld}(O.\mathsf{pow}).l^T).\mathsf{rec}]\} \\ &= [\![F]\!]\{x \leftarrow [l^T = (O.\mathsf{pow} \Leftarrow \mathsf{UFld}(\mathsf{Fld}([\![E]\!])).l^T).\mathsf{rec}]\} \\ &= [\![F]\!]\{x \leftarrow [l^T = (O.\mathsf{pow} \Leftarrow [\![E]\!].l^T).\mathsf{rec}]\} \\ &= [\![F]\!]\{x \leftarrow [l^T = it_x^{[\![E]\!].l^T;[\![F]\!]}.\mathsf{rec}]\} \\ &\equiv [\![F]\!]\{x \leftarrow [l^T = it_x^{n;[\![F]\!]}.\mathsf{rec}]\{n \leftarrow [\![E]\!].l^T\}\} \\ &\equiv [\![F\{x \leftarrow \mathsf{Let}(E, n.\mathsf{Val}((x.F)^n))\}]\!] : [\![\sigma]\!]. \end{aligned}$$

□

3.4. Encoding *FIX* fixpoints

One interesting payoff of the encoding we have given is to look at the connections between *FIX* terms and their encodings. In fact some of the more obscure objects which Abadi and Cardelli discuss when looking at recursion are rendered in a rather trivial form by examining possible pre-images under the translation. For

example, their so-called "self-returning" (untyped) object, $[l = \varsigma(z)z]$, in the typed setting, would be

$$\mathsf{Fld}([l^T = \varsigma(z)\mathsf{Fld}(z)]).$$

But a pre-image under $[\![-]\!]$ is exactly ω. This demonstrates beautifully (in the author's opinion!) how the primitive operation of "recursive" object reduction is captured in the single *FIX* constant ω. Further, the object type system allows the structure map s of *FIX* to be naturally interpreted as a recursive folding, using the standard fold/unfold terms.

It is also interesting to look at the encodings in *OBJ* of the fixpoint operators which are *definable* in *FIX*. Suppose that $\vdash F : T\sigma \Rightarrow T\sigma$ in *FIX*. Then if we define (see [3])

$$\mathsf{Fix}(F) \stackrel{\text{def}}{=} (e.\mathsf{Let}(e, x.F\,x))^{\omega}$$

where $e\colon TT\sigma$, we can show that $\mathsf{Fix}(F) = F\,\mathsf{Fix}(F)$ in *FIX*. The encoding of $\mathsf{Fix}(F)$ in *OBJ* is easily seen to be $it_e^{\hat{\omega}\,;[\![F]\!]\bullet(e.l^T)}.\mathsf{rec}$ where we have defined $\hat{\omega} \stackrel{\text{def}}{=} [\![\omega]\!]$. Noting that $[\![T\sigma \Rightarrow T\sigma]\!] = [\mathsf{arg}\colon [l^T\colon \sigma], \mathsf{val}\colon [l^T\colon \sigma]]$ we can then show the following lemma.

Lemma 3.5. *Suppose that* $\Gamma \vdash F : [\mathsf{arg}\colon [l^T\colon \sigma], \mathsf{val}\colon [l^T\colon \sigma]]$ *in OBJ. Then we have the type assignment* $\Gamma \vdash it_e^{\hat{\omega}\,;F\bullet(e.l^T)}.\mathsf{rec} : [l^T\colon \sigma]$ *and moreover*

$$\Gamma \vdash it_e^{\hat{\omega}\,;F\bullet(e.l^T)}.\mathsf{rec} = F \bullet (it_e^{\hat{\omega}\,;F\bullet(e.l^T)}.\mathsf{rec}) : [l^T\colon \sigma].$$

Proof. Of course, the lemma is an immediate consequence of Theorem 3.4. However, it is instructive to perform the computations directly.

First note that

$$\begin{aligned}
\mathsf{UFld}(\hat{\omega}).l^T &\stackrel{\text{def}}{=} \mathsf{UFld}(\mathsf{Fld}([l^T = \varsigma(z)\mathsf{Fld}(z)])).l^T \\
&= [l^T = \varsigma(z)\mathsf{Fld}(z)].l^T \\
&= \mathsf{Fld}([l^T = \varsigma(z)\mathsf{Fld}(z)]) \\
&\equiv \hat{\omega}.
\end{aligned}$$

