Lecture Notes on Data Abstraction

15-814: Types and Programming Languages
Frank Pfenning

Lecture 15
Thursday, October 28, 2021

1 Introduction

Since we have moved from the pure λ-calculus to functional programming languages we have added rich type constructs starting from functions, disjoint sums, eager and lazy pairs, recursive types, and parametric polymorphism. The primary reasons often quoted for such a rich static type system are discovery of errors before the program is ever executed and the efficiency of avoiding tagging of runtime values. There is also the value of the types as documentation and the programming discipline that follows the prescription of types. Perhaps more important than all of these is the strong guarantees of data abstraction that the type system affords that are sadly missing from many other languages. Indeed, this was one of the original motivation in the development of ML (which stands for MetaLanguage) by Milner and his collaborators [GMM+78]. They were interested in developing a theorem prover and wanted to reduce its overall correctness to the correctness of a trusted core. To this end they specified an *abstract type* of *theorem* on which the only allowed operations are inference rules of the underlying logic. The connection between abstract types and existential types was made made Mitchell and Plotkin [MP88].

In this lecture we will first explore some more consequences of Reynolds's parametricity theorem that are used in modern compilers and then move towards questions of data abstraction and modularity.

2 Theorems for Free!

A slightly different style of application of parametricity is laid out in Philip Wadler's *Theorems for Free!* [Wad89]. Let's see what we can derive from

$$f : \forall\alpha.\, \alpha \to \alpha$$

for a value f. First, parametricity tells us

$$f \sim f \in [\forall\alpha.\, \alpha \to \alpha]$$

This time, we pick types τ and τ' and a *relation* R which is in fact a *function* $R : \tau \to \tau'$. Then

$$f[\tau] \approx f[\tau'] \in [\![R \to R]\!]$$

which means that $f[\tau] \mapsto^* f_\tau$ and $f[\tau'] \mapsto^* f_{\tau'}$ with

$$f_\tau \sim f_{\tau'} \in [R \to R]$$

Now, for arbitrary values $v : \tau$ and $v' : \tau'$, $v\ R\ v'$ actually means $R\,v \mapsto^* v'$. Using the definition of $\sim$ at function type we get

$$f_\tau\, v \approx f_{\tau'}\, (R\, v) \in [\![R]\!]$$

but this in turn means

$$R\,(f_\tau\, v) \mapsto^* w \quad \text{and} \quad f_{\tau'}\,(R\, v) \mapsto^* w \quad \text{for some value } w$$

Wadler summarizes this by stating that for any function $R : \tau \to \tau'$,

$$R \circ f[\tau] = f[\tau'] \circ R$$

that is, f commutes with any function R. If τ is non-empty and we have $v_0 : \tau$ and choose $\tau' = \tau$ and $R = \lambda x.\, v_0$ we obtain

$$\begin{array}{lll} R\,(f[\tau]\, v_0) & \mapsto^* & v_0 \\ f[\tau]\,(R\, v_0) & \mapsto^* & f[\tau]\, v_0 \end{array}$$

so we find $f[\tau]\, v_0 \mapsto^* v_0$ which, since v_0 was arbitrary, is another way of saying that f behaves like the identity function.

3 Parametricity on Lists

For more interesting examples, we extend the notion of logical equivalence to lists. Since lists are inductively defined, we can call upon a general theory to handle them, but since we haven't discussed this theory we give the specific definition. Here, we think of lists defined with

$$\mathsf{list}\ \tau = \mu\alpha.\,(\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \alpha)$$

even though type constructors like list haven't been formally introduced into our language. Then we use a shorthand notation for lists, that is, elaborate the left-hand side into the right-hand side:

$$[e_1, \ldots, e_n] \triangleq \mathsf{fold}\ \mathbf{cons} \cdot \langle e_1, \ldots \mathsf{fold}\ \mathbf{cons} \cdot \langle e_n, \mathsf{fold}\ \mathbf{nil} \cdot \langle\,\rangle\rangle\rangle$$

We then extend the notion of logical equalities to values of list type *inductively* over the structure of the list, which reduces the type of the relation because each element has a smaller type.

