A Generic Programming Toolkit for PADS/ML

Mary Fernández, Kathleen Fisher, Yitzhak Mandelbaum
AT&T Labs Research

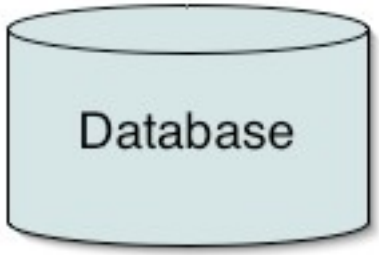
J. Nathan Foster, *Michael Greenberg*
University of Pennsylvania

PADL 2008

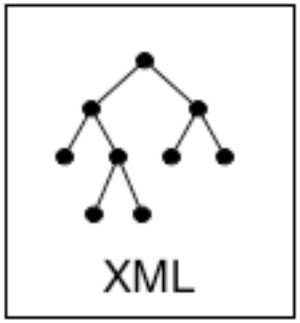
Data, data everywhere!

Incredible amounts of data stored in well-behaved formats:

Databases:



XML:



Tools

- Schema
- Browsers
- Query languages
- Standards
- Libraries
- Books, documentation
- Conversion tools
- Vendor support
- Consultants…

Ad hoc data

- Vast amounts of data in *ad hoc* formats.
- Ad hoc data is semi-structured:
  - Not free text.
  - Not as structured as XML.
  - Different than PL syntax.
- Examples from many different areas:
  - Data mining
  - Consumer electronics
  - Computer science
  - Computational biology
  - Finance
  - More!

Ad Hoc Data in Biology

format-version: 1.0
date: 11:11:2005 14:24
auto-generated-by: DAG-Edit 1.419 rev 3
default-namespace: gene_ontology
subsetdef: goslim_goa "GOA and proteome slim"

[Term]
id: GO:0000001
name: mitochondrion inheritance
namespace: biological_process
def: "The distribution of mitochondria\, including the mitochondrial genome\, into daughter cells after mitosis or meiosis\, mediated by interactions between mitochondria and the cytoskeleton." [PMID:10873824,PMID:11389764, SGD:mcc]
is_a: GO:0048308 ! organelle inheritance
is_a: GO:0048311 ! mitochondrion distribution

www.geneontology.org

Ad Hoc Data in Finance

```
HA00000000START OF TEST CYCLE
aA00000001BXYZ U1AB00000400001OOB0000004200
HL00000002START OF OPEN INTEREST
d 00000003FZYX G1AB00000300003OOOO
HM00000004END OF OPEN INTEREST
HE00000005START OF SUMMARY
f 00000006NYZX B1QB0005200012000007OOOOB0000500000000520000
  0049000000510O+00000100B0000000530000005250000053500O
HF00000007END OF SUMMARY
k 00000008LYXW B1KB0000065G0000009900100000001000020000
HB00000009END OF TEST CYCLE
```

www.opradata.com

Ad Hoc Data from Web Server Logs (CLF)

```
207.136.97.49 - - [15/Oct/1997:18:46:51
-0700]
   "GET /tk/p.txt HTTP/1.0" 200 30
tj62.aol.com - - [16/Oct/1997:14:32:22
-0700]
    "POST /scpt/dd@grp.org/confirm
HTTP/1.0" 200 941
234.200.68.71 - - [15/Oct/1997:18:53:33
-0700]
   "GET /tr/img/gift.gif HTTP/1.0” 200 409
240.142.174.15 - - [15/Oct/1997:18:39:25
-0700]
   "GET /tr/img/wool.gif HTTP/1.0" 404 178
188.168.121.58 - - [16/Oct/1997:12:59:35
-0700]
   "GET / HTTP/1.0" 200 3082
214.201.210.19 ekf - [17/Oct/1997:10:08:23
-0700]
   "GET /img/new.gif HTTP/1.0" 304 -
```