Hence

$$
\begin{aligned}
& it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{rec} \\
&\quad = \; F \bullet (e \,.\, l^T)\{e \leftarrow it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, l^T\} \\
&\quad = \; F \bullet ([l^T = (it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{pow} \Leftarrow \mathsf{UFld}(it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{pow}) \,.\, l^T) \,.\, \mathsf{rec}] \,.\, l^T) \\
&\quad = \; F \bullet ((it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{pow} \Leftarrow \mathsf{UFld}(it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{pow}) \,.\, l^T) \,.\, \mathsf{rec}) \\
&\quad = \; F \bullet ((it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{pow} \Leftarrow \mathsf{UFld}(\hat{\omega}) \,.\, l^T) \,.\, \mathsf{rec}) \\
&\quad = \; F \bullet ((it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{pow} \Leftarrow \hat{\omega}) \,.\, \mathsf{rec}) \\
&\quad = \; F \bullet (it_e^{\hat{\omega};\, F\bullet(e.l^T)} \,.\, \mathsf{rec}).
\end{aligned}
$$

□

4. The systems FIX_{op} and OBJ_{op}

Here we replay the ideas of Section 2 but turn our attention to operational reductions rather than simple equalities. If we wish to consider using objects to represent the operational behaviour of computational systems, this will be essential. Let us first consider an operational semantics for *FIX*. While this is easily formulated, the details are not published explicitly. Thus we give the definition of FIX_{op}. This system is essentially the *FIX* syntax equipped with a one-step operational semantics, developed by orienting the *FIX* equalities.

The theory FIX_{op} contains rules for deriving judgements of the following forms

$\Gamma \vdash M : \sigma$	*type assignment*
$\Gamma \vdash M \rightsquigarrow M' : \sigma$	*lazy one-step reductions.*

Note that a consequence of deriving an operational judgement is that the two terms involved are well typed; for example if $\Gamma \vdash M \rightsquigarrow M' : \sigma$ is a valid judgement, then $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M' : \sigma$ are valid type assignments. We give the rules for deriving the lazy one-step reductions, which are not present in [3], in Figure 6. The system is lazy: we regard constructor terms as fully evaluated (and hence such terms do not feature as reductands in the rules). The choice has been made because the rules are slightly simpler than the analogous eager system. None-the-less, the lazy system is perfectly fine for explaining the core ideas of our encodings. (We do not describe a notion of value, as we do not consider evaluation semantics [7] in this paper.)

Our last task in this section is to describe the system OBJ_{op} which is new. This system extends *OBJ*. It provides an operational semantics for the *OBJ* syntax. It also has some extra term forming operations which are used to correctly model

$$\frac{\Gamma \vdash P \rightsquigarrow P' : (\sigma, \sigma')\ \Gamma, x\colon \sigma, x'\colon \sigma' \vdash N : \tau}{\Gamma \vdash \mathsf{Split}(P, x.x'.N) \rightsquigarrow \mathsf{Split}(P', x.x'.N) : \tau} \qquad \frac{\Gamma \vdash M : \sigma\ \Gamma \vdash M' : \sigma'\ \Gamma, x\colon \sigma, x'\colon \sigma' \vdash N : \tau}{\Gamma \vdash \mathsf{Split}(\langle M, M'\rangle, x.x'.N) \rightsquigarrow N\{x, x' \leftarrow M, M'\} : \tau}$$

$$\frac{\Gamma \vdash C \rightsquigarrow C' : \sigma + \sigma'\ \Gamma, x\colon \sigma \vdash N : \tau \quad \Gamma, x'\colon \sigma' \vdash N' : \tau}{\Gamma \vdash \mathsf{Case}(C, x.N, x'.N') \rightsquigarrow \mathsf{Case}(C', x.N, x'.N') : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x\colon \sigma \vdash N : \tau \quad \Gamma, x'\colon \sigma' \vdash N' : \tau}{\Gamma \vdash \mathsf{Case}(\mathsf{Inl}(M), x.N, x'.N') \rightsquigarrow N\{x \leftarrow M\} : \tau} \qquad \frac{\Gamma \vdash M' : \sigma' \quad \Gamma, x\colon \sigma \vdash N : \tau \quad \Gamma, x'\colon \sigma' \vdash N' : \tau}{\Gamma \vdash \mathsf{Case}(\mathsf{Inr}(M'), x.N, x'.N') \rightsquigarrow N'\{x' \leftarrow M'\} : \tau}$$