> $v \sim v' \in [\mathsf{list}\ \tau]$ *iff* $v = [v_1, \ldots, v_n]$, $v' = [v'_1, \ldots, v'_n]$ *and* $v_i \sim v'_i \in [\tau]$ *for all* $1 \le i \le n$.

Then we have, for example, a polymorphic *map* function:

$$\begin{array}{l} \mathit{map} : \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \beta \\ \mathit{map} = \Lambda\alpha.\,\Lambda\beta.\,\mathsf{fix}\ m.\,\lambda f.\,\lambda l. \\ \qquad \mathsf{case}\ (\mathsf{unfold}\ l)\ \ (\ \mathbf{nil} \cdot \langle\,\rangle \Rightarrow \mathsf{fold}\ \mathbf{nil} \cdot \langle\,\rangle \\ \qquad\qquad\qquad\qquad \mid \mathbf{cons} \cdot \langle x, l'\rangle \Rightarrow \mathsf{fold}\ \mathbf{cons} \cdot \langle f\ x, m\ f\ l'\rangle\ ) \end{array}$$

The map function then satisfies (for $f : \tau \to \tau'$):

$$\mathit{map}\ [\tau]\,[\tau']\,f\ [v_1, \ldots, v_n] = [f\ v_1, \ldots, f\ v_n]$$

where equality here is Kleene equality (both sides reduce to the same value). The example(s) are easier to understand if we isolate the special case $\mathsf{list}\ R$ for a relation $R : \tau \to \tau'$ which is actually a function. In this case we obtain

$$v \sim v' \in [\mathsf{list}\ R] \text{ for an } R : \tau \to \tau' \text{ iff } (\mathit{map}\ [\tau]\,[\tau']\,R)\,v = v'.$$

Returning to examples, what can the type tell us about a function

$$f : \forall\alpha.\,\mathsf{list}\ \alpha \to \alpha\ \mathsf{list}\ \alpha\ ?$$

If the function is parametric, it should not be able to examine the list elements, or create new ones. However, it should be able to drop elements,

duplicate elements, or rearrange them. We will try to capture this equationally, just following our nose in using parametricity to see what we end up at.

We start with

$f \sim f \in [\forall\alpha.\, \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha]$ *by parametricity.*

Now let $R : \tau \to \tau'$ be a function. Then $f[\tau] \mapsto^* f_\tau$, $f[\tau'] \mapsto^* f_{\tau'}$, and

$f_\tau \sim f_{\tau'} \in [\mathsf{list}\ R \to \mathsf{list}\ R]$ *by definition of* $\approx$.

Using the definition of $\sim$ on function types, we obtain

For any values $l : \mathsf{list}\ \tau$ *and* $l' : \mathsf{list}\ \tau'$ *with* $l\ (R\ \mathsf{list})\ l'$ *we have* $f_\tau\ l\ (R\ \mathsf{list})\ f_{\tau'}\ l'$

By the remark on the interpretation of $R\ \mathsf{list}$ when R is a function, this becomes

If $(map\ [\tau]\ [\tau']\ R)\ l = l'$ *then* $(map\ [\tau]\ [\tau']\ R)\ (f_\tau\ l) = f_{\tau'}\ l'$

or, equivalently,

$(map\ [\tau]\ [\tau']\ R)\ (f\ [\tau]\ l) = f\ [\tau']\ ((map\ [\tau]\ [\tau']\ R)\ l)$.

In short, f commutes with *map* R. This means we can either map R over the list and then apply f to the result, or we can apply f first and then map R over the result. This implies that f could not, say, make up a new element v_0 not in l. Such an element would occur in the list returned by the right-hand side, but would occur as $R\,v_0$ on the left-hand side. So if we have a type with more than one element we can choose R so that $R\,v_0 \neq v_0$ (like a constant function) and the two sides would be different, contradicting the equality we derived.

We can use this equation of improve efficiency of code. For example, if we know that f might reduce the number of elements in the list (for example, skipping every other element), then mapping R over the list after the elements have been eliminated is more efficient than the other way around. Conversely, if f may duplicate some elements then it would be more efficient to map R over the list first and then apply f. The equality we derived from parametricity allows this kind of optimization.

We have, however, to be careful when nonterminating functions may be involved. For example, if R diverges on an element v_0 then the two sides may not be equal. For example, f might drop v_0 from the list l so the right-hand side would diverge while the left-hand side would have a value.

