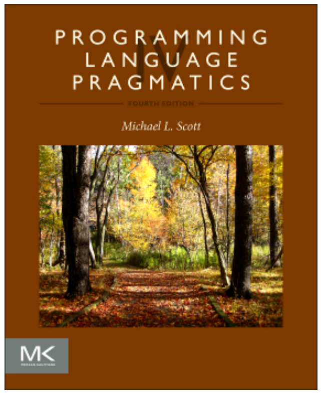
Composite Types

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 8

Ben Titzer

Jonathan Aldrich

Records

- A record has multiple named fields
  - Fields may have different types
  - Order usually doesn’t matter to semantics
  - Layout chosen by compiler
    - or fixed by programmer in C – helps match hardware expectations
- Operations
  - Create: specify initial value for each field   r = { x:5, y:10 }
  - Dereference: read a field   r.x; *// evaluates to 5*
  - Assign: update a field   r.x := 7
    - We’ll model assignment separately later, using references
    - Keeps assignment (and state) orthogonal
- Typing
  - Simple, orthogonal approach: a type for each field

Records

- Syntax
  - Note shorthand for values – *v* is a subset of *e*
  - Notation: overbar indicates a list

$$\begin{array}{lcl} e & ::= & \ldots \mid \{\, \overline{f = e} \,\} \mid e.f \\ v & ::= & n \mid x : \tau \Rightarrow e \mid \{\, \overline{f = v} \,\} \\ \tau & ::= & \ldots \mid \{\, \overline{f : \tau} \,\} \end{array}$$

- Field initialization and dereference

$$\frac{e_i \to e_i'}{\{\, \overline{f_{1..i-1} = v_{1..i-1}}, f_i = e_i, \overline{f_{i+1..n} = e_{i+1..n}} \,\} \to \{\, \overline{f_{1..i-1} = v_{1..i-1}}, f_i = e_i', \overline{f_{i+1..n} = e_{i+1..n}} \,\}}\ \textit{congruence-record}$$

$$\frac{}{\{\, \overline{f_{1..i-1} = v_{1..i-1}}, f_i = v_i, \overline{f_{i+1..n} = v_{i+1..n}} \,\}.f_i \to v_i}\ \textit{step-field}$$

Records

- Typing

$$\frac{\Gamma \vdash \overline{e : \tau}}{\Gamma \vdash \{\, \overline{f = e} \,\} : \{\, \overline{f : \tau} \,\}}\ \textit{T-record}$$

$$\frac{\Gamma \vdash e : \{\, \overline{f : \tau} \,\}}{\Gamma \vdash e.f_i : \tau_i}\ \textit{T-field}$$

- Subtyping
  - Depth subtyping example

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\, \overline{f : \tau} \,\} \leq \{\, \overline{f : \tau'} \,\}}\ \textit{S-depth}$$

