Lecture Notes on Representation Independence

15-814: Types and Programming Languages
Frank Pfenning

Lecture 16
Tuesday, November 2, 2021

1 Introduction

In this lecture we prove that we can replace the unary implementation of counters with the binary one without breaking any clients (or vice versa). This is a consequence of parametricity, and the definition of logical equality we developed in the previous two lectures, extended to existential types.

We recall the definition of logical equality from the end of the last lecture.

(∃) $v \sim v' \in [\exists\alpha.\,\tau]$ 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 $v_1 \sim v_1' \in [[R/\alpha]\tau]$.

In our example, we ask if

$$\mathit{NatCtr} \sim \mathit{BinCtr} \in [CTR]$$

which unfolds into demonstrating that there is a relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ such that

$$\langle \mathit{zero}, \langle \mathit{succ}, \mathit{pred}' \rangle \rangle \sim \langle \mathit{bzero}, \langle \mathit{bsucc}, \mathit{bpred}' \rangle \rangle \in [R \times (R \to R) \times (R \to 1 + R)]$$

Since logical equality at type $\tau_1 \times \tau_2$ just decomposes into logical equality at the component types, this just decomposes into three properties we need to check. The key step is to define the correct relation R.

For reference, the complete implementation can be found in `absnat.cbv`. In Listing 1 we show the implementation *NatCtr* and *BinCtr* in LAMBDA. The concrete syntax for an existential type $\exists\alpha.\,\tau$ is `?a.tau`, and a package

```
type bin = $bin. ('b0 : bin) + ('b1 : bin) + ('e : 1)

decl bzero : bin
decl bsucc : bin -> bin
decl bpred : bin -> ('zero : 1) + ('succ : bin)

defn bzero = fold 'e ()
defn bsucc = $bsucc. \x. case (unfold x) of
               ( 'b0 y => fold 'b1 y
               | 'b1 y => fold 'b0 (bsucc y)
               | 'e () => fold 'b1 (fold 'e ()) )
defn bpred = $bpred. \x. case (unfold x) of
               ( 'b0 y => case bpred y of
                            ( 'zero () => 'zero ()
                            | 'succ p => 'succ (fold 'b1 p) )
               | 'b1 y => 'succ (fold 'b0 y)
% may introduce leading 0s
               | 'e () => 'zero () )

type NAT = ?a. a * (a -> a) * (a -> ('zero : 1) + ('succ : a))

decl UNat : NAT
decl BNat : NAT

defn UNat = ([nat], zero, succ, out)
defn BNat = ([bin], bzero, bsucc, bpred)
```

Listing 1: Binary counters as an abstract type

$\langle [\sigma], e \rangle$ is written as `([sigma],e)`. This notation means that, uniformly, types occurring in expressions are enclosed in square brackets. In this code, we have replaced the **none** and some labels by **zero** and **succ**, because the implementation of *NAT* by unary numbers then just becomes unfold.

```
type nat = $nat. ('zero : 1) + ('succ : nat)
decl zero : nat
decl succ : nat -> nat
decl out : nat -> ('zero : 1) + ('succ : nat)

defn zero = fold 'zero ()
defn succ = \n. fold 'succ n
defn out = \n. unfold n
```

In fact, the implementation of *NAT* illustrates a general phenomenon that is

exploited in languages such as Standard ML, OCaml, or Haskell. In these languages we have data type declarations that are *generative* in the sense that each declaration will generate a fresh type that's different from all other existing types. For example,

```
datatype nat = zero | succ of nat
```

will generate a fresh type called `nat` with *constructors* `zero : nat` and `succ : nat -> nat`. However (at least conceptually) is also generates a function `out : nat -> ('zero : 1) + ('succ : nat)` with unique labels `'zero` and `'succ` so we can pattern-match against the structure of natural numbers. These functions are then packaged up as an existential type to guarantee generativity, which is then opened to make the constructors and `out` function available in subsequent code. This is one idea behind the Harper/Stone semantics of Standard ML [HS00].