$$\frac{\Gamma \vdash F \rightsquigarrow F' : \sigma \Rightarrow \sigma'\ \Gamma \vdash A : \sigma}{\Gamma \vdash F\,A \rightsquigarrow F'\,A : \sigma'} \qquad \frac{\Gamma, x\colon \sigma \vdash M : \sigma' \quad \Gamma \vdash A : \sigma}{\Gamma \vdash \lambda(x.M)\,A \rightsquigarrow M\{x \leftarrow A\} : \sigma'}$$

$$\frac{\Gamma \vdash E \rightsquigarrow E' : T\sigma \quad \Gamma, x\colon \sigma \vdash F : T\sigma'}{\Gamma \vdash \mathsf{Let}(E, x.F) \rightsquigarrow \mathsf{Let}(E', x.F) : T\sigma'} \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma, x\colon \sigma \vdash F : T\sigma'}{\Gamma \vdash \mathsf{Let}(\mathsf{Val}(M), x.F) \rightsquigarrow F\{x \leftarrow N\} : T\sigma'}$$

$$\frac{\Gamma \vdash N \rightsquigarrow N' : \mathit{fix}}{\Gamma \vdash (x.F)^{N} \rightsquigarrow (x.F)^{N'} : \sigma} \qquad \frac{\Gamma, x\colon T\sigma \vdash F : \sigma \quad \Gamma \vdash E : T\mathit{fix}}{\Gamma \vdash (x.F)^{s(E)} \rightsquigarrow F\{x \leftarrow \mathsf{Let}(E, n.\mathsf{Val}((x.F)^{n}))\} : \sigma}$$

$$\frac{\Gamma \vdash \omega : \mathit{fix}}{\Gamma \vdash \omega \rightsquigarrow s(\mathsf{Val}(\omega)) : \mathit{fix}}$$

FIGURE 6. Rules for generating $\Gamma \vdash M \rightsquigarrow M' : \sigma$ in *FIX*.

the lazy FIX_{op} reductions. The **types** are the same as those of *OBJ*. The **terms** are given by the same grammar as for *OBJ*, but extended by

M	::=	...	*terms from OBJ*
	|	$\mathsf{Eval}_l(M)$	*eager method and update bodies*
	|	$\mathsf{Letval}\, v = M \,\mathsf{in}\, M$	*eager local declaration*
	|	$\mathsf{Letinv}\, v = M \,\mathsf{in}\, M$	*eager local method invocation.*

The rules for deriving type assignment in OBJ_{op} extend those of *OBJ*. We also have the rule

$$\frac{\Theta \vdash M : [l_i\colon \sigma_i{}^{i \in I}]}{\Theta \vdash \mathsf{Eval}_l(M) : [l_i\colon \sigma_i{}^{i \in I}]}$$

and the rules for the typing of $\mathsf{Letval}\, v = M \,\mathsf{in}\, N$ and $\mathsf{Letinv}\, v = M \,\mathsf{in}\, N$ are exactly the same as those usually associated with let-evaluation terms.

Abadi and Cardelli present one step operational reductions for our *OBJ* syntax. For example, the rules for folding and unfolding appear in [1] (p. 120). The rules for deriving reductions in OBJ_{op} extend these rules. The additional rules are those given in Figure 7. The basic idea is that $\mathsf{Eval}_l(-)$ enforces an eager evaluation of object method bodies and object update bodies. The term $\mathsf{Letval}\, v = M \,\mathsf{in}\, N$ is an

$$\frac{\Theta \vdash M \rightsquigarrow M' : \sigma \quad \Theta, v\colon \sigma \vdash N : \sigma'}{\Theta \vdash \mathsf{Letval}\, v = M \,\mathsf{in}\, N \rightsquigarrow \mathsf{Letval}\, v = M' \,\mathsf{in}\, N : \sigma'}$$

$$\frac{\Theta \vdash M : \sigma \quad \Theta, v\colon \sigma \vdash N : \sigma'}{\Theta \vdash \mathsf{Letval}\, v = R \,\mathsf{in}\, N \rightsquigarrow N\{R \leftarrow v\} : \sigma'}$$

$$\frac{\Theta \vdash M \rightsquigarrow M' : \sigma \quad \Theta, v\colon \sigma \vdash N : \sigma'}{\Theta \vdash \mathsf{Letinv}\, v = M \,\mathsf{in}\, N \rightsquigarrow \mathsf{Letinv}\, v = M' \,\mathsf{in}\, N : \sigma'} \quad \nexists O.\ \nexists l.\ \ M \equiv O \cdot l$$