{ x:int, y:int } ≤ { x:real, y:real }

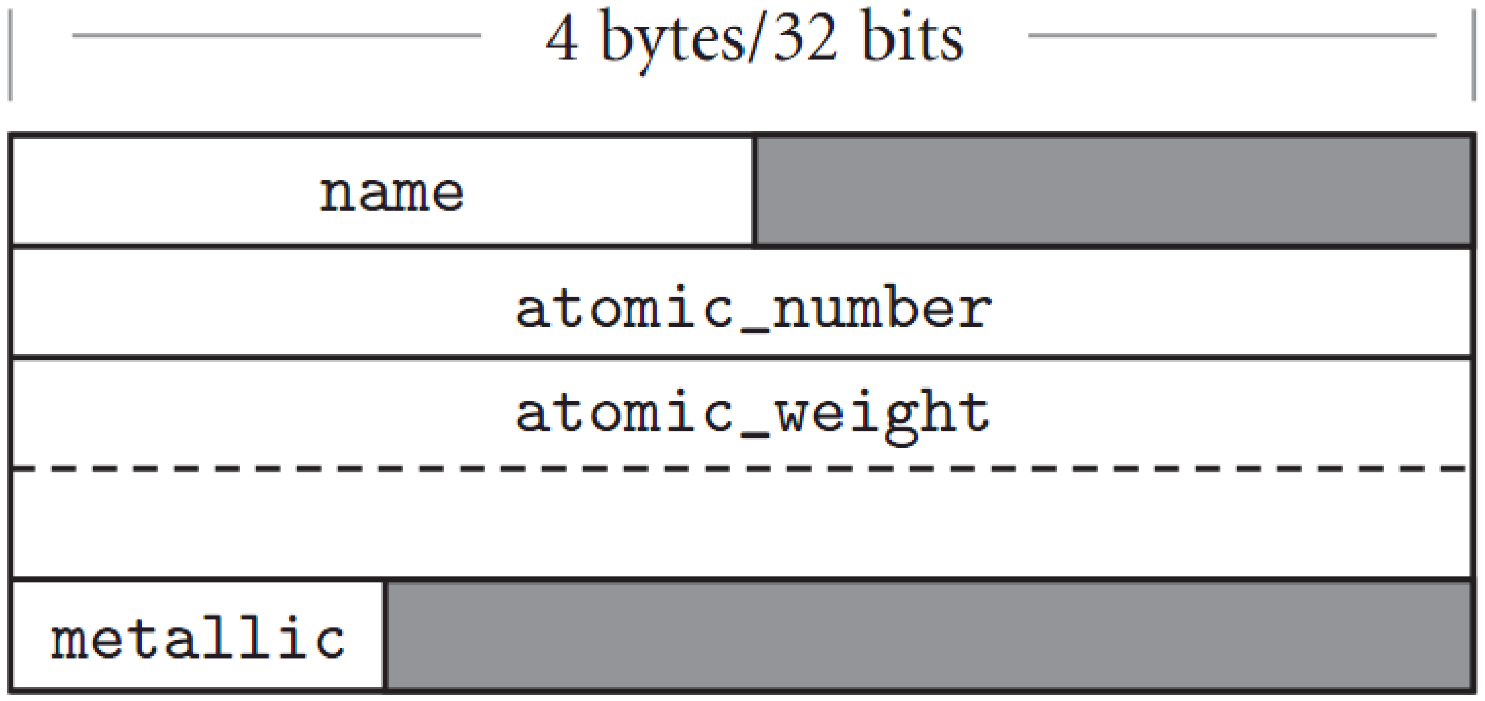
  - Width subtyping example

$$\frac{}{\{\, \overline{f : \tau}, \overline{g : \tau'} \,\} \leq \{\, \overline{f : \tau} \,\}}\ \textit{S-width}$$

{ x:int, y:int, z:int } ≤ { x:int, y:int }

Records (Structures)

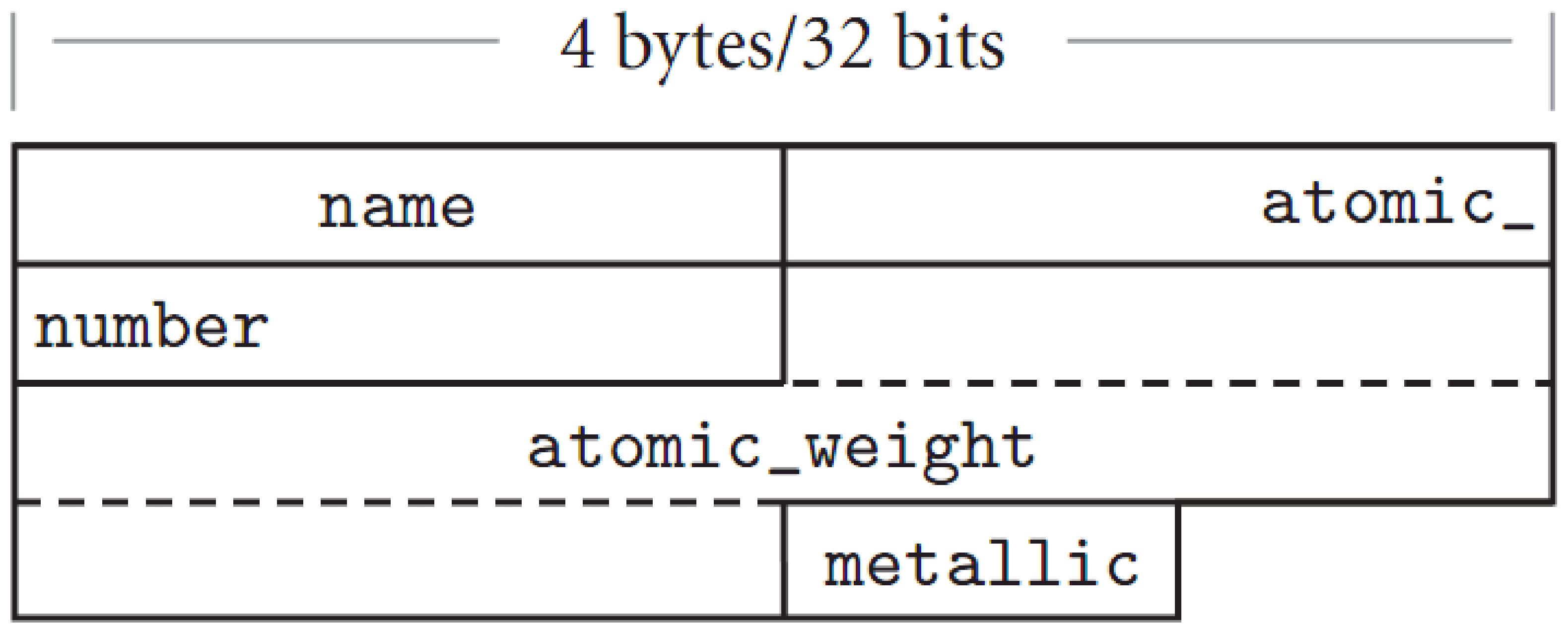
- Memory layout and its impact (structures)