Here are two other similar results provided by Wadler [Wad89].

$f : \forall\alpha.\,(\alpha\ \mathsf{list})\ \mathsf{list} \to \alpha\ \mathsf{list}$
$(\mathit{map}\ [\tau]\,[\tau']\,R)\,(f\ [\tau]\,l) = f\ [\tau']\,((\mathit{map}\ [\mathsf{list}\,\tau]\,[\mathsf{list}\,\tau']\,(\mathit{map}\ [\tau]\,[\tau']\,R))\ l)$

$f : \forall\alpha.\,(\alpha \to \mathit{bool}) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$
$(\mathit{map}\ [\tau]\,[\tau']\,R)\ (f\ [\tau]\,(\lambda x.\,p\,(R\,x))\,l) = f\ [\tau']\,p\,((\mathit{map}\ [\tau]\,[\tau']\,R)\,l)$

These theorems do not quite come "for free", but they are fairly straightforward consequences of parametricity, keeping in mind the requirement of termination.

4 Inductive Types

The theorems in the preceding section treat lists as a new primitive type. On the other hand, we can define every instance $\mathsf{list}\ \tau$ directly as a recursive type, so it is worth considering if we can find a more general characterization of a class of recursive types to which parametricity applies. These are *inductive types* whose values are defined purely inductively. We call these here *purely positive types*. Logical equality then does not have to reference computation at all and write τ^+. It is not a coincidence that we previously introduced this class as the types whose values are directly *observable*. To be explicit, we define the purely positive types as

$$\text{Purely positive types}\quad \tau^+ \quad ::= \quad \tau_1^+ \times \tau_2^+ \mid 1 \mid \textstyle\sum_{i\in I}(i : \tau_i^+) \mid \mu\alpha^+.\,\tau^+ \mid \alpha^+$$

Then we can define:

Recursion: $v \sim v' \in [\mu\alpha^+.\,\tau^+]$ iff $v = \mathsf{fold}\ v_1$ and $v' = \mathsf{fold}\ v_1'$ and $v_1 = v_1' \in [[\mu\alpha^+.\,\tau^+/\alpha^+]\tau^+]$.

Even though the type becomes larger in the last clause, the definition is not circular because the values we are comparing get smaller. In fact, we can prove that $v \sim v' \in [\tau^+]$ iff $v = v'$. So the clauses for positive types are mostly useful if negative types are embedded in them.

This characterization means that we can obtain "theorems for free" for free for functions operating over (polymorphic) purely positive types such as lists, trees, and similar structures.

5 Signatures and Structures

Data abstraction in today's programming languages is usually enforced at the level of modules (if it is enforced at all). As a running example we consider a simple module providing an implementation of a counter with constant *init* and functions *inc* and *dec* to increment and decrement the counter. We will consider two implementations and their relationship. One is using numbers in unary form (type *nat*) and numbers in binary form (type *bin*), and we will eventually prove that they are logically equivalent. We are making up some syntax (loosely based on ML) to specify interfaces between a library and its client.

Below we name *CTR* as the *signature* that describes the interface of a module.

$$
\begin{array}{l}
\mathit{CTR} = \{ \\
\quad \mathbf{type}\ \mathit{ctr} \\
\quad \mathit{init} : \mathit{ctr} \\
\quad \mathit{inc} : \mathit{ctr} \to \mathit{ctr} \\
\quad \mathit{dec} : \mathit{ctr} \to (\mathbf{none} : 1) + (\mathbf{some} : \mathit{ctr}) \\
\}
\end{array}
$$

The value *init* will be a counter with initial value 0. The decrement function *dec* returns an optional counter with the new value, where we consider the predecessor of 0 to be undefined (returning $\mathbf{l} \cdot \langle\rangle$). This provides the only means for the client to observe the value of a counter. The implementations are straightforward so we elide them for now, and just assume we have type *nat* and *bin* and suitable functions on them.

$$
\begin{array}{l}
\mathit{NatCtr} : \mathit{CTR} = \{ \\
\quad \mathbf{type}\ \mathit{ctr} = \mathit{nat} \\
\quad \mathit{init} = \mathit{zero} \\
\quad \mathit{inc} = \mathit{succ} \\
\quad \mathit{dec} = \mathit{pred}' \\
\}
\end{array}
$$