Ad Hoc Data: DNS packets

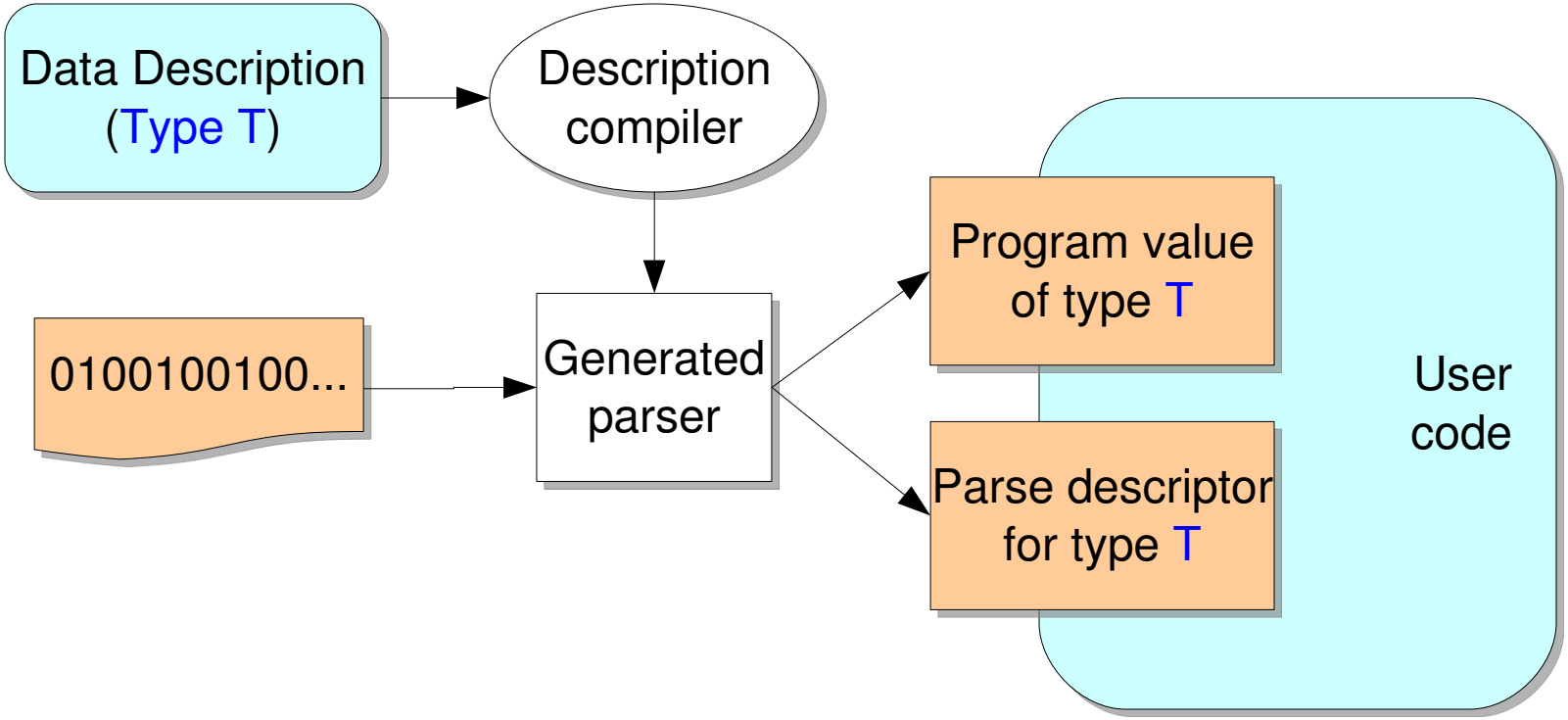
```
00000000: 9192 d8fb 8480 0001 05d8 0000 0000 0872  ...............r
00000010: 6573 6561 7263 6803 6174 7403 636f 6d00  esearch.att.com.
00000020: 00fc 0001 c00c 0006 0001 0000 0e10 0027  ...............'
00000030: 036e 7331 c00c 0a68 6f73 746d 6173 7465  .ns1...hostmaste
00000040: 72c0 0c77 64e5 4900 000e 1000 0003 8400  r..wd.I.........
00000050: 36ee 8000 000e 10c0 0c00 0f00 0100 000e  6...............
00000060: 1000 0a00 0a05 6c69 6e75 78c0 0cc0 0c00  ......linux.....
00000070: 0f00 0100 000e 1000 0c00 0a07 6d61 696c  ............mail
00000080: 6d61 6ec0 0cc0 0c00 0100 0100 000e 1000  man.............
00000090: 0487 cf1a 16c0 0c00 0200 0100 000e 1000  ................
000000a0: 0603 6e73 30c0 0cc0 0c00 0200 0100 000e  ..ns0...........
000000b0: 1000 02c0 2e03 5f67 63c0 0c00 2100 0100  ......_gc...!...
000000c0: 0002 5800 1d00 0000 640c c404 7068 7973  ..X.....d...phys
000000d0: 0872 6573 6561 7263 6803 6174 7403 636f  .research.att.co
```