2 Defining a Relation Between Implementations

The relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ we seek needs to relate natural numbers in two different representations. It is convenient and general to define such relations by using inference rules. In particular, this will allow us to prove properties by *rule induction*. An alternative approach would be to define such relations as functions, but because representations are often not unique this is not quite as general.

Once we have made this decision, the relation could be based on the structure of $x : \mathit{bin}$ or on the structure of $n : \mathit{nat}$. The latter may run into difficulties because each number actually corresponds to infinitely many numbers in binary form: just add leading zeros that do not contribute to its value. Therefore, we define it based on the binary representation. In order to define it concisely, we use a representation function for (mathematical) natural numbers k into our language of values defined by

$$\begin{array}{lcl} \overline{0} & = & \mathsf{fold}\ \mathbf{zero} \cdot \langle\rangle \\ \overline{n+1} & = & \mathsf{fold}\ \mathbf{succ} \cdot \overline{n} \end{array}$$

We also write binary number representations in compressed form with the least significant bit first:

$$\begin{array}{lcl} 0x & = & \mathsf{fold}\ \mathbf{b0} \cdot x \\ 1x & = & \mathsf{fold}\ \mathbf{b1} \cdot x \\ \mathsf{e} & = & \mathsf{fold}\ \mathbf{e} \cdot \langle\rangle \end{array}$$

Recall the ambiguity that e, 0e, 00e etc. all represent the natural number 0.

We then define:

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e \qquad \frac{\overline{k}\ R\ x}{\overline{2k}\ R\ 0x}\ R_0 \qquad \frac{\overline{k}\ R\ x}{\overline{2k+1}\ R\ 1x}\ R_1$$

As usual, we consider $n\ R\ x$ to hold if and only if we can derive it using these rules.

3 Verifying the Relation

Because our signature exposes three constants, we now have to check three properties:

$$\begin{array}{l} \mathit{zero} \sim \mathit{bzero} \in [R] \\ \mathit{succ} \sim \mathit{bsucc} \in [R \to R] \\ \mathit{out} \sim \mathit{bpred} \in [R \to 1 + R] \end{array}$$

We already have by definition that $v \sim v' \in [R]$ iff $v\ R\ v'$. For convenience, we also define the notation $e\ \mathbb{R}\ e'$ to stand for $e \approx e' \in [\![R]\!]$, which means that $e \mapsto^* v$ and $e' \mapsto^* v'$ with $v\ R\ v'$

Lemma 1 *zero* $\sim$ *bzero* $\in [R]$.

Proof: Since $\overline{0} = \mathit{zero}$ and $\mathsf{e} = \mathit{bzero}$ this is just the contents of rule R_e. □

Lemma 2 *succ* $\sim$ *bsucc* $\in [R \to R]$.

Proof: By definition of logical equality, this is equivalent to showing

> *For all values* n : *nat and* x : *bin with* $n\ R\ x$ *we have* (*succ* n) $\mathbb{R}$ (*bsucc* x).

Since R is defined inductively by a collection of inference rules, the natural attempt is to prove this by rule induction on the given relation, namely $n\ R\ x$.

Case: Rule

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

with $n = \overline{0}$ and $x = \mathsf{e}$. We have to show that (*succ* $\overline{0}$) $\mathbb{R}$ (*bsucc* e)

succ $\overline{0} \mapsto^* \overline{1}$	By defn. of *succ*
bsucc $\mathsf{e} \mapsto^* 1\mathsf{e}$	By defn. of *bsucc*
$\overline{1}\ R\ 1\mathsf{e}$	By rules R_1 and R_e

Case: Rule

$$\frac{\overline{k}\ R\ y}{\overline{2k}\ R\ 0y}\ R_0$$

where $x = 0y$ and $n = \overline{2k}$. To prove is $(\mathit{succ}\ \overline{2k})\ \mathbb{R}\ (\mathit{bsucc}\ 0y)$.

succ $\overline{2k} \mapsto^* \overline{2k+1}$	By defn of *succ*
bsucc $0y \mapsto^* 1y$	By defn. of *bsucc*
$\overline{k}\ R\ y$	Premise in this case
$\overline{2k+1}\ R\ 1y$	By rule R_1