Mjlfmz!mbzpvu!jo!nfnpsz!pg!pckfdut!pg!uzqf hdmp hqwpo b!43.cju!nbdijof/Alignn fou!sftusjdujpot!mfbe!up!uif! tibefe!ëipmft/±

Records (Structures)

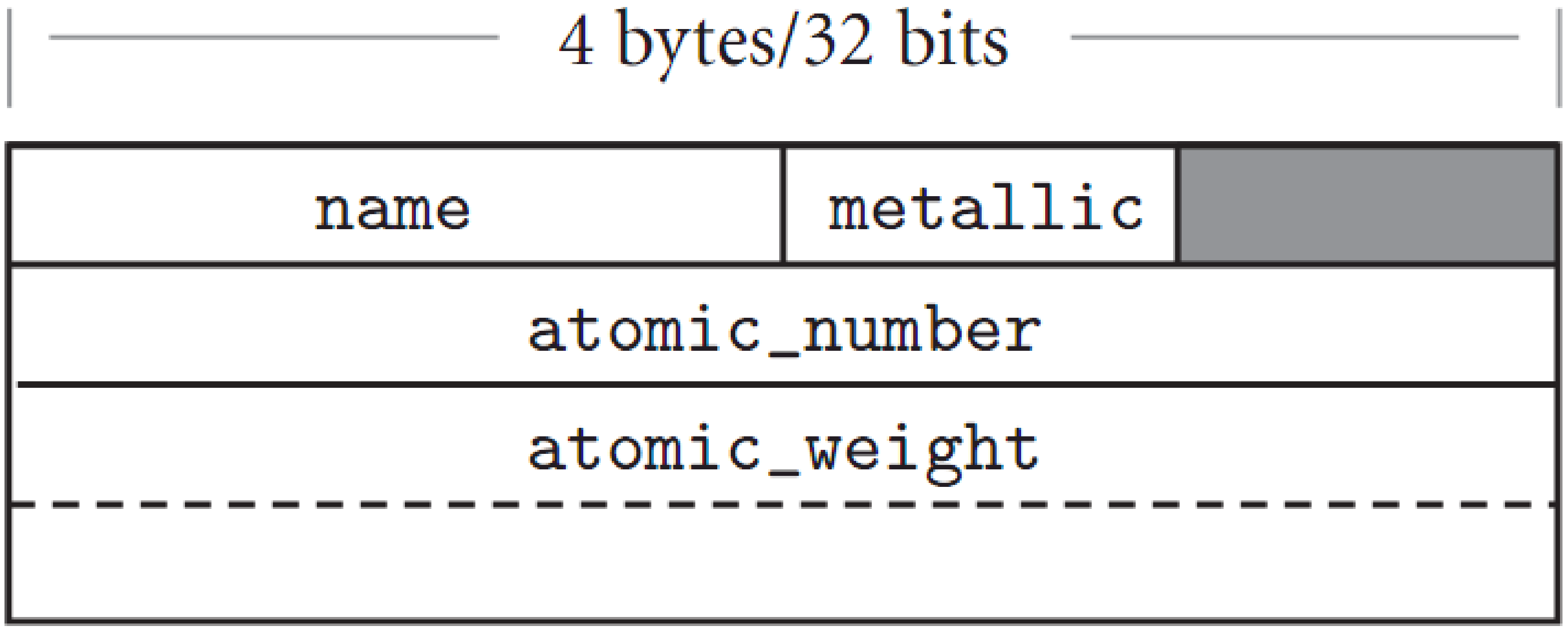
- Memory layout and its impact (structures)