$$\frac{\Theta \vdash O : [l_i\colon \sigma_i{}^{i\in I}] \quad \Theta, v\colon \sigma_j \vdash N : \sigma'}{\Theta \vdash \mathsf{Letinv}\, v = O \cdot l_j \,\mathsf{in}\, N \rightsquigarrow N\{v \leftarrow B_j\{z_j \leftarrow O\}\} : \sigma'} \quad O \equiv [l_i = \varsigma(z_i) B_i{}^{i\in I}]$$

$$\frac{\Theta \vdash M \rightsquigarrow M' : [l_i\colon \sigma_i{}^{i\in I}]}{\Theta \vdash \mathsf{Eval}_l(M) \rightsquigarrow \mathsf{Eval}_l(M') : [l_i\colon \sigma_i{}^{i\in I}]}$$

$$\frac{\Theta, z_j\colon [l_i\colon \sigma_i{}^{i\in I}] \vdash B_j \rightsquigarrow B_j' : \sigma_j}{\Theta \vdash \mathsf{Eval}_{l_j}([l_i = \varsigma(z_i) B_i{}^{i\in I}]) \rightsquigarrow [l_i = \varsigma(z_i) B_i{}^{i\neq j}, l_j = \varsigma(z_j) B_j'] : [l_i\colon \sigma_i{}^{i\in I}]}$$

$$\frac{\Theta, z_j\colon [l_i\colon \sigma_i{}^{i\in I}] \vdash B_j \rightsquigarrow B_j' : \sigma_j \quad \Theta \vdash M : [l_i\colon \sigma_i]^{i\in I}}{\Theta \vdash \mathsf{Eval}_{l_j}(M \cdot l_j \Leftarrow \varsigma(z_j) B_j) \rightsquigarrow M \cdot l_j \Leftarrow \varsigma(z_j) B_j' : [l_i\colon \sigma_i]^{i\in I}}$$

Note that we shall also write $\mathsf{Letval}\, v_i = M_i{}^{i\in\{1,\ldots,r\}} \,\mathsf{in}\, N$ as an abbreviation for

$$\mathsf{Letval}\, v_1 = M_1 \,\mathsf{in}\, (\mathsf{Letval}\, v_2 = M_2 \,\mathsf{in}\, (\ldots (\mathsf{Letval}\, v_r = M_r \,\mathsf{in}\, N)\ldots))$$

provided that none of the variables v_i appear in any of the M_j for any $i, j \in \{1, \ldots, r\}$

The notation $\mathsf{Letinv}\, v_i = M_i{}^{i\in\{1,\ldots,r\}} \,\mathsf{in}\, N$ will be used analogously.

FIGURE 7. Additional rules for generating $\Theta \vdash M \rightsquigarrow M' : \sigma$ in OBJ_{op}.

eager local declaration, evaluating M as far as possible before binding the result to v in N. The term $\mathsf{Letinv}\, v = M \,\mathsf{in}\, N$ is an eager local declaration, evaluating M to an object of the form $[l_i = \varsigma(v_i) B_i{}^{i\in I}] \cdot l_j$ before binding the invocation of l_j to v in N. We shall write also define $\mathsf{Eval}_l^1(M) \stackrel{\text{def}}{=} \mathsf{Eval}_l(M)$ and for $n \geq 1$ we set $\mathsf{Eval}_l^{n+1}(M) \stackrel{\text{def}}{=} \mathsf{Eval}_l(\mathsf{Eval}_l^n(M))$. This notation will be used in Theorem 5.4.

- $it_v^{N;F} \stackrel{\text{def}}{=} [l^T = \varsigma(z\colon IT^\sigma)\xi, \mathsf{pow} = N, \mathsf{rec} = \varsigma(z')\mathsf{Letval}\, v = z' \,.\, l^T \,\mathsf{in}\, F]$
  where $\xi \stackrel{\text{def}}{=} \mathsf{Eval}^3_{l^T}([l^T = (\mathsf{Eval}^2_{\mathsf{pow}}(z \,.\, \mathsf{pow} \Leftarrow \mathsf{UFld}(z \,.\, \mathsf{pow}) \,.\, l^T)) \,.\, \mathsf{rec}])$
- $fun_v^M \stackrel{\text{def}}{=} [\mathsf{arg} = \uparrow, \mathsf{val} = \varsigma(z)\mathsf{Letinv}\, v = z \,.\, \mathsf{arg}\,\mathsf{in}\, M]$

FIGURE 8. Auxiliary definitions for translating FIX_{op} into OBJ_{op}.