Case: Rule

$$\frac{\overline{k}\ R\ y}{\overline{2k+1}\ R\ 1y}\ R_1$$

where $n = \overline{2k+1}$ and $x = 1y$. To show: $(\mathit{succ}\ \overline{2k+1})\ \mathbb{R}\ (\mathit{bsucc}\ 1y)$.

succ $\overline{2k+1} \mapsto^* \overline{2k+2}$	By defn. of *succ*
bsucc $1y \mapsto^*$ *b0* $(\mathit{bsucc}\ y) \mapsto^* 0z$ where *bsucc* $y \mapsto^* z$	By defn. of *bsucc*
Remains to show: $\overline{2k+2}\ R\ 0z$	
$\overline{k}\ R\ y$	Premise in this case
$(\mathit{succ}\ \overline{k})\ \mathbb{R}\ (\mathit{bsucc}\ y)$	By ind. hyp.
$\overline{k+1}\ R\ z$	By defn. of $\mathbb{R}$ and *succ*
$\overline{2(k+1)}\ R\ 0z$	By rule R_0
$\overline{2k+2}\ R\ 0z$	By arithmetic

□

In order to prove the relation between the implementation of the predecessor function we write out the interpretation of the type $(\mathbf{zero} : 1) + (\mathbf{succ} : R)$.

$v \sim v' \in [(\mathbf{zero} : 1) + (\mathbf{succ} : R)]$ iff ($v = \mathbf{zero} \cdot \langle\,\rangle$ and $v' = \mathbf{zero} \cdot \langle\,\rangle$)
or ($v = \mathbf{succ} \cdot v_1$ and $v' = \mathbf{succ} \cdot v_1'$ and $v_1\ R\ v_1'$.

Lemma 3 *out* $\sim$ *bpred* $\in [R \to (\mathbf{zero} : 1) + (\mathbf{succ} : R)]$

Proof: By[1] definition of logical equality, this is equivalent to showing

> *For all values* n : *nat and* x : *bin with* $n\ R\ x$ *we have out* $n \approx$ *bpred* $x \in [\![1 + R]\!]$.

We break this down into two properties, based on n.

(i) For all $\overline{0}\ R\ x$ we have *out* $\overline{0} \approx$ *bpred* $x \in [\![(\mathbf{zero} : 1)]\!]$.

(ii) For all $\overline{k+1}\ R\ x$ we have *out* $\overline{k+1} \approx$ *bpred* $x \in [\![(\mathbf{succ} : R)]\!]$.

For part (i), we note that *out* $\overline{0} \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$, so all that remains to show is that *bpred* $x \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$ for all $\overline{0}\ R\ x$. We prove this by rule induction on the derivation of $\overline{0}\ R\ x$.

Case(i):

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

where $x = \mathsf{e}$. Then *bpred* $x =$ *bpred* $\mathsf{e} \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$.

Case(i):

$$\frac{\overline{k}\ R\ y}{\overline{2k}\ R\ \mathsf{0}y}\ R_0$$

where $x = \mathsf{0}y$ and $2k = 0$ and therefore also $k = 0$. Then

bpred $y \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$	By ind. hyp.
bpred $\mathsf{0}y \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$	By defn. of *bpred* and computation

Case(i):

$$\frac{\overline{k}\ R\ y}{\overline{2k+1}\ R\ \mathsf{1}y}\ R_1$$

This case is impossible since $2k + 1 \neq 0$.

Now we come to Part (ii). We note that *out* $\overline{k+1} \mapsto^* \mathbf{succ} \cdot \overline{k}$ so what we have to show is that

(ii)′ For all $\overline{k+1}\ R\ x$ we have *bpred* $x \mapsto^* \mathbf{succ} \cdot y$ with $\overline{k}\ R\ y$.

[1] We skipped this part of the proof in lecture.

We prove this by rule induction on the derivation of $\overline{k+1}\ R\ x$.

Case(ii):

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

is impossible since $\overline{0} \neq \overline{k+1}$.