Mjlfz h fn psz hbzpvupstqbdlfe hdnp hqw sfdpset/The dwrp Ifbqxp ehu and dwrp Ifbz hljkw fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

Records (Structures)

- Memory layout and its impact (structures)



Sfbssbohjoh!sfdpse!fimet!up!njojnj{f!ipmft/By sor ting fields according to the size pg!uif!jsbmjhon fou!
dpotusbjou-!b!dpn qjmfs!dbo!njojnj{f!uif!tqbdf!efwpufe!up!ipmft-!xijmf!lffqjoh!uif!fimet!bmjhofe/

Unions (a.k.a. datatypes, …)

- A construct that has 2 or more *variants*
  - Every instance is one variant or the other
  - Comes in two forms:
    - Tagged: The runtime uses a tag to keep track of which variant you have, allows you to test the tag; may enforce consistency
    - Untagged: You have to know which variant of the union is intended. You can "roll your own tag" if needed. May be unsafe.

- Example from C

```
struct address {
   int is_street;   // we use this as a tag
   union {
          int po_box;
          char *street_address;
   } address_details;
}
```

- Example from OCaml

```
// OCaml tracks the tag for us
type address =
   po_box of int
|  street_address of string
```

Formalizing Unions as Sum Types

- Syntax for "sum types" – simple unions with tags
  - We model just two possibilities – easy to generalize
  - Instead of arbitrary names, we use "right" and "left"
  - **inr** *e* "injects" a value into a union using the right (r) variant
  - A **case** construct tests the tag and evaluates e_l or e_r

$$\begin{array}{lcl} e & ::= & \ldots \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \\ v & ::= & \ldots \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\ \tau & ::= & \ldots \mid \tau_l + \tau_r \end{array}$$

ELSEVIER

Dynamic Semantics of Sums

$$\begin{array}{lcl} e & ::= & \ldots \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \\ v & ::= & \ldots \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\ \tau & ::= & \ldots \mid \tau_l + \tau_r \end{array}$$

- Congruence rules handle evaluation when injecting into a union or evaluating the input to a case
- The step rules test the tag and run one body or the other—like an if statement

$$\frac{e \to e'}{\texttt{inl}\ e \to \texttt{inl}\ e'}\ \textit{congruence-inl}$$

$$\frac{e \to e'}{\texttt{inr}\ e \to \texttt{inr}\ e'}\ \textit{congruence-inr}$$

$$\frac{e \to e'}{\texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \to \texttt{case}\ e'\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r}\ \textit{congruence-case}$$

$$\frac{}{\texttt{case}\ \texttt{inl}\ v\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \to [v/x]e_l}\ \textit{step-case-inl}$$

$$\frac{}{\texttt{case}\ \texttt{inr}\ v\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \to [v/x]e_r}\ \textit{step-case-inr}$$

Example of using sums

- Consider modeling addresses as above. The left variant will be PO boxes and the right is street addresses. When we ship, we must use USPS for PO boxes. This function implements that:

ship(address) = **case** address **of inl** n => usps(n), **of inr** a => fedex(a)

ship(**inr** “5000 Forbes Ave”) *// ship to CMU!*
→ **case** (**inr** “5000 Forbes Ave”) **of inl** n => usps(n), **of inr** a => fedex(a)
→ fedex(“5000 Forbes Ave”)

ship(**inl** 1492) *// 1492 is the PO box we are shipping to*
→ **case** (**inl** 1492) **of inl** n => usps(n), **of inr** a => fedex(a)
→ usps(1492)

