Programming Languages
Data Abstraction
Robert Harper
Spring, 1999

What is Data Abstraction?

Reynolds: "Types are a syntactic mechanism for enforcing levels of abstraction."

What does this mean in practical terms? How can we use a type system to enforce abstraction?

Can we precisely characterize what we mean by an abstract type?

Type Safety and Abstraction

Type safety is a manifestation of Reynolds' principle!

Type safety states that there is no "confusion" between values of distinct types.

1. Cannot apply a number to a boolean.

2. Cannot add two functions.

The representations of values are hidden from the programmer.

Type Safety and Abstraction

MinML types are abstract because we can only use the designated operations for creating and using values of that type.

- Integers are "created" by writing numerals, and are used by the arithmetic and comparison operations.

- Booleans are "created" by `true` and `false` and are used by the conditional expression.

- Functions are "created" by lambda abstractions, and are used by application.

Abstract Types

An abstract type consists of two things:

1. A **representation type** for the values of the ADT. The representation type is hidden from the user of the ADT.

2. A collection of **operations** for creating and using values of the abstract type. The operations are implemented in terms of the representation type, but their code is hidden from the user of the ADT.

Client-Implementor Model

This suggests a separation between the **client** of the abstraction and the **implementor** of the abstraction.

- The **implementor** knows the representation details. *e.g.*, the compiler-writer knows how integers and functions are represented.

- The **client** knows only the type name and the operations on it. *e.g.*, the programmer does not know how integers and functions are represented.

User-Defined ADT's

Built-in ADT's provide a set of generally-useful, type-safe tools.

- Numeric types.

- Vectors, arrays.

- Function types.

But these mechanisms are **per-language**, rather than **per-program**!

User-Defined ADT's

It is critical to good software development to be able to introduce and enforce **user-defined** levels of abstraction.

- "This value is a file descriptor."

- "This value is a Y2K-compliant date representation."

- "This value is a red-black tree."

User-Defined ADT's

The goal of an abstraction mechanism is to provide the means by which the programmer can

- Introduce **new** abstract types, consisting of a representation type and a set of operations on it.
- Ensure that client code is **insulated** from the representation to the extent chosen by the implementor (not the client!).

Example: Integer Queues

Representation type: `queue`.

Operations:

1. `empty`, the empty queue.
2. `insert`(n,q), taking an integer n and a queue q, yielding a queue with that element inserted at the back.
3. `remove`(q), taking a queue q and returning the front element and a queue with that element removed.

Example: Integer Queues

That's what the client sees. The implementor must provide

1. a definition for the representation of type `queue`;
2. implementations of the operations consistent with that representation.

ADT's as Abstract Machines

Abstract types are closely-related to **API's (Application Programmer Interfaces)** and **Abstract Machines**.

Think of values of the representation type as **states** of an abstract machine.

Think of the operations of the ADT as **instructions** of the abstract machine.

There must be a way to "load" the machine, *i.e.*, to **create** values of the abstract type, and there must be a way to "unload" the machine, *i.e.*, to **retrieve** values out of the representation type.

Defining ADT's in ML

The interaction between the client and implementor is mediated by a **signature** specifying the representation type and its operations.

```
signature INT_QUEUE =
  sig
    type queue
    exception Empty
    val empty : queue
    val insert : int * queue -> queue
    val remove : queue -> int * queue
  end
```


Defining ADT's in ML

The ADT is implemented by a **structure** binding, with an **opaque** signature ascription to hide the representation type.

```
structure IntQueue :> INT_QUEUE =
  struct
    type queue = int list
    exception Empty
    val empty = nil
    fun insert (x, xs) = x::xs
    fun remove nil = raise Empty
      | remove [x] = (x, [])
      | remove (x::xs) =
          let val (y, xs') = remove xs in (y, x::xs') end
  end
```


