Supplementary Notes on Aggregate Data Structures

15-312: Foundations of Programming Languages
Frank Pfenning
modified by Jonathan Aldrich

Lecture 7
Sep 16, 2003

Now we come to various language extensions which make MinML a more realistic language without changing its basic character.

Products. Introducing products just means adding pairs and a unit element to the language [Ch. 19.1]. We could also directly add n-ary products, but we will instead discuss records later when we talk about object-oriented programming. MinML is a call-by-value language. For consistency with the basic choice, the pair constructor also evaluates its arguments—otherwise we would be dealing with *lazy pairs*.[1] In addition to the `pair` constructor, we can extract the first and second component of a pair.[2]

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1, e_2) : \texttt{cross}(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \texttt{cross}(\tau_1, \tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \texttt{cross}(\tau_1, \tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2}$$

For the unit type we only have a constructor but no destructor, since there are no components to extract.

$$\frac{}{\Gamma \vdash \texttt{unitel} : \texttt{unit}}$$

[1]See Assignment 3

[2]An alternative treatment is given in [Ch. 19.1], where the destructor provides access to both components of a pair simultaneously. Also, the unit type comes with a corresponding `check` construct.

We often adopt a more mathematical notation according to the table at the end of these notes. However, it is important to remember that the mathematical shorthand is just that: it is just a different way to shorten higher-order abstract syntax or make it easier to read.

A pair is a value if both components are values. If not, we can use the search rules to reduce, using a left-to-right order. Finally, the reduction rules extract the corresponding component of a pair.

$$\frac{e_1 \ \mathsf{value} \quad e_2 \ \mathsf{value}}{\mathtt{pair}(e_1, e_2) \ \mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1, e_2) \mapsto \mathtt{pair}(e_1', e_2)} \quad \frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathtt{pair}(v_1, e_2) \mapsto \mathtt{pair}(v_1, e_2')}$$

$$\frac{e \mapsto e'}{\mathtt{fst}(e) \mapsto \mathtt{fst}(e')} \quad \frac{e \mapsto e'}{\mathtt{snd}(e) \mapsto \mathtt{snd}(e')}$$

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value}}{\mathtt{fst}(\mathtt{pair}(v_1, v_2)) \mapsto v_1} \quad \frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value}}{\mathtt{snd}(\mathtt{pair}(v_1, v_2)) \mapsto v_2}$$

Since it is at the core of the progress property, we make the value inversion property explicit.

> If $\cdot \vdash v : \mathtt{cross}(\tau_1, \tau_2)$ and v value then $v = \mathtt{pair}(v_1, v_2)$ for some v_1 value and v_2 value.

Unit Type. The unit types does not yield any new search or reduction rules, only a new value. At first it may not seem very useful, but we will see an application when we add references to the language.

$$\frac{}{\mathtt{unitel} \ \mathsf{value}}$$

The value inversion property is also simple.

> If $\cdot \vdash v : \mathtt{unit}$ then $v = \langle\,\rangle$.

Sums. Unions, as one might now them from the C programming language, are inherently not type safe. They can be abused in order to access the underlying representations of data structures and intentionally violate any kind of abstraction that might be provided by the language. Consider, for example, the following snippet from C.

```
union {
   float f;
   int   i;
} unsafe;

unsafe.f = 5.67e-5;
printf("%d", unsafe.i);
```

Here we set the member of the union as a floating point number and then print the underlying bit pattern as if it represented an integer. Of course, much more egregious examples can be imagined here.

In a type-safe language we replace unions by disjoint sums. In the implementation, the members of a disjoint sum type are tagged with their origin so we can safely distinguish the cases. In order for every expression to have a unique type, we also need to index the corresponding injection operator with their target type.[3]

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{inl}(\tau_1, \tau_2, e_1) : \mathtt{sum}(\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{inr}(\tau_1, \tau_2, e_2) : \mathtt{sum}(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}(\tau_1, \tau_2) \quad \Gamma, x_1{:}\tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2{:}\tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathtt{case}(e, x_1.e_1, x_2.e_2) : \sigma}$$

Note that we require both branches of a `case`-expression to have the same type σ, just as for a conditional, because we cannot be sure at type-checking time which branch will be taken.

$$\frac{e_1 \ \mathsf{value}}{\mathtt{inl}(\tau_1, \tau_2, e_1) \ \mathsf{value}} \qquad \frac{e_2 \ \mathsf{value}}{\mathtt{inr}(\tau_1, \tau_2, e_2) \ \mathsf{value}}$$

$$\frac{e \mapsto e'}{\mathtt{case}(e, x_1.e_1, x_2.e_2) \mapsto \mathtt{case}(e', x_1.e_1, x_2.e_2)}$$

$$\frac{v_1 \ \mathsf{value}}{\mathtt{case}(\mathtt{inl}(\tau_1, \tau_2, v_1), x_1.e_1, x_2.e_2) \mapsto \{v_1/x_1\}e_1}$$

$$\frac{v_2 \ \mathsf{value}}{\mathtt{case}(\mathtt{inr}(\tau_1, \tau_2, v_2), x_1.e_1, x_2.e_2) \mapsto \{v_2/x_2\}e_2}$$

We also state the value inversion property.

