

References and Exceptions

Types and Programming Languages, Spring 2005, SoE UCSC

PRESENTER Mark Storer
SCRIBE Aaron Tomb

April 22, 2005

1 Imperative vs. Functional Programming

We began with a discussion of the merits of both the imperative and functional styles of programming. Some of the characteristics of each are as follows.

Imperative

- Allows aliasing between variables. That is, multiple variables can refer to the same object.
- May make it easier to write efficient programs.
- May be more natural or intuitive. However, this may also be due to the fact that most computer science curricula begin with discussions of imperative programming.
- Intuitive nature may also be due to people's experience with the changing state of the world around them.
- May make programming easier, but verification and confidence in correctness more difficult.
- Handles I/O easily, and reactive programming styles somewhat easily.

Functional

- Does not allow aliasing. Programs typically are referentially transparent.
- Easier to reason about, because “stuff never changes out from under you”.
- More mathematically abstract.
- More declarative. Not purely declarative, however — still has an “animator”.

- Requires the use of a garbage collector.
- Can't handle a reactive programming style.
- Handles I/O “badly, if at all”.

Our tentative conclusion was that purely functional programming was of limited usefulness. However, the use of a functional style where possible leads to programs that are easier to reason about, and imperative constructs are best used sparingly.

2 Formal Description of References

Given that we want these imperative constructs, how do we add them to the language? One possibility is as follows.

Syntax

We discussed a syntax consisting of that used in the simply typed λ -calculus, with the following extensions:

Terms t	$::=$	\dots	(existing terms)
		$!t$	(dereferencing)
		$t := t$	(assignment)
		$ref\ t$	(allocation)
		l	(locations)
Values v	$::=$	\dots	(existing values)
		l	(locations)
Types T	$::=$	\dots	(existing types)
		$Ref\ T$	(reference types)

Operational Semantics

The constructs present in the simply typed λ -calculus evaluate as before, but carry a store argument (μ) along with them. For instance, the application rule becomes:

$$\frac{t_1|\mu \rightarrow t'_1|\mu'}{t_1\ t_2|\mu \rightarrow t'_1\ t_2|\mu'}$$

and the other rules change similarly.

We discussed the following small-step semantics for the new constructs:

$$\frac{\mu' = \mu[l := v] \quad l \notin Dom(\mu)}{ref\ v|\mu \rightarrow l|\mu'} \qquad \frac{t|\mu \rightarrow t'|\mu'}{ref\ t|\mu \rightarrow ref\ t'|\mu'}$$

$$\begin{array}{c}
\frac{t|\mu \rightarrow t'|\mu'}{!t|\mu \rightarrow !t'|\mu'} \\
\frac{t_1|\mu \rightarrow t'_1|\mu'}{t_1 := t_2|\mu \rightarrow t'_1 := t_2|\mu'} \\
\frac{t_2|\mu \rightarrow t'_2|\mu'}{l := t_2|\mu \rightarrow l := t'_2|\mu'} \\
\frac{\mu' = [l \mapsto v]\mu}{l := v|\mu \rightarrow \mathit{unit}|\mu'}
\end{array}$$

Type Rules

Finally, we added the following type rules to those already present in the simply typed λ -calculus.

$$\begin{array}{c}
\frac{\Sigma(l) = T_1}{\Gamma|\Sigma \vdash l : Ref\ T_1} \\
\frac{\Gamma|\Sigma \vdash t_1 : Ref\ T \quad \Gamma|\Sigma \vdash t_2 : T}{\Gamma|\Sigma \vdash t_1 := t_2 : Unit} \\
\frac{\Gamma|\Sigma \vdash t : Ref\ T}{\Gamma|\Sigma \vdash !t : T}
\end{array}$$

3 Miscellaneous Notes on References

We discussed the various type and value mappings present in our language description. Thus far, we have Γ , which maps from variables to types; Σ , which maps locations to values; and μ , which maps locations to types. What about mapping variables to values? It turns out we don't need to explicitly map variables to values in our language, as variable values are explicitly passed around through substitution.

We also discussed the possibility of doing recursive computation without a fixpoint construct. Given the simply typed λ -calculus, extended with references (as well as conditionals, arithmetic operations, and `let` expressions for convenience), we can write the factorial function as follows (where `id` is the identity function):

$$\mathit{let\ fact} = \mathit{ref\ id\ in\ fact} := \lambda n. \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * (!\mathit{fact})(n - 1)$$

4 Safety of References

Can we now show that the language described so far retains the safety properties present in the simply typed λ -calculus? We discussed both progress and preservation, but the proof of progress is a trivial extension of that for the original language, so we focused on preservation.

One possible statement of the preservation property could proceed as follows: If $\Gamma|\Sigma \vdash t : T$ and $t|\mu \rightarrow t'|\mu'$ then $\Gamma|\Sigma \vdash t' : T$.

This doesn't work, however. If $\Gamma = \emptyset$, $\Sigma = \emptyset$, and $t = \mathit{ref}0$, then $t' = l$. But l can't be typed when $\Sigma = \emptyset$. We need to ensure that when μ gets bigger, Σ gets bigger, as well.

Another problem also arises. Consider the case where $t = 1 + !l$, $\Sigma(l) = Nat$ and $\mu(l) = unit$. Then t has a valid type, but $t' = 1 + unit$, which has no valid type. So, we also need to ensure that μ and Σ are actually consistent with each other.

We can solve both of these problems by introducing the concept of a well typed store, and using it to define preservation.