Case(ii):

$$\frac{\overline{j}\ R\ y}{\overline{2j}\ R\ \mathsf{0}y}\ R_0$$

where $k+1 = 2j$ and $x = \mathsf{0}y$.

$j = j' + 1$ for some j'	Since $j > 0$ by arithmetic
bpred $y \mapsto^* \mathbf{succ} \cdot z$ with $\overline{j'}\ R\ z$	By ind. hyp.
bpred $\mathsf{0}y \mapsto^* \mathbf{succ} \cdot \mathsf{1}z$	By defn. of *bpred*
$\overline{2j'+1}\ R\ \mathsf{1}z$	By rule R_1
$\overline{k}\ R\ \mathsf{1}z$	By equality

Case(ii):

$$\frac{\overline{j}\ R\ y}{\overline{2j+1}\ R\ \mathsf{1}y}\ R_1$$

for $k+1 = 2j+1$ and $x = \mathsf{1}y$. Then

bpred $\mathsf{1}y \mapsto^* \mathbf{succ} \cdot \mathsf{0}y$	By defn. of *bpred*
$\overline{j}\ R\ y$	Premise in this case
$\overline{2j}\ R\ \mathsf{0}y$	By rule R_0
$\overline{k}\ R\ \mathsf{0}y$	By equality

□

4 Concrete Types vs. Abstract Types[2]

An interesting observation about the logical equivalence of the two implementation of counters is that, had we omitted the decrement operation from

[2]Not covered in lecture this year

the interface, then universal relation ($n\ U\ x$ for all values $n : \mathit{nat}$ and $x : \mathit{bin}$) also allows us to prove equivalence. This is because without the decrement we can create a counter and increment it, but can never observe any of its properties.

This raises the question how we should more generally observe properties of elements of abstract type. There is no universal answer: different applications or libraries require different choices. A particularly frequent and useful technique is to endow abstract types with a *view*, realized by a function called *expose* or *out*.

As an example, let's reconsider the (concrete) type of binary numbers:

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

This concrete type allows clients to construct numbers with leading zeros, which may be undesirable because it complicates certain algorithms (e.g., equality of binary numbers). In this case, one solution would be to split the type *bin* into positive numbers *pos* and numbers in standard form *std* (with no leading zeros), which we did in an earlier exercise. However, now all client code has to be aware of these two types and use them appropriately. Alternatively, we can create an abstract type providing the constructors in the interface. to start with, we would have

$$\begin{array}{lll} \mathit{BIN} = \exists\alpha.\ (\alpha \to \alpha) & & \%\ \mathit{b0} \\ \qquad\qquad \times(\alpha \to \alpha) & & \%\ \mathit{b1} \\ \qquad\qquad \times\alpha & & \%\ \mathit{e} \\ \qquad\qquad \times \ldots & & \end{array}$$

The implementation of these constructors can make sure that only numbers with no leading zeros are ever created. But how do we *observe* a value of the abstract type? The technique is to provide a function $\mathit{out} : \alpha \to \tau$ where τ is usually a sum that the client can pattern match against. Here we would have

$$\begin{array}{lll} \mathit{BIN} = \exists\alpha.\ (\alpha \to \alpha) & & \%\ \mathit{b0} \\ \qquad\qquad \times(\alpha \to \alpha) & & \%\ \mathit{b1} \\ \qquad\qquad \times\alpha & & \%\ \mathit{e} \\ \qquad\qquad \times(\alpha \to (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)) & & \%\ \mathit{out} \end{array}$$

The result $\mathit{out}\ v$ where v is a value of the abstract type allows one level of pattern matching. The value tagged by **b0** or **b1** is again of abstract type and we must apply *out* again. If we want to allow multiple levels of pattern matching we would need some special syntax to designate *out* as a view

with a corresponding pattern constructor, say, out^{-1}. Then matching the value $v : \alpha$ against the pattern $out^{-1}\ p$ will evaluate $out\ v \mapsto^* w$ and match w against p.