Challenges of Ad hoc Data

- Data arrives "as is."
- Documentation is often out-of-date or nonexistent.
  - Hijacked fields.
  - Undocumented "missing value" representations.
- Data is buggy.
  - Missing data, "extra" data, …
  - Human error, malfunctioning machines, software bugs (e.g. race conditions on log entries), …
  - Errors are sometimes the *most* interesting portion of the data.

Describing Data with Types

- Types can simultaneously describe both external and internal forms of data.



A PADS/ML Description: Cisco IOS

```
ip vrf 1023
 description ANTI-PESTO S.W.A.T. TEAM|
 export map To_NY_VPN
 route-target 100:3
 maximum routes 150 80
```

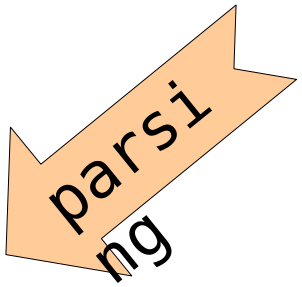
```
ptype ip_vrf_command =
  Description  of "description "   * pstring('|') * '|'
| Export       of "export map "    * pstring('\n')
| Route_target of "route-target " * pint * ':' * pint
| Max_routes   of "max routes "    * pint * ' ' * pint

ptype ip_vrf = {
 header : "ip vrf " * pint * '\n';
 commands : ip_vrf_command plist('\n')
}
```

Describing Data with Types

- Data description describes on-disk layout in a type notation.
- Data description *also* describes type of run-time data.
- Each parsing type has a corresponding program type.
  - `pstring('|')` becomes a `string`
  - `pint`, `pint32`, `pint_FW(3)` become `int`
  - `(α*β)` becomes `(α*β)`
  - …

Parsing

```
ip vrf 1023
 description ANTI-PESTO S.W.A.T. TEAM|
 export map To_NY_VPN
 route-target 100:3
 maximum routes 150 80
```




```
ptype ip_vrf_command =
  Description  of "description "   * ...
| Export       of "export map "    * ...
| Route_target of "route-target "  * ...
| Max_routes   of "max routes "    * ...

ptype ip_vrf = {
 header : "ip vrf " * pint * '\n';
 commands : ip_vrf_command plist('\n')
}
```

```
{   header: 1023,
  commands: [Description "ANTI-PESTO S.W.A.T. TEAM";
             Export "To_NY_VPN";
             Route_target (100, 3);
             Max_routes (150, 80)] }
```

Using Data Descriptions

- Given a data description...
  - Select
  - Summarize
  - Translate
- There are some very specific programs.
  - Intrusion detection given system logs
  - Translate GO to RDF
- Some programs are common to many formats.
  - Serialization to/from XML
  - Statistical analysis

Generic Programming: Theory

- Many of these generic programs can be written as a case analysis on types.
- Each type is built up from base types (int, string, etc.) and structured types:
  - Records, “product types”: `{ f1: t1, ... , fn: tn }`
  - Options, “sum types”: `(O1 t1 | ... | On tn)`
  - Homogeneous lists: `t list`

Typecase: conversion to XML

```
let rec to_xml T v = typecase T v with
  { f1: t1, ... , fn: tn } { f1: v1, ... , fn: vn } ->
    <f1>to_xml t1 v1</f1> ... <fn>to_xml tn vn</fn>
 |(O1 t1 | ... | On tn) Oi vi -> <Oi>to_xml ti vi</Oi>
 |t list [v1; ... ; vn] ->
    <elt>to_xml t v1</elt> ...
    <elt>to_xml t vn</elt>
 |int x -> string_of_int x
 |...
```

Typecase in O'Caml

- Problem: no typecase or run-time types in O'Caml!
- We create run-time type representations.
  - Manually definable
  - Compiler generated
- Representations for each type constructor.
  - Products, sums, base types, etc.
- Generic functions (typecase) encoded as records.
  - One field for each constructor.
- Representations are functions taking a generic function as their first argument.
  - Project and use appropriate field of the generic function.