An interesting aspect of this definition is that, for example, $\mathit{zero} : \mathit{nat}$ while the interface specifies $\mathit{init} : \mathit{ctr}$. But this is okay because the type *ctr* is in fact implemented by *nat* in this version. Next, we show the implementation using numbers in binary representation (type *bin*). Again, we assume we have suitable functions *plus1* and *minus1* on binary numbers already defined.

$$\mathit{bin} = (\mathbf{b0} : \mathit{bin}) + (\mathbf{b1} : \mathit{bin}) + (\mathbf{e} : 1)$$

$$
\begin{array}{l}
\mathit{bzero} : \mathit{bin} \\
\mathit{bsucc} : \mathit{bin} \to \mathit{bin} = \ldots \\
\mathit{bpred}' : \mathit{bin} \to \mathit{bin} = \ldots \\
\\
\mathit{BinCtr} : \mathit{CTR} = \{ \\
\quad \mathbf{type}\ \mathit{ctr} = \mathit{bin} \\
\quad \mathit{init} = \mathit{bzero} \\
\quad \mathit{inc} = \mathit{bsucc} \\
\quad \mathit{dec} = \mathit{bpred}' \\
\}
\end{array}
$$

Now what does a client look like? Assume it has an implemention $C : \mathit{CTR}$. It can then "open" or "import" this implementation to use its components, but it will not have any knowledge about the type of the implementation. For example, we would write

$$
\begin{array}{l}
\mathbf{open}\ C : \mathit{CTR} \\
\mathit{isZero} : \mathit{ctr} \to \mathit{bool} \\
\mathit{isZero} = \lambda x.\, \mathsf{case}\ \mathit{dec}\ x\ (\ \mathbf{none} \cdot \_ \Rightarrow \mathit{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{some} \cdot \_ \Rightarrow \mathit{false}\ )
\end{array}
$$

but **not**

$$
\begin{array}{l}
\mathbf{open}\ C : \mathit{CTR} \\
\mathit{isZero} : \mathit{ctr} \to \mathit{bool} \\
\mathit{isZero} = \lambda n.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\ \mathbf{zero} \cdot \_ \Rightarrow \mathit{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{succ} \cdot \_ \Rightarrow \mathit{false}\ )
\end{array}
$$

because the latter supposes that the library $C : \mathit{CTR}$ implements the type *ctr* by *nat*, which it may not.

6 Formalizing Abstract Types

We will write a signature such as

$$
\begin{array}{l}
\mathit{CTR} = \{ \\
\quad \mathbf{type}\ \mathit{ctr} \\
\quad \mathit{init} : \mathit{ctr} \\
\quad \mathit{inc} : \mathit{ctr} \to \mathit{ctr} \\
\quad \mathit{dec} : \mathit{ctr} \to 1 + \mathit{ctr} \\
\}
\end{array}
$$

in abstract form as

$$\exists\alpha.\ \underbrace{\alpha}_{\mathit{init}} \times \underbrace{(\alpha \to \alpha)}_{\mathit{inc}} \times \underbrace{(\alpha \to 1 + \alpha)}_{\mathit{dec}}$$

where the name annotations are just explanatory and not part of the syntax. Note that α stands for *ctr* which is bound here by the existential quantifier.

Now what should an expression

$$e : \exists\alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to 1 + \alpha)$$

look like? It should provide a concrete implementation type (such as *nat* or *bin*) for α, as well as an implementation of the three functions. We obtain this with the following rule

$$\frac{\Gamma \vdash \sigma\ \mathit{type} \quad \Gamma \vdash e : [\sigma/\alpha]\tau}{\Gamma \vdash \langle [\sigma], e\rangle : \exists\alpha.\,\tau}\ \mathsf{tp/paire}$$

Besides checking that σ is indeed a type with respect to all the type variables declared in Γ, the crucial aspect of this rule is that the implementation e is at type $[\sigma/\alpha]\tau$.

For example, to check that *init*, *inc*, and *dec* are well-typed we substitute the implementation type for α (namely *nat* in one case and *bin* in the other case) before we proceed to check the definitions.