- $[\![\Gamma \vdash \omega : fix]\!] \stackrel{\text{def}}{=} \mathsf{UFld}(\mathsf{Fld}(O)) \,.\, l^T$ where $O \stackrel{\text{def}}{=} [l^T = \varsigma(z)\mathsf{Fld}([l^T = \mathsf{UFld}(\mathsf{Fld}(z)) \,.\, l^T])]$
- $[\![\Gamma \vdash (v.F)^N : \sigma]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \mathsf{Letval}\, n = [\![N]\!]\,\mathsf{in}\, it_v^{n;[\![F]\!]} \,.\, \mathsf{rec} : [\![\sigma]\!]$

FIGURE 9. Translating FIX_{op} into OBJ_{op}.

5. A SOUND TRANSLATION FOR OPERATIONAL JUDGEMENTS

5.1. THE TRANSLATION

We define a translation function $[\![-]\!]\colon FIX_{op} \to OBJ_{op}$ on the types and terms of FIX_{op}. The definition of this function is similar to the one given in Section 3. More precisely, the definition of $- \mapsto [\![-]\!]$ is the same as that in Figure 5, except that we use the auxiliary definitions given in Figure 8, and one new clause given in Figure 9.

5.2. A PROOF OF SOUNDNESS

The soundness of the encodings depends on three key lemmas. The motivation is the same as for the equational setting (see Sect. 3.3). If R is any binary relation, we shall write R^* for its reflexive, transitive closure.

Lemma 5.1. *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma, v\colon \sigma \vdash N : \tau$ *in FIX, then for* $[\![-]\!]\colon FIX \to OBJ_{op}$ *we have*

$$[\![N]\!]\{v \leftarrow [\![M]\!]\} \equiv [\![N\{v \leftarrow M\}]\!]$$

where $\equiv$ *denotes* ***syntactic identity.***

Proof. This is proved easily by induction on the structure of N. We omit the details. □

Lemma 5.2. *If* $\Theta, x\colon \sigma \vdash M : \tau$ *and* $\Theta \vdash N : \sigma$ *in* OBJ_{op} *, then*

$$\Theta \vdash fun_x^M \bullet N \rightsquigarrow^* \mathsf{Letinv}\, x = N\,\mathsf{in}\, M : \tau.$$

Proof. A simple calculation on OBJ_{op}. We have

$$\begin{aligned} fun_x^M \bullet N &\stackrel{\text{def}}{=} [\mathsf{arg} = \uparrow, \mathsf{val} = \varsigma(z)\mathsf{Letinv}\, v = z \,.\, \mathsf{arg} \text{ in } M] \,.\, \mathsf{arg} \Leftarrow N \,.\, \mathsf{val} \\ &\rightsquigarrow [\mathsf{arg} = N, \mathsf{val} = \varsigma(z)\mathsf{Letinv}\, v = z \,.\, \mathsf{arg} \text{ in } M] \,.\, \mathsf{val} \\ &\rightsquigarrow \mathsf{Letinv}\, v = [\mathsf{arg} = N, \mathsf{val} = \varsigma(z)\mathsf{Letinv}\, v = z \,.\, \mathsf{arg} \text{ in } M] \,.\, \mathsf{arg} \text{ in } M \\ &\rightsquigarrow M\{v \leftarrow N\}. \end{aligned}$$

□

Lemma 5.3. *All judgements in this lemma are in OBJ_{op}. Suppose that $\Theta \vdash M_i : \sigma_i$ where $1 \leq i \leq n$, and $\Theta, x_k\colon \sigma_k{}^{k \in S_j} \vdash N_j : \tau$ where $1 \leq j \leq m$ and $S_j \subset \{1, \ldots, n\}$ and $L_j \stackrel{\text{def}}{=} \{ l_k \mid k \in S_j \}$. Let*

$$GEN_X^{\sigma_1;\ldots;\sigma_n} \stackrel{\text{def}}{=} [l_i\colon \sigma_i{}^{i \in n}, \mathsf{cnxt}_{L_j}\colon X^{\,j \in m}, \mathsf{res}\colon X]$$

and also

$$gen_{X;j}^{M_1;\ldots;M_n} \stackrel{\text{def}}{=} [l_i = M_i{}^{i \in n}, \mathsf{cnxt}_{L_j} = \uparrow{}^{\,j \in m}, \mathsf{res} = \varsigma(z\colon GEN_X^{\sigma_1;\ldots;\sigma_n})z \,.\, \mathsf{cnxt}_{L_j}]$$

of type $GEN_X^{\sigma_1;\ldots;\sigma_n}$.