ELSEVIER

Typechecking Sums

- If we inject a value of type int, the sum type can be int + anything
  - In real languages we know what it is; in the formalism we "guess"
  - We could avoid "guessing" by annotating the **inl** with the expected sum type
- The case rule expects *e* to be a sum, and types the branches assuming the variable has the left and right type, respectively.
  - The branches must have the same type as each other – that way the program can use the result no matter which branch is chosen

$$\frac{\Gamma \vdash e : \tau_l}{\Gamma \vdash \texttt{inl}\ \ e : \tau_l + \tau_r}\ \textit{T-inl}$$

$$\frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash \texttt{inr}\ \ e : \tau_l + \tau_r}\ \textit{T-inr}$$

$$\frac{\Gamma \vdash e : \tau_l + \tau_r \quad \Gamma, x : \tau_l \vdash e_l : \tau \quad \Gamma, x : \tau_r \vdash e_r : \tau}{\Gamma \vdash \texttt{case}\ \ e\ \ \texttt{of}\ \ \texttt{inl}\ \ x \Rightarrow e_l, \texttt{inr}\ \ x \Rightarrow e_r : \tau}\ \textit{T-case}$$

Sum Subtyping

- Just like depth subtyping for records, one sum is a subtype of another if the component types are in the same relationship. "If I'm expecting a dog or a cat, and you give me a Poodle or a Siamese, I'll be OK with that"

$$\frac{\tau_l \leq \tau_l' \quad \tau_r \leq \tau_r'}{\tau_l + \tau_r \leq \tau_l' + \tau_r'} \; \textit{S-sum}$$

- If we were modeling sums with more than 2 variants, then a sum with n variants would be a subtype of a sum with m>n variants that includes the n from the first sum. "If I'm expecting a cat, dog, or horse, and you give me a cat or dog, I'm OK with that. But not vice versa!"

- **Exercise: write a rule for this! (assume n-ary sums τ_1+…+ τ_n)**

Sum Subtyping

- Just like depth subtyping for records, one sum is a subtype of another if the component types are in the same relationship. “If I’m expecting a dog or a cat, and you give me a Poodle or a Siamese, I’ll be OK with that”

$$\frac{\tau_l \leq \tau_l' \quad \tau_r \leq \tau_r'}{\tau_l + \tau_r \leq \tau_l' + \tau_r'} \; \textit{S-sum}$$

- If we were modeling sums with more than 2 variants, then a sum with n variants would be a subtype of a sum with m>n variants that includes the n from the first sum. “If I’m expecting a cat, dog, or horse, and you give me a cat or dog, I’m OK with that. But not vice versa!”