The pair $\langle [\sigma], e\rangle$ is sometimes referred to as a *package*, which is opened up by the destructor. This destructor is often called open, but for uniformity with all analogous cases, and to support general pattern matching, we'll write is as a case.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \ldots \mid \exists\alpha.\,\tau \\ \text{Expressions} & e & ::= & \ldots \mid \langle\sigma, e\rangle \mid \mathsf{case}\ e\ (\langle\alpha, x\rangle \Rightarrow e') \end{array}$$

The elimination form provides a new name α for the implementation types and a new variable x for the (eager) pair making up the implementations.

$$\frac{\Gamma \vdash e : \exists\alpha.\,\tau \qquad \begin{array}{l}(\alpha \notin \mathrm{dom}(\Gamma) \cup \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\tau')) \\ \Gamma, \alpha\ \mathit{type}, x : \tau \vdash e' : \tau'\end{array}}{\Gamma \vdash \mathsf{case}\ e\ (\langle\alpha, x\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}$$

The fact that the type α must be *new* is explicit here in the conditions that it does not already appear in Γ or τ' (that is, is not in the set of its free

type variables $\mathrm{FTV}(-)$. Such a condition is often left implicit, relying on the well-formedness presuppositions of the judgments. For example, the presupposition that Γ may not contain any repeated variables means that if we happened to have used the name α before then we can just rename it and then apply the rule. It is crucial for data abstraction that this variable α is new because we cannot and should not be able to assume anything about what α might stand for, except the operations that are exposed in τ and are accessible via the name x. Among other things, α may not appear in τ'.

To be a little more explicit about this (because it is critical here), whenever we write $\Gamma \vdash e : \tau$ we make the following *presuppositions*:

1. All the variables and type variables in Γ are distinct.
2. Γ *ctx*
3. $\Gamma \vdash \tau$ *type*

where the validity of context is defined by the following rules:

$$\frac{}{(\cdot)\ \mathit{ctx}}\ \mathsf{ctx/emp} \qquad \frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \mathsf{ctx/tpvar} \qquad \frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \mathsf{ctx/var}$$

With these presuppositions the condition on α in the tp/casee rule is automatically satisfied. Whenever we write a rule we assume this presuppositions holds for the conclusion and we have to make sure they hold for all the premises. Let's look at case/exists again in this light.

1. We assume all variables in Γ are distinct, which also means they are distinct in the first premise. In the second premise they are distinct because that's how we interpret $\Gamma, \alpha\ \mathit{type}, x : \tau$, which may include an implicit renaming of the type variable α or the variable x bound in the the expression $\langle \alpha, x \rangle \Rightarrow e'$.
2. By presupposition (from the conclusion), Γ *ctx*, which means that there are no *free* type variables in it, but variables declared in it can occur to their right. But what about τ? Actually, it is okay (and in fact mostly needed) for α to appear in τ.
3. By presupposition (from the conclusion), $\Gamma \vdash \tau'$ *type*. This covers the second premise. Often, this rule is given with an explicit premise $\Gamma \vdash \tau'$ *type* to emphasize τ' must be independent of α. Indeed, the scope of α is the type of x and the expression e'.

We also see that the client e' is *parametric* in α, which means that it cannot depend on what α might actually be at runtime. It is this parametricity that will allow us to swap one implementation out for another without affecting the client as long as the two implementations are equivalent in an appropriate sense.

The dynamics is straightforward and not very interesting.

$$\frac{v\ \mathit{value}}{\langle[\sigma], v\rangle\ \mathit{value}}\ \mathsf{val/paire} \qquad \frac{e \mapsto e'}{\langle[\sigma], e\rangle \mapsto \langle[\sigma], e'\rangle}\ \mathsf{step/pack}_1$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle\alpha, x\rangle \Rightarrow e_1) \mapsto \mathsf{case}\ e_0'\ (\langle\alpha, x\rangle \Rightarrow e_1)}\ \mathsf{step/casee}_0$$

$$\frac{}{\mathsf{case}\ \langle[\sigma], v\rangle\ (\langle\alpha, x\rangle \Rightarrow e) \mapsto [\sigma/\alpha, v/x]e}\ \mathsf{step/casee/paire}$$