We shall also write (see Fig. 7 for the notation)

$$\begin{aligned} O \circ N_1 \circ N_2 \circ \ldots \circ N_m \stackrel{\text{def}}{=} (O \quad & \,.\, \mathsf{cnxt}_{L_1} \Leftarrow \varsigma(z)\mathsf{Letinv}\, x_k = z \,.\, l_k{}^{k \in S_1} \text{ in } N_1 \\ & \qquad \vdots \\ & \,.\, \mathsf{cnxt}_{L_m} \Leftarrow \varsigma(z)\mathsf{Letinv}\, x_k = z \,.\, l_k{}^{k \in S_m} \text{ in } N_m).\mathsf{res}. \end{aligned}$$

Then it follows that

$$\Theta \vdash (\Lambda(X.gen_{X;j}^{M_1;\ldots;M_n}))_\tau \circ N_1 \circ N_2 \circ \ldots \circ N_m \rightsquigarrow {}^* N_j\{x_k \leftarrow M_k{}^{k \in S_j}\} : \tau.$$

Proof. The proof is a simple calculation of reductions in OBJ_{op}.

$$\begin{aligned} (\Lambda(X.gen_{X;j}^{M_1;\ldots;M_n}))_\tau \circ N_1 \circ N_2 \circ \ldots \circ N_m &\rightsquigarrow gen_{\tau;j}^{M_1;\ldots;M_n} \circ N_1 \circ N_2 \circ \ldots \circ N_m \\ &\rightsquigarrow^* \xi \,.\, \mathsf{res} \\ &\rightsquigarrow \xi \,.\, \mathsf{cnxt}_{L_j} \\ &\rightsquigarrow \mathsf{Letinv}\, x_k = \xi \,.\, l_k{}^{k \in S_j} \text{ in } N_j \\ &\rightsquigarrow^* N_j\{x_k \leftarrow M_k{}^{k \in S_j}\}. \end{aligned}$$

where

$$\xi \stackrel{\text{def}}{=}$$

$$[l_i = M_i{}^{i\in n}, \mathsf{cnxt}_{L_j} = \varsigma(z)\mathsf{Letinv}\, x_k = z \,.\, l_k{}^{k\in S_j} \,\mathsf{in}\, N_j{}^{j\in m}, \mathsf{res} = \varsigma(z\colon GEN_X^{\sigma_1;\cdots;\sigma_n})z \,.\, \mathsf{cnxt}_{L_j}].$$

□

Theorem 5.4 states that object calculi encode *FIX* reductions:

Theorem 5.4. *We have given an encoding of FIX into* OBJ_{op}. *More precisely, if* $\Gamma \vdash M : \sigma$ *in FIX, then* $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\sigma]\!]$ *in* OBJ_{op}, *and if* $\Gamma \vdash M \rightsquigarrow M' : \sigma$ *in FIX, then* $[\![\Gamma]\!] \vdash [\![M]\!] \rightsquigarrow {}^*[\![M']\!] : [\![\sigma]\!]$ *in* OBJ_{op}.

Proof. The proof proceeds by induction on the derivations of type assignments and reductions. For type assignments, the details are routine, and we omit them. For reductions, we give a few details. Consider the first rule in Figure 6. We have to verify that if $\Gamma \vdash [\![P]\!] \rightsquigarrow [\![P']\!] : [\![(\sigma, \sigma')]\!]$ then $\Gamma \vdash [\![P]\!] \circ [\![N]\!] \rightsquigarrow [\![P']\!] \circ [\![N]\!] : [\![\tau]\!]$. Recall that $[\![P]\!] \circ [\![N]\!]$ is simply an update and invocation of the object $[\![P]\!]$; thus soundness follows from the OBJ_{op} rules

$$\frac{\Theta \vdash O \rightsquigarrow O' : \sigma}{\Theta \vdash O \,.\, l \rightsquigarrow O' \,.\, l : \sigma} \qquad \frac{\Theta \vdash O \rightsquigarrow O' : \sigma}{\Theta \vdash O \,.\, l \Leftarrow \varsigma(z)B \rightsquigarrow O' \,.\, l \Leftarrow \varsigma(z)B : \sigma}.$$

The soundness of the remaining rules in Figure 6 with a similar format follows analogously, apart from the rule

$$\frac{\Gamma \vdash N \rightsquigarrow N' : \mathit{fix}}{\Gamma \vdash (x.F)^N \rightsquigarrow (x.F)^{N'} : \sigma}.$$

Recalling that $[\![\Gamma \vdash (x.F)^N : \sigma]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!] \vdash \mathsf{Letval}\, n = [\![N]\!] \,\mathsf{in}\, it_x^{n;[\![F]\!]} \,.\, \mathsf{rec} : [\![\sigma]\!]$ one can see that soundness follows from the first rule in Figure 7.