Defining ADT's in ML

Another implementation of queues, as a pair of lists:

```
structure IntQueue :> INT_QUEUE =
  struct
    type queue = int list * int list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (nil, nil) = raise Empty
      | remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```


Defining "ADT's" in C

It's not really possible. Here's how to approximate it.

The **header file** serves as the interface. It consists of

1. A type definition for the representation type.

2. The prototypes of the operations.

The representation type must be made public in the header file!

(Anyway we can cast to expose the "real" representation of a value.)

Example: Queues in C

```
typedef struct queue {
  int elt;
  struct queue *next;
} QUEUE;

QUEUE *empty();
QUEUE *insert (int x, QUEUE *q);
QUEUE *front (QUEUE *q);
QUEUE *delete (QUEUE *q);
```


Information Hiding: Pro's

Why is it advantageous to hide the representation type?

- The **representation can be changed** without affecting the correctness of user-level code.
- **Safety is ensured** because values of a type cannot be abused in an implementation-dependent manner.
- The implementor can **enforce invariants**.

Information Hiding: Con's

Doesn't data abstraction limit expressiveness?

- It depends on what you mean by expressiveness! What is important is to have the **choice** to hide or expose the representation type. Any **fixed** policy (always hidden, always exposed) is worse than having a choice.
- Expressiveness depends as much on the **strictures** as the **affordances** of a language! The Cole Porter dictum ("anything goes") is a **bad** principle of language design because it deprives you of choice!

Information Hiding: Con's

Doesn't data abstraction limit efficiency?

- At the lowest level, yes — certain "cool hacks" are not possible without knowing the details of the representations.

  But watch out: Apple spent man-years optimizing their 2d graphics code at the lowest level. But their core algorithm was $O(n^2)$, whereas an $O(n \lg n)$ algorithm existed!
- There are many notions of efficiency, *e.g.*, cost of maintenance, cost of modification, ease of team development, that are not measured in instructions per second.

Enforcing Invariants

Using an ADT definition mechanism the implementor can **enforce** his or her own levels of abstraction, called **representation invariants (RI's)**.

Since the representation type is hidden, the client can **only** use the operations of the ADT to manipulate its values.

Consequently, the **only** possible violations of the representation invariant are those that are introduced by the operations of the ADT.

This **dramatically reduces** the points at which errors can arise! If the language supports data abstraction, you can be certain that any RI violation must be in the implementor's code.

Example: Dictionaries

Dictionaries with string keys:

```
signature DICT =
  sig
    type 'a dict
    exception Empty
    val empty : 'a dict
    val lookup : 'a dict * string -> 'a
    val insert : 'a dict * string * 'a -> 'a dict
  end
```


Example: Red-Black Trees

Dictionaries can be implemented with red-black trees.

```
structure Dict :> DICT =
  struct
    datatype 'a dict =
      Empty |
      Red of 'a dict * string * 'a dict |
      Black of 'a dict * string * 'a dict
    ...
  end
```


Example: Red-Black Trees

Representation invariants:

1. Every child of a red node is black. That is, no red node has a red child.

2. The **black height** of the left and the right child of a node is the same. Black height = maximum number of black nodes on any path to a leaf.

If the representation invariant holds, the tree is **balanced** — the height of an n-node tree is $O(\lg n)$. That is, there are no long paths in the tree.

Example: Red-Black Trees

The job of the implementor is to **maintain** the red-black invariants:

1. Each operation may **assume** that the invariant holds on entry. That is, if the input is a value of type `'a dict`, then the rbt invariants may be assumed to hold.

2. Each operation must **ensure** that the invariants hold on exit. That is, if the result is a value of type `'a dict`, it is incumbent on the implementor to ensure that the rbt invariants hold.

Example: Red-Black Trees

From an **external** point of view, dictionaries are always balanced. The client only ever sees well-balanced trees.

From an **internal** point of view, the dictionary may go through intermediate states in which the red-black invariant does not hold.

- A newly-inserted element is colored red.

- The code removes red-red violations while preserving the black height condition.

ADT's as Critical Sections

The implementation of an ADT is a **critical section**.

During execution of the operations, the dictionary may be in a "bad" state.

The operations restore it before anyone notices.

ADT's as Critical Sections

This is just like **locking** in a concurrent program!

- When the lock is released, the protected data is in a good state. **Releasing a lock = asserting a representation invariant!**

- While holding the lock, the data might be in a bad state. The lock prevents anyone else from noticing.

Supporting Team Development

Data abstraction supports **team development** and **code evolution** precisely because it allows program units to be isolated from one another.

One team need not be concerned (and cannot be accidentally concerned!) with what another team is doing.

Experience shows that **integration** bugs are far less problematic in a language with ADT's than in one without.

This goes double for functional languages, because the use of side effects is relatively rare.

Representation Independence

Abstract types enjoy an important property called **representation independence**, which means that the behavior of client code does not change if we change the implementation of an ADT.

For example, the behavior of the client of the `Queue` structure is not affected by whether we use the one-list or two-lists representation of queues.

Why? Because the client code is **polymorphic** in the representation type of the ADT! Representation independence is an instance of **parametricity**!

ADT's and Polymorphism

Opaque signature ascription ensures that the representation type is **hidden** from the client.

For example, the client code of the `Queue` ADT looks like this:

```
... Queue.empty ... Queue.insert (x, q)
... Queue.remove (q') ...
```

These operations are linked by the **abstract** type `Queue.queue`.

ADT's and Polymorphism

Let the client's code be the expression e_{client} with type τ_{client} not involving `Queue.queue`.

The client is, in effect, the **polymorphic** function

```
Fun Queue.queue in                 ;; the representation type
  fn Queue.empty : Queue.queue in
  fn Queue.insert : int * Queue.queue -> Queue.queue in
  fn Queue.remove : Queue.queue -> int * Queue.queue in
     e_client
  end
  ...
end
```

ADT's and Polymorphism

The type of the client code is

$$\forall q.q{\rightarrow}\texttt{int*}q{\rightarrow}q{\rightarrow}q{\rightarrow}\texttt{int*}q{\rightarrow}\tau_{client}$$

(The type `Queue.queue` is renamed to q for clarity.)

By parametricity, the client code must behave uniformly with respect to the choice of representation type for q.

In other words, the client must be **stable** under change of representation.

"New" = Renaming of Bound Variables

In the polymorphic version of the client code, the type `Queue.queue` is a type variable that may be renamed at will.

In particular, the name of the type can be chosen to be **distinct** from all other "active" type variables to ensure that it is "new". **This is what is meant by an abstract type being a "new" type!**

Summary: Advantages of Data Abstraction

A data abstraction mechanism lets you **enforce** invariants in a program.

- Client code cannot violate these invariants.
- Any violations are traceable to a **single location** in the code.

Summary: Advantages of Data Abstraction

A data abstraction mechanism supports **maintenance** and **team development**.

- Client code is insulated from changes to the abstraction.
- Teams can work relatively independently, with little fear of integration nightmares.

Summary: Why Not Use Data Abstraction?

Most popular languages **do not support** enforced data abstraction.

- C, C++, Lisp, Cobol, Fortran do not.
- Java does, via `final` classes.

Summary: Why Not Use Data Abstraction?

Using ADT's requires **careful planning**.

- Must carefully design interfaces. This takes time and effort. Revision is unavoidable.
- Must pay a sizable "up front" cost to design abstractions. Pay-off is not immediate, but the benefits are huge.

Summary: Why Not Use Data Abstraction?

If absolute maximum efficiency is paramount, the cost of abstraction may be too high.

- A good abstraction mechanism will let you selectively expose the representation, so you need not abandon ADT's entirely.
- Do those tiny constant factors really matter?

Most people do not understand these principles; . . . but **you** do!

Programming Languages
Data Abstraction II
Robert Harper
Spring, 1999

Abstract Data Types

An **abstract data type** consists of

1. A **representation type** for the values of the abstract type.

2. One or more **operations** acting on values of the representation type.

Abstract Data Types

For example, the abstract type of (integer) queues consists of

1. A representation of queues as values of some pre-existing type.

2. Operations to insert and remove elements, test for empty, *etc.*

Abstract Data Types

Not every value of the representation type need be a value of the abstract type!

Only values satisfying a given **representation invariant (RI)** are valid!

For example, a set of integers might be represented as a sorted list of elements. Not every integer list is the representation of a set!

Abstract Data Types

To ensure that the RI holds of the ADT, we make a sharp distinction between the **client** and the **implementor** of the ADT.

The **client** must be **polymorphic** in the representation type. The client knows only that there is **some** type supporting the given operations; the client **does not** know the actual representation type.

The **implementor** defines the representation type and defines the operations in terms of the representation. The implementor knows all.

Abstract Data Types

This rigid separation between the client and implementor allows us to **localize responsibility** for maintenance of the RI to the unique spot in the program where the representation type and operations are defined.

The client **cannot** violate the representation invariant, other than by invoking buggy implementations of the operations. In effect, the client "proves" that the RI holds of a value by constructing it out of the given operations.

The implementor must ensure that the operations **preserve** the RI by assuming it to hold on input, and ensuring that it holds on output.

ADT's in ML

An abstract data type is implemented by a **structure** that defines the representation type and the implementation of the operations.

The separation between the client and the implementor is achieved by **opaque ascription** of a **signature** to the implementing structure. The signature specifies precisely what information is to be propagated to the client; it must be a valid signature for the implementation.

In contrast **transparent ascription** does **not** provide data abstraction — the representation type is "leaked" to the client, in violation of the intended separation.

Example: Queues in ML