We show the implementation of this abstract type in LAMBDA. In this example, the *out* function just has to unfold the recursive type to expose the sum underneath.

```
type BIN = ?a. (a -> a)    % b0 = \n. 2n
             * (a -> a)    % b1 = \n. 2n+1
             * a           % e = 0
             * (a -> (('b0 : a) + ('b1 : a) + ('e : 1))) % out

decl Bin : BIN
defn Bin = ([bin], \x. case x of ( fold 'e () => e
                                 | _  => fold 'b0 x ),
                   \x. fole 'b1 x, e, \x. unfold x)
```

The only other interesting part of this is the constructor corresponding to the tag **b0** ensures that it never constructs 0e but returns e instead, thereby making the representation unique.

5 Polymorphic Lists

In functional languages lists are usually represented by a so-called *type constructor list* : *type* $\rightarrow$ *type*. That is, for any type τ, we would have

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

We have not introduced type constructors into our language, so we cannot express this directly. But we can formulate it as an abstract type. Essentially, the implementation is a *function* which takes an element type τ as an argument and returns in instance of an existential type for this particular τ.

```
type LIST = !a. ?b. b             % nil
                  * (a * b -> b)  % cons x l
                  * (b -> ('nil : 1) + ('cons : a * b)) % out l
```

There is, however, a quirk with the implementation that often comes up with abstract types. If we have an implementation of lists, for example

```
decl List : LIST
defn List = /\a. ([$list. ('nil : 1) + ('cons : a * list)],
                  fold 'nil (),
                  \p. fold 'cons p,
                  \l. unfold l)
```

then two different uses of this, e.g., *List* [*nat*] and *List* [*nat*] are incompatible because there is no way the type checker can know that the different abstract types are actually equal. We summarize this sometimes by saying that abstract types are *generative* as explained earlier in this lecture because every time an implementation of an abstract type is opened, a fresh type variable is generated to stand for the implementation type.

This implementation of lists, by the way, is called a *functor* in languages in the ML family, because it is a module-level function. We think of it this way because it is a function that returns an abstract type when given a type.

6 The Upshot

Because the two implementations are logically equal we can replace one implementation by the other without changing any client's behavior. This is because all clients are parametric, so their behavior does not depend on the library's implementation.

It may seem strange that this is possible because we have picked a particular relation to make this proof work. Let us reexamine the tp/casee rule:

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

In the second premise we see that the client e' is checked with a fresh type α and $x : \tau$ which may mention α. If we reify this into a function, we find

$$\Lambda\alpha.\,\lambda x.\,e' : \forall\alpha.\,\tau \to \tau'$$

where τ' does not depend on α.

By Reynolds's parametricity theorem we know that this function is parametric. This can now be applied for any σ and σ' and relation $R : \sigma \leftrightarrow \sigma'$ to conclude that if $v_0 \sim v_0' \in [[R/\alpha]\tau]$ then $(\Lambda\alpha\lambda x.\,e')[\sigma]\,v_0 \approx (\Lambda\alpha.\,\lambda x.\,e')[\sigma']\ v_0' \in [\![[R/\alpha]\tau']\!]$. But α does not occur in τ', so this is just saying that $[\sigma/\alpha, v_0/x]e' \approx [\sigma'/\alpha, v_0'/x]e' \in [\![\tau']\!]$. So the result of substituting the two different implementations is equivalent.

Exercises

Exercise 1 We can represent integers a as pairs $\langle x, y\rangle$ of natural numbers where $a = x - y$. We call this the *difference representation* and call the repre-

sentation type *diff*.

$$\begin{array}{lcl} \mathit{nat} & = & \mu\alpha.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{alpha}) \\ \mathit{diff} & = & \mathit{nat} \times \mathit{nat} \end{array}$$

In your answers below you may use *constructors zero* : *nat* and *succ* : *nat*→*nat* to construct terms of type *nat*. If you need auxiliary functions on natural numbers, you should define them.

1. Define a function *nat2diff* : *nat* → *diff* that, when given a representation of the natural number n returns an integer representing n.