The soundness of the rules in Figure 6 which involve the destruction of a constructor term (such as the second rule, but excluding the penultimate rule) follows from Lemmas 5.2 and 5.3.

This leaves the final two rules in Figure 6, for which we give an explicit calculation. One states that

$$\Gamma \vdash (x.F)^{s(E)} \rightsquigarrow F\{x \leftarrow \mathsf{Let}(E, n.\mathsf{Val}((x.F)^n))\} : \sigma.$$

We have, using Lemma 5.1 and setting $O \stackrel{\text{def}}{=} it_x^{\mathsf{Fld}([\![E]\!]);[\![F]\!]}$

$$
\begin{aligned}
[\![\Gamma]\!] \vdash [\![(x.F)^{s(E)}]\!] &\stackrel{\text{def}}{=} \mathsf{Letval}\,\mathsf{Fld}([\![E]\!]) = n \,\mathsf{in}\, it_x^{n;[\![F]\!]} \cdot \mathsf{rec} \\
&\rightsquigarrow O \cdot \mathsf{rec} \\
&\rightsquigarrow \mathsf{Letval}\, x = O \cdot l^T \,\mathsf{in}\, [\![F]\!] \\
&\rightsquigarrow \mathsf{Letval}\, x = \mathsf{Eval}^3_{l^T}([l^T = (\mathsf{Eval}^2_{\mathsf{pow}}(O \cdot \mathsf{pow} \Leftarrow \mathsf{UFld}(O \cdot \mathsf{pow}) \cdot l^T)) \cdot \mathsf{rec}]) \,\mathsf{in}\, [\![F]\!] \\
&\rightsquigarrow \mathsf{Letval}\, x = \mathsf{Eval}^2_{l^T}([l^T = (\mathsf{Eval}^1_{\mathsf{pow}}(O \cdot \mathsf{pow} \Leftarrow \mathsf{UFld}(\mathsf{Fld}([\![E]\!])) \cdot l^T)) \cdot \mathsf{rec}]) \,\mathsf{in}\, [\![F]\!] \\
&\rightsquigarrow \mathsf{Letval}\, x = \mathsf{Eval}^1_{l^T}([l^T = (O \cdot \mathsf{pow} \Leftarrow [\![E]\!] \cdot l^T) \cdot \mathsf{rec}]) \,\mathsf{in}\, [\![F]\!] \\
&\rightsquigarrow \mathsf{Letval}\, x = [l^T = it_x^{[\![E]\!].l^T;[\![F]\!]} \cdot \mathsf{rec}] \,\mathsf{in}\, [\![F]\!] \\
&\rightsquigarrow [\![F]\!]\{x \leftarrow [l^T = it_x^{[\![E]\!].l^T;[\![F]\!]} \cdot \mathsf{rec}]\} \\
&\equiv [\![F]\!]\{x \leftarrow [l^T = it_x^{n;[\![F]\!]} \cdot \mathsf{rec}]\{n \leftarrow [\![E]\!] \cdot l^T\}\} \\
&\equiv [\![F\{x \leftarrow \mathsf{Let}(E, n.\mathsf{Val}((x.F)^n))\}]\!] : [\![\sigma]\!].
\end{aligned}
$$

The last states that

$$\frac{\Gamma \vdash \omega : fix}{\Gamma \vdash \omega \rightsquigarrow s(\mathsf{Val}(\omega)) : fix}.$$

Setting $O \stackrel{\text{def}}{=} [l^T = \varsigma(z)\mathsf{Fld}([l^T = \mathsf{UFld}(\mathsf{Fld}(z)) \cdot l^T])]$ we have

$$
\begin{aligned}
[\![\Gamma]\!] \vdash [\![\omega]\!] &\stackrel{\text{def}}{=} \mathsf{UFld}(\mathsf{Fld}(O)) \cdot l^T \\
&\rightsquigarrow O \cdot l^T \\
&\rightsquigarrow \mathsf{Fld}([l^T = \mathsf{UFld}(\mathsf{Fld}(O)) \cdot l^T]) \\
&\equiv [\![s(\mathsf{Val}(\omega))]\!].
\end{aligned}
$$

□

6. Conclusions

In this paper we have attempted to explain, in a general and uniform manner, the ways in which object calculi are able to encode other type theories. In particular, we have shown that object calculi can encode many (simple) type systems, in a way which is reflective of the theory of arities and expressions. These ideas have been illustrated with examples by Abadi, Cardelli and others, but we have shown how these examples fit into a general framework. Further, we have given new encoding results for the computational let calculus and the *FIX* system. The precise details of the encoding are enlightening, especially with regard to the fixpoint type. Notice that Lemmas 3.2 and 3.3, together with Lemmas 5.2 and 5.3,

encapsulate the soundness of our translation. It is only those terms and equations associated with the *fix* or computation types which require separate treatment in the proofs of type soundness and soundness of equalities.