```
signature QUEUE =
  sig
    type queue
    exception Empty
    val empty : queue
    val insert : int * queue -> queue
    val remove : queue -> int * queue
  end
```


Example: Queues in ML

```
structure QueueR :> QUEUE =
  struct
    type queue = int list
    exception Empty
    val empty = nil
    fun insert (x, xs) = x::xs
    fun remove nil = raise Empty
      | remove xs =
        let val (x::rxs) = rev xs
         in (x, rev rxs) end
  end
```

Example: Queues in ML

Opaque ascription ensures that the client "knows" only what is in the signature!

1. There exists **some** type `QueueR.queue` (but it is not known what type this is).

2. There is an operation `insert` of type `int * QueueR.queue -> QueueR.queue`, *etc.*

The client code is **polymorphic** in `QueueR.queue` — it's an implicit type parameter to the client code.

Change of Representation

An important advantage of abstract types is that the implementor may change the representation without affecting the behavior of the clients. This is called **representation independence**.

The implementor can choose a **new** representation type or a **new** implementation of the operations, or both.

This change will not affect the clients, **provided that** the new and the old implementation are suitably related.

Change of Representation

But what do we mean by "suitably related"? Under what conditions can the implementation be changed without affecting the client?

Clearly some changes can lead to changes in client behavior. For example, we might mis-implement a queue as a stack (it's been known to happen ...).

What sorts of changes are guaranteed to preserve client behavior?

Change of Representation

It is helpful to recall the analogy between ADT's and Abstract Machines.

- The representation type is the state of the AM.

- The operations are instructions that move from state to state.

- There are operations to "load" and "unload" the machine.

Change of Representation

When are two machines equivalent? When both yield the same observable behavior for a given start state, regardless of how they may differ on intermediate states.

How can we show that two machines are equivalent in this sense?

Change of Representation

Consider the following alternative representation of queues:

```
stucture QueueF :> QUEUE
  struct
    type queue = int list
    exception Empty
    val empty = nil
    fun insert (x, xs) =
        let val rxs = rev xs
         in rev (x::rxs) end
    fun remove nil = raise Empty
      | remove (x::xs) = (x, xs)
  end
```


Change of Representation

We've defined two abstract machines, the `QR` machine and the `QF` machine.

The `QR` machine (defined earlier) enqueues on the **rear** of the list. The `QF` machine (just defined) enqueues on the **front** of the list.

Clearly, the two machines are equivalent in the sense that if we start with the empty queue, and perform any sequence of `insert`'s and `remove`'s, we'll get the same observable behavior from both.

Change of Representation

For example, consider the following sequence of instructions: `E`, `I 1`, `I 2`, `R`, `I 3`, `R`, `R`.

Regardless of whether we interpret these as instructions for the `QF` or `QR` machines, we should see the same sequence of dequeued elements.

In the `QR` machine the values of the representation type corresponding to these instructions is: `[]`, `[1]`, `[2,1]`, `[2]`, `[3,2]`, `[3]`, `[]`.

In the `QF` machine, the sequence is: `[]`, `[1]`, `[1,2]`, `[2]`, `[2,3]`, `[3]`, `[]`.

In both cases the dequeued elements are `1`, `2`, `3`.

Proving Equivalence of ADT Implementations

How can we prove that `QueueF` and `QueueR` are equivalent in the sense that the engender the same behavior in all clients?

First, we exhibit a **simulation relation** between the types `QueueR.queue` and `QueueF.queue` that is **preserved** by the operations.

Second, we appeal to Reynolds **parametricity theorem** to conclude that the existence of a simulation relation is enough to ensure that clients cannot observe any difference.

Simulation Relations

A **simulation relation** between `QueueF` and `QueueR` consists of

1. A binary relation $\sim$ between values of type `QueueF.queue` and values of type `QueueR.queue`.

2. A proof that each operation of the ADT **preserves** the simulation relation — given two related implementation values, applying corresponding operations to these values gives related results.

Simulation Relations

Important: the simulation relation is the identity at **all** types **except** for the ADT under consideration.

That way we can ensure that the observable outputs of the two ADT's are the same, while allowing the "internal states" to be wildly different (but related).

Simulation Relations

For example, here is a simulation relation between `QueueR` and `QueueF`:

$$rq \sim fq \quad \textit{iff} \quad rq = \texttt{rev}(fq)$$

Here is an informal proof that each of the operations preserves this relation.

Simulation Relations

The values `QueueF.empty` and `QueueR.empty` are both `nil`. These are clearly related by the simulation relation.

Suppose that $rq \sim fq$ and that we insert the same element into both, using their respective operations. We are to show that

$$\mathtt{QueueR.insert}(n, rq) \sim \mathtt{QueueF.insert}(n, fq)$$

Simulation Relations

Consulting the definitions, we deduce that

$$\mathtt{QueueR.insert}(n, rq) = n\texttt{::}rq$$

and that

$$\mathtt{QueueF.insert}(n, fq) = rev(n\texttt{::}rev(fq))$$

Using the definition of `rev` and simulation assumption,

$$rev(rev(n\texttt{::}rev(fq))) = n\texttt{::}rev(fq) = n\texttt{::}rq,$$

as desired.

(The `remove` operation is handled similarly.)

ADT Correctness

OK, so we've exhibited a simulation relation between `QueueR` and `QueueF`. So what?

Reynolds' Parametricity Theorem tells us that the observable behavior (eventual outcome) of any client is the same if we execute it using either of two related ADT implementations.

Why is this so? Intuitively, it is because the client is **polymorphic** in the representation type. By Parametricity, it must be **uniform** with respect to that type.

ADT Correctness

