Impure Features and Computational Effects

Most practical programming languages include impure features, also known as **computational effects**, that extend beyond yielding simple results. These effects include:

- Assignments to mutable variables
- Input and output operations
- Non-local transfers of control via exceptions
- Inter-process synchronization and communication

This chapter focuses on adding **mutable references** to the calculi, dealing explicitly with a **store (or heap)**.

Introduction to References

Nearly every programming language provides an assignment operation that changes the contents of a previously allocated piece of storage. Languages like ML keep name binding and assignment separate.

- `x` can be a variable with a value (e.g., 5).
- `y` can be a variable with a reference to a mutable cell (a pointer) whose content is 5.

In C-like languages, every variable name refers to a mutable cell with implicit dereferencing.

Basics of References

The fundamental operations on references include:

- **Allocation:** Creating a new reference using the `ref` operator.
  - Example: `r = ref 5; r : Ref Nat`
- **Dereferencing:** Reading the current value of a cell using the `!` operator.
  - Example: `!r; 5 : Nat`
- **Assignment:** Changing the value stored in a cell using the `:=` operator.
  - Example: `r := 7; unit : Unit`

Side Effects and Sequencing

The fact that the result of an assignment expression is the trivial `unit` value fits nicely with the sequencing notation.

- Example: `(r:=succ(!r); !r); 8 : Nat` instead of the more cumbersome, `(!r) (r := succ(!r)); 9 : Nat`

Restricting the type of the first expression to `Unit` allows the typechecker to catch errors.

References and Aliasing

It's crucial to distinguish between the reference bound to `r` and the cell in the store pointed to by this reference. If `s = r; s : Ref Nat`, copying `r` only copies the reference, not the cell. `r` and `s` become **aliases** for the same cell.

Evaluating the expression `a = {ref 0, ref 0}` and `b = (λx:Ref Nat. {x,x}) (ref 0)` will have different effects. In the first case, two different reference cells are created. In the second case, only one reference cell is created, and both fields of the record point to the same cell.

Diagram illustrating references and aliasing:

The references r and s are aliases for the same cell.

Shared State

Aliasing can make programs with references complex. However, it also enables **shared state** between different program parts. Consider the following example:

```
c = ref 0;
c : Ref Nat
incc = λx:Unit. (c := succ (!c); !c);
incc : Unit → Nat
decc = λx:Unit. (c := pred (!c); !c);
decc : Unit → Nat
o = {i = incc, d = decc};
o : {i:Unit→Nat, d:Unit→Nat}
```

Calling `incc unit` results in changes to `c` that `decc` can observe. This creates a simple form of an object with shared state.

References to Compound Types

Reference cells can hold values of any type, including functions. This can be used to implement data structures like arrays.

```
NatArray = Ref (Nat→Nat);
newarray = ref (λn:Nat.0);
newarray : Unit → NatArray
lookup = λa:NatArray. λn:Nat. (!a) n;
lookup : NatArray → Nat → Nat
update = λa:NatArray. λm:Nat. λv:Nat. let oldf = !a in a := (λn:Nat. if equal
update : NatArray → Nat → Nat → Unit
```

A compact definition of update like `update = λa:NatArray. λm:Nat. λv:Nat. a := (λn:Nat. if equal m n then v else (!a) n);` would not behave the same because `(!a)` is evaluated each time `n` is not equal to `m`, whereas the original version stores the original function (!a) in `oldf` and calls that.

References to values containing other references are useful for defining mutable lists and trees.

Garbage Collection

Storage deallocation is crucial. This design relies on garbage collection to collect and reuse unreachable cells. Explicit deallocation can lead to type safety issues like dangling references.

Typing

The typing rules for `ref`, `:=`, and `!` are straightforward.

```
Γ |- t1 : T1
----------------
Γ |- ref t1 : Ref T1
```

```
Γ |- t1 : Ref T1
----------------
Γ |- !t1 : T1
```

```
Γ |- t1 : Ref T1  Γ |- t2 : T1
-----------------------
Γ |- t1:=t2 : Unit
```

Evaluation

The operational semantics needs to account for the store. The evaluation rules now take a store as an argument and return a new store: `t | μ → t' | μ'`.

Values and Terms

The syntax is extended to include locations in the set of values:

```
v ::= λx:T.t | unit | l
t ::= x | λx:T.t | t t | unit | ref t | !t | t:=t | l
```

Evaluation Rules for Dereferencing

```
t1 | μ → t1' | μ'
---------------------
!t1 | μ → !t1' | μ'
```

```
μ(l) = v
-------------
!l | μ → v | μ
```

Evaluation Rules for Assignment

```
t1 | μ → t1' | μ'
---------------------
t1:=t2 | μ → t1':=t2 | μ'
```

```
t2 | μ → t2' | μ'
---------------------
v1:=t2 | μ → v1:=t2' | μ'
```

```
l:=v2 | μ → unit | [l -> v2]μ
```

Evaluation Rules for References

```
t1 | μ → t1' | μ'
---------------------
ref t1 | μ → ref t1' | μ'
```

```
l ∉ dom(μ)
---------------------
ref v1 | μ → l | (μ, l -> v1)
```

The evaluation rules don't include garbage collection, allowing the store to grow without bound.