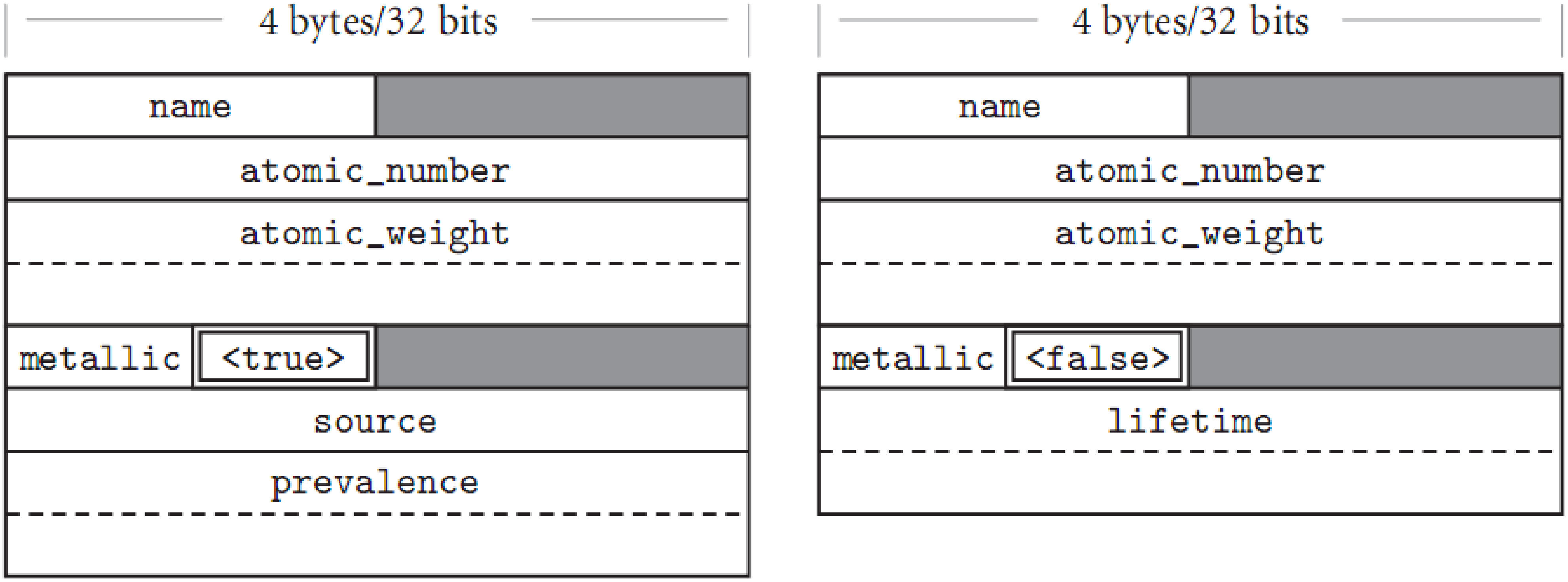
- **Answer to exercise: write a rule for this (assuming n-ary sums $\tau_1 + \ldots + \tau_n$)**

$$\frac{}{\tau_1 + \ldots + \tau_n \leq \tau_1 + \ldots + \tau_n + \ldots + \tau_m} \; \textit{S-sum-width}$$

- **Note that this is the “opposite” of width subtyping for records!**

Records (Structures) and Variants (Unions)

- Memory layout and its impact (unions)



Mjlfmz!mbzpvu!jo!nfnpsz!pg!fmfnfou!wbsjbout/!Uif!wbmvf!pg!uif!bbuvsbmz!‘!pddvssjoh!gjfme!)tipxo!ifsf!xjui!b!epvcmf! cpsefs*!jt!joufoefe!up!joejdbuf!xijdi!pg!uif!joufsqsfubujpot!pg!uif!sfnbjojoh!tqbdf!jt!wbmje/!Gjfme!tpvsdf!jt!bttvnfe!up!qpjou!up!b!tusjoh!uibu! ibt!cffo!joefqfoefoumz!bmmpdbufe/

Pointers

- Pointers serve two purposes:
  - Efficient access to objects on the stack (as in C)
    - Can be unsafe if not carefully managed
    - Rust has a type system that enforces safety
  - Dynamic creation of linked data structures, in conjunction with a heap storage manager
    - Can also be unsafe if dangling pointers are dereferenced
    - Garbage collection can ensure safety

- Languages like Java provide a higher level "reference" model, "building in" pointers
  - We can model references with pointers though

Modeling Pointers

- We model pointers with three constructs:
  - A **new** operation, as in C++ or Java
  - A C-style dereference operation, *p
  - C-style pointer assignments, *p = e
- Types include pointer types τ* (read from right to left, as in C)
- For modeling execution, we'll track locations ℓ on the heap
- A store *S* maps locations to values
- We track the types of locations in the store in a store typing Σ

$$\begin{array}{lll}
e & ::= & \ldots \mid \texttt{new}\ e \mid *e \mid *e := e \\
v & ::= & \ldots \mid \ell \\
\tau & ::= & \ldots \mid \tau* \\
S & : & Location \to Value \\
\Sigma & : & Location \to Type
\end{array}$$