Typecase: Conversion to XML

```
let rec to_xml = {
  int      = fun n ->  string_of_int n
  product = fun a_ty b_ty (a,b) ->
      <fst>a_ty to_xml a</fst>
      <snd>b_ty to_xml b</snd>
  sum      = fun a_ty b_ty v ->
      match v with
          Left a -> <left>a_ty to_xml a</left>
        | Right b -> <right>b_ty to_xml b</right>
  list     = fun ty ls ->
      List.map
        (fun v -> <elt>ty to_xml v</elt>)
        ls
}
```

Typecase: Conversion to XML

```
type gf_to_xml = {
  int      : int -> xml

  product : 'a 'b . 'a tyrep -> 'b tyrep
                  -> ('a * 'b) -> xml

  sum      : 'a 'b . 'a tyrep -> 'b tyrep
                  -> ('a,'b) sum -> xml

  list     : 'a . 'a tyrep -> -> 'a list -> xml
}
and 'a tyrep = gf_to_xml -> 'a -> xml
```

Generic Functions: Final Technicalities

- Our definition of tyrep is too specific.
  - `'a tyrep = gf_to_xml -> 'a -> xml`
  - Can't use same type representation for from_xml or analyze.
- Use higher-order polymorphism to define parameterized type representations for *classes* of generic functions.
  - `('a,'b) consumer = 'a -> 'b`
  - `('a,'b) producer = 'b -> 'a`
  - Artifact of OCaml's type system.

Generic Functions: Summary

- Generics for the masses...of ad hoc data.
- Compiler-generated type representations.
  - Typecase
  - Higher-order polymorphism
- Third-parties can write advanced tools that work for all data descriptions.