Store Typings

The type of a location depends on the store's contents. A **store typing** (Σ) maps locations to types.

```
Σ(l) = T1
-----------
Γ | Σ |- l : Ref T1
```

The typing rules are parameterized on a store typing rather than a concrete store.

```
Γ | Σ |- t1 : T1
------------------
Γ | Σ |- ref t1 : Ref T1
```

```
Γ | Σ |- t1 : Ref T11
--------------------
Γ | Σ |- !t1 : T11
```

```
Γ | Σ |- t1 : Ref T11   Γ | Σ |- t2 : T11
-------------------------
Γ | Σ |- t1:=t2 : Unit
```

Safety

Standard type safety properties hold for the calculus with references. The progress theorem remains mostly unchanged.

Well-Typed Stores

A store μ is considered well-typed with respect to a typing context Γ and a store typing Σ, denoted as $\Gamma|\Sigma\mu$, if two conditions are met:

1. The domain of μ is equal to the domain of Σ (dom(μ) = dom(Σ)).
2. For every location l in the domain of μ, the value in the store at location l, $\mu(l)$, has the type predicted by the store typing at that location, $\Sigma(l)$. Formally, $\Gamma|\Sigma\mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$.

   In simpler terms, a store μ is consistent with a store typing Σ if every value in the store has the type predicted by the store typing.

Exercise

Find a context Γ, a store μ, and two different store typings $\Sigma 1$ and $\Sigma 2$ such that both $\Gamma|\Sigma 1\mu$ and $\Gamma|\Sigma 2\mu$ hold true.

Preservation Property

If $\Gamma|\Sigma t : T$ and $\Gamma|\Sigma\mu$, and if evaluating term t in store μ results in term t' and store μ' (i.e., $t|\mu \to t'|\mu'$), then $\Gamma|\Sigma t' : T$. (Less wrong.)

This statement holds for all evaluation rules except the allocation rule E-RefV, because this rule extends the store's domain.

Extending the Store Typing

Since the store can grow during evaluation, we must allow the store typing to grow as well. This leads to the final statement of the type preservation property.

Theorem: Preservation

If $\Gamma|\Sigma t : T$ and $\Gamma|\Sigma\mu$, and if evaluating term t in store μ results in term t' and store μ' (i.e., $t|\mu \to t'|\mu'$), then for some $\Sigma' \supseteq \Sigma$, $\Gamma|\Sigma' t' : T$ and $\Gamma|\Sigma'\mu'$.

The preservation theorem asserts the existence of a store typing $\Sigma' \supseteq \Sigma$ such that the new term t' is well-typed with respect to Σ'. Σ' either equals Σ or is $(\mu', l : T_1)$, where l is a newly allocated location and T_1 is the type of the initial value bound to l in the extended store $(\mu', l : v_1)$.

Lemmas for Preservation Proof

1. **Substitution Lemma:** If $\Gamma, x : S|\Sigma t : T$ and $\Gamma|\Sigma s : S$, then $\Gamma|\Sigma[x \mapsto s]t : T$.
2. If $\Gamma|\Sigma\mu$ and $\Sigma(l) = T$ and $\Gamma|\Sigma v : T$, then $\Gamma|\Sigma[l \mapsto v]\mu$.
3. **Weakening Lemma for Stores:** If $\Gamma|\Sigma t : T$ and $\Sigma' \supseteq \Sigma$, then $\Gamma|\Sigma' t : T$.

Progress Theorem

Theorem: Progress

Suppose t is a closed, well-typed term (that is, $\emptyset|\Sigma t : T$ for some T and Σ). Then either t is a value, or else, for any store μ such that $\emptyset|\Sigma\mu$, there is some term t' and store μ' with $t|\mu \to t'|\mu'$.

Exercise

Is the evaluation relation normalizing on well-typed terms? If so, prove it. If not, write a well-typed factorial function in the present calculus (extended with numbers and booleans).

Notes

The content of this chapter is adapted from Harper (1994, 1996) and Wright and Felleisen (1994).

Aliasing

Static prediction of possible aliasing is a long-standing problem in compiler implementation (alias analysis) and programming language theory. Reynolds (1978, 1989) coined the term "syntactic control of interference."