The hypothetical open construct now corresponds a pattern match, with the scope of the openend module extending to the end of the case expression. For example, we can test an implementation of *CTR* by creating a fresh counter and then verifying that incrementing it followed by a decrement has a well-defined answer. In the definition of *test* we exploit general pattern matching so an exception is raised if a decrement of zero was attempted.

$$\begin{array}{l}\mathit{test} : \mathit{CTR} \to 1 \\ \mathit{test} = \lambda c.\ \mathsf{case}\ c\ (\ \langle\alpha, \langle \mathit{init}, \langle \mathit{inc}, \mathit{dec}\rangle\rangle\rangle \Rightarrow \\ \qquad\qquad \mathsf{case}\ \mathit{dec}\ (\mathit{inc}\ \mathit{init})\ (\ \mathbf{some} \cdot \_ \Rightarrow \langle\,\rangle\ )\ )\end{array}$$

Note that the case opens the package (representing the module) and matches against its components so it can refer to them in the body of the function. The following two expressions will evaluate to unit (instead of raising an exception) if the implementation is correct (to the very limited extent that is tested here).

test NatCtr
test BinCtr

7 Existential Types and Parametricity

We have said that the client of a module (expressed as having an existential type) is parametric in the implementation type. Let's recall the crucial rules.

$$\frac{\Gamma \vdash \sigma \; \mathit{type} \quad \Gamma \vdash e : [\sigma/\alpha]\tau}{\Gamma \vdash \langle [\sigma], e \rangle : \exists \alpha.\, \tau} \; \mathsf{tp/paire}$$

$$\frac{\Gamma \vdash e : \exists \alpha.\, \tau \quad \Gamma, \alpha \; \mathit{type}, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\; e\; (\langle \alpha, x \rangle \Rightarrow e') : \tau'} \; \mathsf{tp/casee}$$

The client here is e' in the tp/casee rule. From typing judgment for e' in the second premise we can infer

$$\frac{\dfrac{\Gamma, \alpha \; \mathit{type}, x : \tau \vdash e' : \tau'}{\Gamma, \alpha \; \mathit{type} \vdash \lambda x.\, e' : \tau \to \tau'} \; \mathsf{tp/lam}}{\Gamma \vdash \Lambda \alpha.\, \lambda x.\, e' : \forall \alpha.\, \tau \to \tau'} \; \mathsf{tp/tplam}$$

to see that, indeed, $\lambda x.\, e'$ (and therefore also e') is parametric in α.

8 Logical Equality for Existential Types

We extend our definition of logical equivalence to handle the case of existential types. Following the previous pattern for parametric polymorphism, we cannot talk about arbitrary instances of the existential type, but we must instantiate it with a relation between the two given implementation types.

Recall from Lecture 14:

(∀) $v \sim v' \in [\forall \alpha.\, \tau]$ iff for all closed types σ and σ' and relations $R : \sigma \leftrightarrow \sigma'$ we have $v\,[\sigma] \approx v'\,[\sigma'] \in [\![[R/\alpha]\tau]\!]$

(*R*) $v \sim v' \in [R]$ iff $v \; R \; v'$.

We add

(∃) $v \sim v' \in [\exists \alpha.\, \tau]$ iff $v = \langle [\sigma], v_1 \rangle$ and $v' = \langle [\sigma'], v_1' \rangle$ for some closed types σ, σ' and values v_1, v_1', and there is a relation $R : \sigma \leftrightarrow \sigma'$ such that $v_1 \sim v_1' \in [[R/\alpha]\tau]$.

It is critical that we only need to find *some* relation R between the types; requiring this to hold for all relations would be much to strong a condition. We can see this from the example: the two implementations of counters should be equivalent from the clients point of view. But there are many relations between *nat* and *bin* where the implementations are not related, such as one that swaps 0 and 1. Then *zero* is not related to $\mathsf{fold}\ (\mathsf{e} \cdot \langle\rangle)$ and we would refute the equivalence of the unary and binary implementations of the counter signature.

References

[GMM+78] Michael J.C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In A. Aho, S. Zillen, and T. Szymanski, editors, *Conference Record of the 5th Annual Symposium on Principles of Programming Languages (POPL'78)*, pages 119–130, Tucson, Arizona, January 1978. ACM Press.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[Wad89] Philip Wadler. Theorems for free! In J. Stoy, editor, *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359, London, UK, September 1989. ACM.