2. Define a constant *d_zero* : *diff* representing the integer 0 as well as functions *dplus* : *diff* → *diff* → *diff* and *dminus* : *diff* → *diff* → *diff* representing addition and subtraction on integers, respectively.

3. Consider the type

   $$\mathit{ord} = (\mathbf{lt} : 1) + (\mathbf{eq} : 1) + (\mathbf{gt} : 1)$$

   that represents the outcome of a comparison (**lt** = "less than", **eq** = "equal", **gt** = "greater than"). Define a function *dcompare* : *diff* → *diff* → *ord* to compare the two integer arguments. Again, you may use *lt*, *eq* and *gt* as constructors.

Exercise 2 We consider an alternative *signed representation* of integers where

$$\mathit{sign} = (\mathbf{pos} : \mathit{nat}) + (\mathbf{neg} : \mathit{nat})$$

where $\mathbf{pos} \cdot x$ represents the integer x and $\mathbf{neg} \cdot x$ represents the integer $-x$. In your answers below you may use *pos* and *neg* as data constructors, to construct elements of type *sign*. Define the following functions in analogy with Exercise 1:

1. *nat2sign* : *nat* → *sign*

2. *s_zero* : *sign*

3. *s_plus* : *sign* → *sign* → *sign*

4. *s_minus* : *sign* → *sign* → *sign*

5. *s_compare* : *sign* → *sign* → *ord*

Exercise 3 In this exercise we pursue two different implementations of an integer counter, which can become negative (unlike the natural number counter in this lecture). The functions are simpler than the ones in Exercise 1 and Exercise 2 so that the logical equality argument is more manageable. We specify a signature

$$
\begin{array}{l}
\mathit{INTCTR} = \{ \\
\quad \mathbf{type}\ \mathit{ictr} \\
\quad \mathit{new} : \mathit{ictr} \\
\quad \mathit{inc} : \mathit{ictr} \to \mathit{ictr} \\
\quad \mathit{dec} : \mathit{ictr} \to \mathit{ictr} \\
\quad \mathit{is0} : \mathit{ictr} \to \mathit{bool} \\
\}
\end{array}
$$

where *new*, *inc*, *dec* and *is0* have their obvious specification with respect to integers, generalizing the *CTR* type defined in the last lecture and used in this one.

1. Write out the definition of *INTCTR* as an existential type.
2. Define the constants and functions *d_zero*, *d_inc*, *d_dec* and *d_is0* for the implementation where type $\mathit{ictr} = \mathit{diff}$ from Exercise 1.
3. Define the constants and functions *szero*, *s_inc*, *s_dec* and *s_is0* for the implementation where type $\mathit{ictr} = \mathit{sign}$ from Exercise 2.

Now consider the two definitions

$$
\begin{array}{l}
\mathit{DiffCtr} : \mathit{INTCTR} = \langle \mathit{diff}, \langle \mathit{d\_zero}, \mathit{d\_inc}, \mathit{d\_dec}, \mathit{d\_is0} \rangle \rangle \\
\mathit{SignCtr} : \mathit{INTCTR} = \langle \mathit{sign}, \langle \mathit{s\_zero}, \mathit{s\_inc}, \mathit{s\_dec}, \mathit{s\_is0} \rangle \rangle
\end{array}
$$

4. Prove that $\mathit{DiffCtr} \sim \mathit{SignCtr} \in [\mathit{INTCTR}]$ by defining a suitable relation $R : \mathit{diff} \leftrightarrow \mathit{sign}$ and proving that

$$
\begin{array}{c}
\langle \mathit{d\_zero}, \mathit{d\_inc}, \mathit{d\_dec}, \mathit{d\_is0} \rangle \sim \langle \mathit{s\_zero}, \mathit{s\_inc}, \mathit{s\_dec}, \mathit{s\_is0} \rangle \\
\in [R \times (R \to R) \times (R \to R) \times (R \to \mathit{bool})]
\end{array}
$$

References

[HS00] Robert Harper and Christopher A. Stone. A type-theoretic interpretation of Standard ML. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 341–388. MIT Press, 2000.