Case Studies

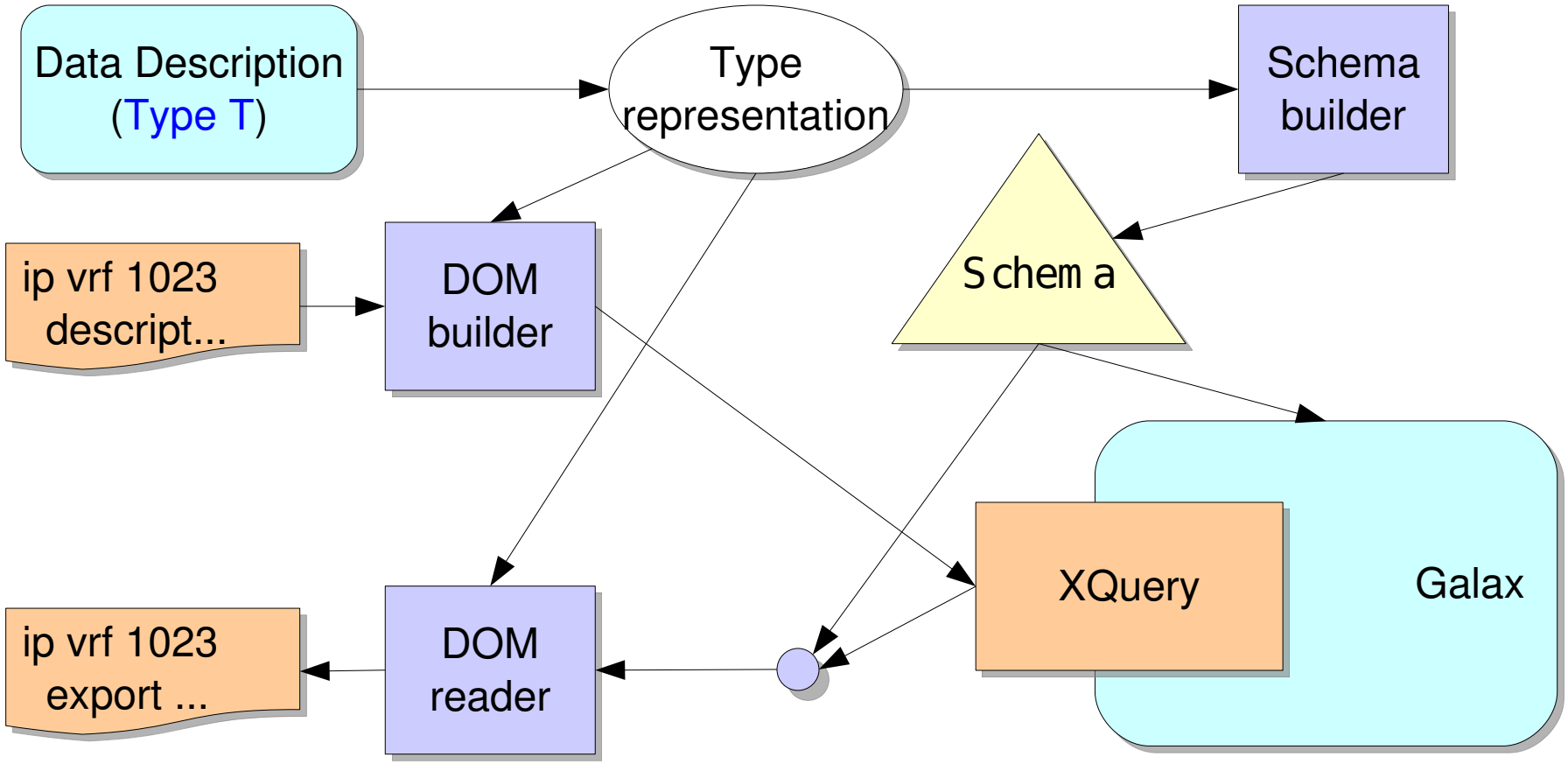
- **PADX version 2**
- Harmony

PADX

- XQuery on ad hoc data.
  - Query language for XML/DOM
  - Existing implementation: Galax
- PADX v1
  - Used older, C version of PADS
  - Complex addition to the compiler
  - Uni-directional
- Goals for v2:
  - Reimplement for PADS/ML
  - Bidirectionality

PADX

- PADX v2 comprises three tools.



Case Studies

- PADX version 2
- **Harmony**

Harmony

- Bidirectional transformations (“lenses”) and translations for trees.
  - Format conversion
  - Synchronization
  - Views and user interfaces
- Works with unordered edge-labeled trees.
  - “Last mile” problem
  - Parsers and printer to load/unload each format.
- Two generic PADS tools to load/unload Harmony trees.
  - 2 + n hand-written programs, not 2n
  - Standard representation

Future work

- Control granularity of generic functions
- Dependent types

  { size: pint; content: pstring_FW(size) }
- Theorems
  - Safety of generated type representations
  - When are two tools inverses?
  - How do schema generation and producers interact?

Summary

- Type-based data description languages are well-suited to describing ad hoc data.
- There are many programs common to many data formats.
- It is possible to encode generic programs over types in O'Caml using type representations and higher-order polymorphism.
- The PADS type-representation system is practical.

For more information on PADS, visit www.padsproj.org.