We have dealt with both equational and operational systems. The material on equational systems appears to work out very neatly. In contrast, the material on operational semantics is rather less pleasing. We have attempted to demonstrate, in a rather intentional way, that the operational behaviour of a well know type theory presented *via* transitions can be captured by the rather finer grained transitions of an object theory. However, in order to cope with the rather "neutral" object transition semantics, which is neither clearly call-by-value or call-by-name, we have introduced some additional syntax which forces expression evaluations. This is not particularly desirable. Of course, our results would perhaps look neater if we translated the FIX type theory into an "abstract machine for objects". In doing this, the high level term formers (such as $\mathsf{Eval}_l(M)$) would be rendered as simple machine instructions. The author and S. Ambler have considered the use of the results in this paper to provide a framework which will use object calculi as a mechanized metalanguage to represent programming language semantics. When mechanizing theories, uniformity is essential to reduce coding, and the results here provide this. In order to pursue this, we would ideally need stronger results about the properties of the translation, such as computational adequacy. More precisely, one might consider the standard *OBJ* operational semantics and associated program equivalences, and prove the translation adequate.

Other work includes an account of type theories with higher order contexts and their connection with objects; looking at direct implementations of the *FIX* theories and comparing them with *OBJ* implementations; and a deeper investigation into issues of eager and lazy reduction in the setting of object calculi.

I would like to thank two anonymous referees for comments on this paper. The first referee asked that the motivation and general ideas be made more clear; I think this resulted in a substantial improvement. The second referee pointed out a technical problem with my exposition of the OBJ_{op} semantics, which has now been remedied, and made a number of useful comments about the second half of the paper. Some of these comments are incorporated in this conclusion. Finally I would like to thank Simon Ambler (Leicester), Andy Gordon (Microsoft, Cambridge) and Guy McCusker (Oxford) for useful comments on the original draft of this paper.

REFERENCES

[1] M. Abadi and L. Cardelli, *A Theory of Objects*. Springer-Verlag, *Monogr. Comput. Sci.* (1996).

[2] R.L. Crole, Functional Programming Theory (1995). Department of Mathematics and Computer Science Lecture Notes, LaTeX format iv+68 pages with index.

[3] R.L. Crole and A.M. Pitts, New Foundations for Fixpoint Computations: FIX Hyperdoctrines and the FIX Logic. *Information and Computation* **98** (1992) 171–210. LICS '90 Special Edition of Information and Computation.

[4] A.D. Gordon, Bisimilarity as a theory of functional programming. *Electron. Notes Theor. Comput. Sci.* **1** (1995).
[5] A.D. Gordon, Everything is an object. Seminar Notes, Microsoft Research U.K. (1997).
[6] C.A. Gunter, *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press (1992).
[7] G. Kahn, Natural semantics, edited by K. Fuchi and M. Nivat, *Programming of Future Generation Computers.* Elsevier Science Publishers B.V. North Holland (1988) 237–258.
[8] Z. Luo, *Computation and Reasoning.* Oxford University Press, *Monogr. Comput. Sci.* **11** (1994).
[9] E. Moggi, Notions of computation and monads. *Theoret. Comput. Sci.* **93** (1989) 55–92.
[10] B. Nordström, K. Petersson and J.M. Smith, *Programming in Martin-Löf's Type Theory.* Oxford University Press, *Monogr. Comput. Sci.* (1990).
[11] A.M. Pitts, Operationally Based Theories of Program Equivalence, edited by P. Dybjer and A.M. Pitts, *Semantics and Logics of Computation* (1997).
[12] G.D. Plotkin, A structural approach to operational semantics. Technical Report DAIMI–FN 19. Department of Computer Science, University of Aarhus, Denmark (1981).
[13] G. Winskel, *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, Cambridge, Massachusetts (1993).

Communicated by Z. Ezik.
Received November 15, 1998. Accepted March 14, 2000.