ELSEVIER

Pointer Evaluation Rules

- Congruence rules do the expected thing
- But the program is now a combination of an expression and a store!
- Other rules
  - Create references and add them to the store S, creating a new store S'
  - Dereference a value, looking it up in the store S
  - Assign a new value, updating the store

$$\frac{S, e \to S', e'}{S, \texttt{new}\ \ e \to S', \texttt{new}\ \ e'}\ \textit{congruence-new}$$

$$\frac{S, e \to S', e'}{S, *e \to S', *e'}\ \textit{congruence-deref}$$

$$\frac{S, e_1 \to S', e_1'}{S, *e_1 := e_2 \to S', *e_1' := e_2}\ \textit{congruence-assign-left}$$

$$\frac{S, e_2 \to S', e_2'}{S, *v_1 := e_2 \to S', *v_1 := e_2'}\ \textit{congruence-assign-right}$$

$$\frac{\ell \notin domain(S) \quad S' = [\ell \mapsto v]S}{S, \texttt{new}\ \ v \to S', \ell}\ \textit{step-new}$$

$$\frac{S[l] = v}{S, *\ell \to S, v}\ \textit{step-deref}$$

$$\frac{S' = [\ell \mapsto v]S}{S, *\ell := v \to S', v}\ \textit{step-assign}$$

Pointer Typing Rules

- A new expression has pointer type
- To type a dereference, we look up the type of the pointer and take away the *
- In an assignment, we require *p on the left, where p has pointer type
- The right hand side's type must be a subtype of the pointer type

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{new}\ \ e : \tau *}\ \textit{T-new}$$

$$\frac{\Gamma \vdash e : \tau *}{\Gamma \vdash *e : \tau}\ \textit{T-deref}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 * \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash *e_1 := e_2 : \tau_2}\ \textit{T-assign}$$

Pointer Subtyping

- As mentioned, we can assign a subtype value to a variable that’s a pointer to its supertype:

```
float *r = new 5.0;
*r = 7;   // the compiler will insert a coercion here
```

- But, we can’t assign an int * to a float *, or vice versa! That’s because int and float have different representations; if we write via one pointer and read from the other, the compiler won’t know to insert a conversion, and we’ll get garbage.
- Thus, $\tau_1* \leq \tau_2*$ only if $\tau_1 = \tau_2$
- When we study objects, we’ll see that in C++ a Dog* is a subtype of an Animal*, but that only works if we use the pointers in a limited way as object references, and do not assign into them.

Recursive Types

- Recursive types refer to a type inside its definition
  - Required to describe recursive data structures
- In practice, combined with other type features
  - C structs are records + recursion
  - OCaml datatypes are unions + recursion
- Running example (Ocaml) – integer lists
  - A datatype with a record in one variant

```
type IntList =
   Cons of { value:int, next:IntList }
|  Nil
```

Modeling Recursive Types

type IntList =
  Cons **of** { value:**int**, next:IntList }
| Nil

- We add named recursive types to our type grammar
  - Must also be able to refer to the name

$$\tau \quad ::= \quad \ldots \mid \mathtt{rec}\ T.\tau \mid T$$

- Now we can model lists as follows
  - We use recursive types, sum types, and a record type
  - The names Cons and Nil are just right and left branches of the sum type

rec IntList . { value:**int**, next:IntList } + **unit**

Semantics of Recursive Types

- There are two ways to model the semantics of recursive types
- Both involve *unfolding*
  - We unfold a type by taking the body of the recursive type, and substituting the recursive type for the name everywhere it appears

$$unfold(\mathbf{rec}\ \ T.\tau) = [\mathbf{rec}\ \ T.\tau/T]\tau$$

- The simplest approach, conceptually, is *equi-recursive* types
  - Equi-recursive means the recursive type is equivalent to its unfolding

$$\mathbf{rec}\ \ T.\tau \equiv [\mathbf{rec}\ \ T.\tau/T]\tau$$

- An example of this equivalence for IntList:

rec IntList . { value:**int**, next:IntList } + **unit**

=

{ value:**int**, next:**rec** IntList . { value:**int**, next:IntList } + **unit** } + **unit**

Iso-Recursive Types

- Equi-recursive types are attractive, but hard to implement
  - When does the compiler apply the fold/unfold equality?
- A more common approach is *iso-recursive* types
  - Here, a recursive type is *isomorphic* to its unfolding
  - Isomorphic means they behave the same way, but you have to convert between them
  - The compiler inserts a fold when you create an instance of a recursive type; it inserts an unfold when you access it (e.g. with a case or field dereference)
- An operational way to think about fold and unfold:
  - **fold** makes an object into a recursive type, so we can put it in a data structure
  - **unfold** converts an object back to a sum or record, so we can get its contents

ELSEVIER

Formalizing Iso-Recursive Types

- We now have fold and unfold in expressions. A fold around a value is a value. Remember, these are inserted by the compiler—you don't write them in any real language.