Reynolds' Theorem makes precise what we mean by uniformity through the notion of simulation relations — if you apply a polymorphic function to two related types, the result is related.

Since the relations are the identity everywhere except on the ADT, the observable behavior of the client must be same!

In other words: the basis of data abstraction is (parametric) polymorphism!

Equality of Expressions

The "proof" of equivalence of `QueueR` and `QueueF` relies on

1. Reynolds' Theorem, which lies beyond our reach.

2. Equality of expressions, which we will discuss in more detail.

When are two expressions equivalent? What do we mean by equality of expressions?

Equality of Expressions

Leibniz's Principle: two things are equal iff you can't tell them apart.

The right-to-left direction is the one with content. But how do we tell things apart?

Given two black boxes, we can try to tell them apart by performing **experiments** on them and seeing whether they behave differently.

If **some** experiment turns out differently, they're different. if **all** experiments turn out the same, they're the same.

Equality of Expressions

Applying Leibniz's Principle to MinML, two expressions (perhaps with free variables) are equal iff we cannot tell them apart.

But what constitutes an experiment on the two expressions? What criteria should we use to tell them apart?

Morris's Principle: two expressions are equivalent iff whenever we use one in a complete program, using the other instead will not change the outcome.

Important: complete programs are closed expressions of **finitely observable** type. For us, this means `int` or `bool`; functions are not finitely observable.

Equality of Expressions

For example, are x+y and y+x equal?

Well, consider **any** complete program (say of integer type)

$$\ldots\ x\text{+}y\ \ldots$$

Run it, and find out whether it diverges or returns a number n.

Equality of Expressions

Now replace x+y by y+x:

. . . y+x . . .

Run it, and see whether it diverges or returns a number m.

If both diverge, or $m = n$, then the experiment succeeds; otherwise it fails.

Equality of Expressions

The fly in the ointment is that we must consider **all possible** programs that use the two expressions to be assessed for equality.

This can be a pretty onerous undertaking. In fact, it's very, very difficult in general to prove that two expressions are equal in this sense.

To avoid this, many sophisticated methods have been developed. We'll outline a few valid principles of reasoning, without proving that they are valid. (The proofs are, in most cases, very difficult.)

Observational Equivalence

The relation $\Gamma \vdash e_1 \cong e_2 : \tau$, where $\Gamma \vdash e_i : \tau$ $(i = 1, 2)$, means that e_1 and e_2 are interchangeable in all complete programs.

It enjoys a number of important properties, which we outline here. Most of these properties generalize to ML, **provided** that we stick with the purely functional subset (no `ref` types).

Checking equivalence of expressions with `ref` cells is much more difficult and subtle. Moral: stick with FP whenever possible!

General Laws of Equivalence

Equivalence is an equivalence relation:

$$\frac{}{\Gamma \vdash e \cong e : \tau}$$

$$\frac{\Gamma \vdash e_2 \cong e_1 : \tau}{\Gamma \vdash e_1 \cong e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 \cong e_2 : \tau \qquad \Gamma \vdash e_2 \cong e_3 : \tau}{\Gamma \vdash e_1 \cong e_3 : \tau}$$

General Laws of Equivalence

We may substitute equals by equals, obtaining equals.

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1 \quad \cdots \quad \Gamma \vdash e_n \cong e_n' : \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \cong o(e_1', \ldots, e_n') : \tau}$$

$$\frac{\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e \cong e' : \tau_2}{\Gamma \vdash \texttt{fun}\, f\ (x{:}\tau_1){:}\tau_2\ \texttt{is}\, e\, \texttt{end} \cong \texttt{fun}\, f\ (x{:}\tau_1){:}\tau_2\ \texttt{is}\, e'\, \texttt{end} : \tau_1 \to \tau_2}$$