[3]Strictly speaking, some of this information is redundant, but it is easier read if we are fully explicit here.

> If $\cdot \vdash v : \mathtt{sum}(\tau_1, \tau_2)$ then either $v = \mathtt{inl}(\tau_1, \tau_2, v_1)$ with v_1 value or $v = \mathtt{inr}(\tau_1, \tau_2, v_2)$ with v_2 value.

Void type. The empty type `void` can be thought of as a zero-ary sum. It has no values, and can only be given to expressions that do not terminate. For example,

$$\frac{\Gamma, x{:}\mathtt{void} \vdash x : \mathtt{void}}{\Gamma \vdash \mathtt{rec}(\mathtt{void}, x.x) : \mathtt{void}}$$

The value inversion property here just expresses that there are no values of void type.

> If $\cdot \vdash v : \mathtt{void}$ then we have a contradiction.

Run-Time Errors. Next, we discuss how run-time errors can be handled in a type-safe language [Ch. 9.3]. Consider extending our MinML language by a partial division operator, $\mathtt{div}(e_1, e_2)$. Besides the usual typing rules and search rules for the operational semantics, we would also have the following reduction rule:

$$\frac{(n_2 \neq 0)}{\mathtt{div}(\mathtt{num}(n_1), \mathtt{num}(n_2)) \mapsto \mathtt{num}(\lfloor n_1/n_2 \rfloor)}$$

The condition $n_2 \neq 0$ means that there is no rule for $\mathtt{div}(\mathtt{num}(n), \mathtt{num}(0))$ and evaluation gets stuck. Progress would be violated.

We can restore an amended progress theorem if we introduce a new contruct `error` representing a run-time error. This expression allows us to state that the program terminates in an orderly way after an error rather than continuing in some random state. The error expression is created when a divide-by-zero error occurs:

$$\frac{}{\mathtt{div}(\mathtt{num}(n_1), \mathtt{num}(0)) \mapsto \mathtt{error}}$$

We can think of the error expression having any type; thus it is safe to replace any expression with error:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau}$$

However, we are not finished, because an expression such as

$$\mathtt{plus}(\mathtt{div}(\mathtt{num}(3), \mathtt{num}(0)), \mathtt{num}(2))$$

must also abort, but we have no rule that allows us to conclude this. So in addition to the search rules we have "error propagation" rules that propagate run-time errors up to the overall program we are trying to evaluate. We show the two rules for application as an example; similar rules are necessary for all search rules to account for a possible abort.

$$\frac{}{\texttt{apply}(\texttt{error}, e_2) \mapsto \texttt{error}}$$

$$\frac{v_1 \ \textsf{value}}{\texttt{apply}(v_1, \texttt{error}) \mapsto \texttt{error}}$$

Now we can refine the statement of progress to account for the new judgment. Note that preservation and determinism do not change, because replacing any expression with `error` preserves the type.

1. (Preservation) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.
2. (Progress) If $\cdot \vdash e : \tau$ then either
   (i) $e \mapsto e'$ for some e', or
   (ii) e value, or
   (iii) e is `error`.
3. (Determinism) If $\cdot \vdash e : \tau$ then exactly one of
   (i) $e \mapsto e'$ for some unique e', or
   (ii) e value.

We do not give her a proof of these properties. In a future lecture, however, we will discuss how the language might be extended with a `try`...`handle`...`end` construct in order to catch error conditions.

Higher-Order		
Abstract Syntax	Concrete Syntax	Mathematical Syntax
`arrow`(τ_1, τ_2)	τ_1 `->` τ_2	$\tau_1 \to \tau_2$
`cross`(τ_1, τ_2)	τ_1`*`τ_2	$\tau_1 \times \tau_2$
`unit`	`unit`	`1`
`sum`(τ_1, τ_2)	τ_1`+`τ_2	$\tau_1 + \tau_2$
`void`	`void`	`0`
`pair`(e_1, e_2)	`(`e_1`,`e_2`)`	$\langle e_1, e_2 \rangle$
`fst`(e)	`#1` e	$\pi_1\, e$
`snd`(e)	`#2` e	$\pi_2\, e$
`unitel`	`()`	$\langle\,\rangle$
`inl`(τ_1, τ_2, e_1)	`inl`$(e_1) : \tau_1$`+`τ_2	$\mathsf{inl}_{\tau_1+\tau_2}(e_1)$
`inr`(τ_1, τ_2, e_2)	`inr`$(e_2) : \tau_1$`+`τ_2	$\mathsf{inr}_{\tau_1+\tau_2}(e_2)$
`case`$(e, x_1.e_1, x_2.e_2)$	`case` e	$\mathsf{case}(e, x_1.e_1, x_2.e_2)$
	`of` `inl`(x_1) `=>` e_1	
	`|` `inr`(x_2) `=>` e_2	
	`esac`	
`abort`(τ, e)	`abort`$(e) : \tau$	$\mathsf{abort}_\tau(e)$