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \mathbf{fold}_{\tau}\ e \mid \mathbf{unfold}\ e \\
v & ::= & \ldots \mid \mathbf{fold}_{\tau}\ v \\
\tau & ::= & \ldots \mid \mathbf{rec}\ T.\tau \mid T
\end{array}
$$

Formalizing Iso-Recursive Types

- Let's look at how OCaml IntLists turn into iso-recursive types:

type IntList = Cons **of** { value:**int**, next:IntList } | Nil
let list = Cons { value = 3, next = Nil }
in match list **with**
        Cons r => r.value
        Nil => 0

Object created;
compiler inserted folds

➔

let list = **fold**$_{IList}$(**inl** { value = 3, next = **fold**$_{\text{IntList}}$(**inr** ()) }
in case unfold$_{IList}$(list) **of inl** r => r.value, **of inr** u => 0

datatype match;
compiler inserts unfold

where I've abbreviated the single-unfolded IntList type as
IList = { value:**int**, next:**rec** IntList . { value:**int**, next:IntList } + **unit**} + **unit**

Formalizing Iso-Recursive Types

- Congruence allows evaluation inside fold/unfold
- When we unfold something that is folded, they cancel:

$$\frac{e \to e'}{\mathtt{fold}_\tau\ e \to \mathtt{fold}_\tau\ e'}\ \textit{congruence-fold}$$

$$\frac{e \to e'}{\mathtt{unfold}\ \ e \to \mathtt{unfold}\ \ e'}\ \textit{congruence-unfold}$$

$$\frac{}{\mathtt{unfold}\ \ \mathtt{fold}_\tau\ v \to v}\ \textit{step-unfold}$$

let list = **fold**$_{IList}$(**inl** { value = 3, next = **fold**$_{IList}$(**inr** ()) }
in case unfold$_{IList}$(list) **of inl** r => r.value, **of inr** u => 0
→
case unfold$_{IList}$(**fold**$_{\text{IntList}}$(**inl** { value = 3, next = **fold**$_{IList}$(**inr** ()) })
of inl r => r.value, **of inr** u => 0

Formalizing Iso-Recursive Types

- Congruence allows evaluation inside fold/unfold
- When we unfold something that is folded, they cancel:

$$\frac{e \to e'}{\texttt{fold}_\tau\ e \to \texttt{fold}_\tau\ e'}\ \textit{congruence-fold}$$

$$\frac{e \to e'}{\texttt{unfold}\ \ e \to \texttt{unfold}\ \ e'}\ \textit{congruence-unfold}$$

$$\frac{}{\texttt{unfold}\ \ \texttt{fold}_\tau\ v \to v}\ \textit{step-unfold}$$

case unfold$_{IList}$(**fold**$_{IList}$(**inl** { value = 3, next = **fold**$_{\text{IntList}}$(**inr** ()) })
of inl r => r.value, **of inr** u => 0

→

case (**inl** { value = 3, next = **fold**$_{\text{IntList}}$(**inr** ()) }) **of inl** r => r.value, **of inr** u => 0

→

{ value = 3, next = **fold**$_{IList}$(**inr** ()) }.value

→ 3

Now the typing rules are easy

$$\frac{\Gamma \vdash e : [\mathtt{rec}\ \ T.\tau/T]\tau}{\Gamma \vdash \mathtt{fold}_{\tau}\ e : \mathtt{rec}\ \ T.\tau}\ \textit{T-fold}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}\ \ T.\tau}{\Gamma \vdash \mathtt{unfold}\ \ e : [\mathtt{rec}\ \ T.\tau/T]\tau}\ \textit{T-unfold}$$

- So, we can typecheck a folded object as follows:

fold*IList*(**inr** ()) } : **rec** IntList . { value:**int**, next:IntList } + **unit**

again, I've abbreviated
IList = { value:**int**, next:**rec** IntList . { value:**int**, next:IntList } + **unit**} + **unit**

Composite Types

- Today we covered the semantics of a number of different composite types
  - Records
  - Unions, datatypes, and sums
  - Reference types
  - Recursive types