General Laws of Equivalence

Equivalence is stable under substitution of **values** (of appropriate type).

$$\frac{\Gamma[x{:}\tau] \vdash e \cong e' : \tau'}{\Gamma \vdash [v/x]e \cong [v/x]e' : \tau'}$$

This rule is **not true** in general for expressions that are not values!

Symbolic Evaluation

Primitive operations:

$$\frac{}{\Gamma \vdash o(v_1, \ldots, v_n) \cong v : \tau,}$$

Conditionals:

$$\frac{}{\Gamma \vdash \texttt{if}\, \texttt{true}\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \cong e_1 : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{if}\, \texttt{false}\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \cong e_2 : \tau}$$

Symbolic Evaluation

Function calls:

$$\frac{}{\Gamma \vdash v(v_1) \cong [v, v_1/f, x]e : \tau_2}$$

Important: note that v and v_1 might have free variables! We regard variables as values since they can only ever be bound to values in a call-by-value language.

Extensionality

Function expressions really behave like functions:

$$\frac{\Gamma[x{:}\tau_1] \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \rightarrow \tau_2}$$

Strictness Properties

Let Ω_τ be a divergent (looping) expression of type τ. If any expression is equivalent to Ω, then it diverges. Using this we can express evaluation order by equations.

$$\frac{}{\Gamma \vdash o(e_1, \ldots, e_{i-1}, \Omega_{\tau_i}, e_{i+1}, \ldots, e_n) \cong \Omega_\tau : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{if}\, \Omega_{\texttt{bool}}\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \cong \Omega_\tau : \tau}$$

Strictness Properties

$$\frac{}{\Gamma \vdash \Omega_{\tau_2 \rightarrow \tau}(e_2) \cong \Omega_\tau : \tau}$$

$$\frac{}{\Gamma \vdash e_1(\Omega_{\tau_2}) \cong \Omega_\tau : \tau}$$

Arithmetic Laws

Expected mathematical properties of operations on integers hold.

$$\frac{}{\Gamma \vdash e_1\texttt{+}e_2 \cong e_2\texttt{+}e_1 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash e_1\texttt{+}(e_2\texttt{+}e_3) \cong (e_1\texttt{+}e_2)\texttt{+}e_3 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash e_1\texttt{=}e_2 \cong e_2\texttt{=}e_1 : \texttt{bool}}$$

List Properies

If we extend MinML with lists, we obtain a number of new equations. (See notes for details.)

$$\frac{}{\Gamma \vdash \texttt{listcase}\,\texttt{nil}\,\texttt{of}\,\texttt{nil} \texttt{=>} e' \mid x\texttt{::}y \texttt{=>} e''\,\texttt{end} \cong e' : \tau'}$$

$$\frac{}{\Gamma \vdash \texttt{listcase}\,v_1\texttt{::}v_2\,\texttt{of}\,\texttt{nil} \texttt{=>} e' \mid x_1\texttt{::}x_2 \texttt{=>} e''\,\texttt{end} \cong [v_1, v_2/x_1, x_2]e'' : \tau'}$$

List Induction

To prove that

$$\Gamma[x{:}\tau\,\texttt{list}] \vdash e_1 \cong e_2 : \tau',$$

it suffices to show the following two facts:

1. $\Gamma \vdash [\texttt{nil}/x]e_1 \cong [\texttt{nil}/x]e_2 : \tau'$

2. For every $\Gamma \vdash v_h : \tau$ and $\Gamma \vdash v_t : \tau\,\texttt{list}$, if $\Gamma \vdash [v_t/x]e_1 \cong [v_t/x]e_2 : \tau'$, then $\Gamma \vdash [v_h\texttt{::}v_t/x]e_1 \cong [v_h\texttt{::}v_t/x]e_2 : \tau'$.

Prop $x{:}\tau\,\texttt{list}, y{:}\tau\,\texttt{list} \vdash \texttt{rev}(\texttt{app}(x, y)) \equiv \texttt{app}(\texttt{rev}(y))\texttt{rev}(x) : \tau\,\texttt{list}$

Proof By list induction on x.

Prop $x{:}\tau\,\texttt{list} \vdash \texttt{rev}(\texttt{rev}(x)) \equiv x : \tau\,\texttt{list}$

Proof by list induction on x, making use of the preceding result.

Summary

Representation Independence = stability of clients under change of implementation to a related implementation.

Simulation relation = binary relation on values that is preserved by the operations of the ADT.

Parametricity Theorem = simulation relation suffices to ensure stability of clients.

Leibniz's Principle = Morris's Principle = expressions are equal if you can't tell them apart.