Definition 4.1 (Well-Typed Store). *A store μ is said to be well typed with respect to a typing context Γ and a store typing Σ , written $\Gamma|\Sigma \vdash \mu$, if $Dom(\mu) = Dom(\Sigma)$ and $\Gamma|\Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in Dom(\mu)$.*

Preservation can now be taken to mean that, if

$$\begin{array}{l} \Gamma|\Sigma \vdash t : T \\ \Gamma|\Sigma \vdash \mu \\ t|\mu \rightarrow t'|\mu' \end{array}$$

then for some $\Sigma' \supseteq \Sigma$,

$$\begin{array}{l} \Gamma|\Sigma' \vdash t' : T \\ \Gamma|\Sigma' \vdash \mu' \end{array}$$

One interesting property of this theorem is that the definition of a well typed store includes Γ in the judgement $\Gamma|\Sigma \vdash \mu$. This suggests that free variables may occur as parts of values residing in the store. This can, in fact, happen in our language because λ abstractions are values, and can this exist in the store, possibly with free variables in their bodies.

In many languages, “code” cannot be stored in this way, so such a judgement would not need to refer to a typing context.

5 Formal Description of Exceptions

Exceptions allow programs to perform a restricted form of non-local control flow when errors occur.

Syntax

The simple exceptions we discussed only affect the syntax of terms. The set of values and types remains the same.

$$\begin{array}{lll} \text{Terms } t & ::= & \dots \quad (\text{existing terms}) \\ & | & \text{error} \quad (\text{error term}) \\ & | & \text{try } t \text{ with } t \quad (\text{error handlers}) \end{array}$$

Operational Semantics

The added terms mean that we need a few additional semantic rules, on top of those present in the simply typed λ -calculus.

$$\frac{}{error\ t_2 \rightarrow error} \quad \frac{}{v_1\ error \rightarrow error} \quad \frac{t_1 \rightarrow t'_1}{try\ t_1\ with\ t_2 \rightarrow try\ t'_1\ with\ t_2} \quad \frac{}{try\ v_1\ with\ t_2 \rightarrow v_1}$$

Type Rules

Finally, we need to augment these new terms with a set of typing rules.

$$\frac{}{\Gamma \vdash error : T} \quad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash try\ t_1\ with\ t_2 : T}$$

6 Another Type System for Exceptions

An interesting feature of the presentation of exceptions given above is that it changes the necessary formulation of the progress property. Now, well-typed terms can either reduce to a value, or to *error*.

One possible modification would be to change the progress theorem to say that every well-typed term is either a value, or *error*, or can be reduced to another term.

If we don't want to change our progress theorem, however, then the following type rules might suffice.

$$\frac{\Gamma \vdash t_1 : T, e_1 \quad \Gamma \vdash t_2 : T, e_2}{\Gamma \vdash try\ t_1\ with\ t_2 : T, e_2} \quad \frac{\Gamma \vdash t_1 : T_1 \xrightarrow{e_1} T_2, e_3 \quad \Gamma \vdash t_2 : T_1, e_2}{\Gamma \vdash t_1\ t_2 : T_2, (e_1 \vee e_2 \vee e_3)}$$

$$\frac{\Gamma, x : T \vdash t : T_2, e}{\Gamma \vdash \lambda x : T_1. t : T_1 \xrightarrow{e} T_2, false} \quad \frac{}{\Gamma \vdash error : T, true} \quad \frac{x : T \in Gamma}{\Gamma \vdash x : T, false}$$

In these rules, every type has two parts: a basic type, as we had before, and an error bit, which indicates whether the term can possibly reduce to *error*. Then, if we define a well-typed term t as one satisfying the judgement $\Gamma \vdash t : T, false$, we can leave the progress theorem as originally stated.

This will mean that any program reducible to error is no longer well-typed. We can allow any previously-valid program, however, by simply wrapping it in a *try...with* construct.

As an interesting addition, we can use this system to type natural numbers a couple of basic arithmetic operations as follows:

$$\frac{}{\Gamma \vdash n : Nat, false} \quad \frac{\Gamma \vdash t_1 : Nat, e_1 \quad \Gamma \vdash t_1 : Nat, e_2}{\Gamma \vdash t_1 + t_2 : Nat, (e_1 \vee e_2)} \quad \frac{\Gamma \vdash t_1 : Nat, e_1 \quad \Gamma \vdash t_1 : Nat, e_2}{\Gamma \vdash t_1/t_2 : Nat, true}$$

7 Exceptions Carrying Values

In many situations, the simple result *error* doesn't tell us as much about what went wrong as we would like. To address this problem, we can amend the description of an error to require it to carry an additional parameter, which (presumably) describes the type of error that occurred.

We can begin to address this problem by amending the language with the following additional term and type:

$$\begin{array}{ll} \text{Terms } t & ::= \dots \quad (\text{existing terms}) \\ & | \text{ raise } t \quad (\text{value-carrying error}) \\ \text{Types } T & ::= \dots \quad (\text{existing types}) \\ & | T_{error} \quad (\text{the type of error values}) \end{array}$$

and these type rules:

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{error} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \quad \frac{\Gamma \vdash t : T_{error}}{\Gamma \vdash \text{raise } t : T}$$

In this system, the result of an erroneous execution will change from the simple *error* term to the more informative *raise t*, where *t* can be any term of type T_{error} . The T_{error} type could be Nat , to allow for simple numeric error codes, or something more flexible, like a variant type, to allow for more complex error results.