The End

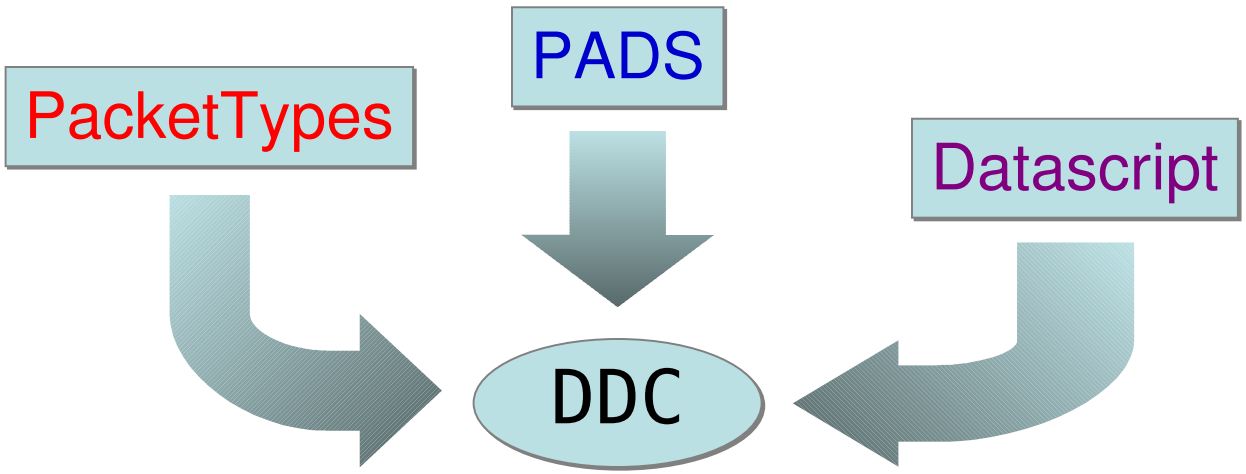
Cut slides follow

Data Description Languages

- Binary Format Description Language (BFD)
- Data Format Description Language (DFDL)
- PacketTypes (*SIGCOMM '00*)
- DataScript (*GPCE '02*)
- Erlang *binaries* (*ESOP '04*)
- PADS (*PLDI '05*)

The Next 700 DDLs

- Develop semantic framework to understand data description languages and guide future development.
- Explain other languages with Data Description Calculus (DDC).
- Give denotational semantics to DDC.



Data Description Calculus

- DDC: calculus of dependent types for describing data.
  - Base types: atomic pieces of data.
  - Type constructors: richer structures.

```
t ::= unit  |  bottom  |  C(e)
      |  Σ x:t.t  |  t + t  |  t &
t  |  {x:t | e}
      |  t seq(t,e,t) |  α  |
μ α .t  |  λ x.e  |  t e
      |  compute (e:σ )  |
absorb(t) |  scan(t)
```

- Specify well-formed types with kinding judgment.

Base Types and Pairs

- Base types: `C (e)`.
  - Abstract.
  - Parameterized by host-language expression.
  - Concrete examples:
    - int, char, stringFW(n), stringUntil(c).
- Ordered pairs: `t * t'` and `Σ x:t.t'`.
  - Variable *x* gives name to first value in pair.
  - Use `t * t'` when `x` does not appear in `t'`.

Base Types and Pairs

```
122Joe|Wright|45|95|
79
n/aEd|Wood|10|47|31
124Chris|Nolan|80|93|
85
125Tim|      Burton|
30|82|71
126George|Lucas|32|
62|40
```

int * stringUntil('|') * char

Base Types and Pairs

```
13C Programming31Types and
13C Programming
Programming Languages20Twenty
Years of PLDI36Modern Compiler
Implementation in ML 27Elements o
f ML Programming
```

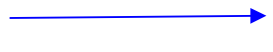
Σ length:int .stringFW(length)

Constraints

- Constrained types: `{x:t | e}` .
  - Enforce the constraint `e` on the underlying type `t`.

```
125Tim|Burton|30|82|71
```

{c:char | c = '|'}
char



$S_c('|')$

$$\Sigma\, min{:}int\,.\, S_c(\text{'|'}) \;*\; \Sigma\, max{:}\{m{:}int \mid min \le m\}\,.\, S_c(\text{'|'}) \;*\; \{avg{:}int \mid min \le avg \,\&\, avg \le max\}$$

Unions

- t + t' : deterministic or : try t; on failure, try t'.
- Example:

122Joe|Wright|45|95|79
n/aEd|Wood|10|47|
124Chris|Nolan|80|93|85
125Tim|Burton|30|82|71
126George|Lucas|32|62|40

S_s("n/a") + int

Absorb, Compute and Scan

- absorb(t) : consume data from source; produce nothing.

```
|                    absorb(Sc('|')
                     )
```

- compute(e:σ) : consume nothing; output result of computation e.

```
10|
12      120          Σ width:int . Sc('|') *
                     Σ length:int.
                       compute(width × length:int)
```

- scan(t) : scan data source for type t.

```
^%$!&_|              scan(Sc('|'))
```

Choosing a Semantics

- Set semantics? But fails to account for:
  - Relationship between internal and external data.
  - Error handling.
  - Types of representation and parse descriptor.
- Parsing semantics.
- Representation semantics.

Semantics Overview

$[\{x:t \mid e\}]_{rep} =$

$[t]_{rep} + [t]_{rep}$

$[\Sigma\ x:t.t']_{rep} =$

$[t]_{rep} * [t']_{rep}$

$[\{x:t \mid e\}]_{pd} = hdr *$

$[t]_{pd}$

$[\Sigma\ x:t.t']_{pd} = hdr *$

$[t]_{pd} * [t']_{pd}$

Description

t

Interpretations of t

$[t]_{rep}$

$[t]_{pd}$

0100100100...

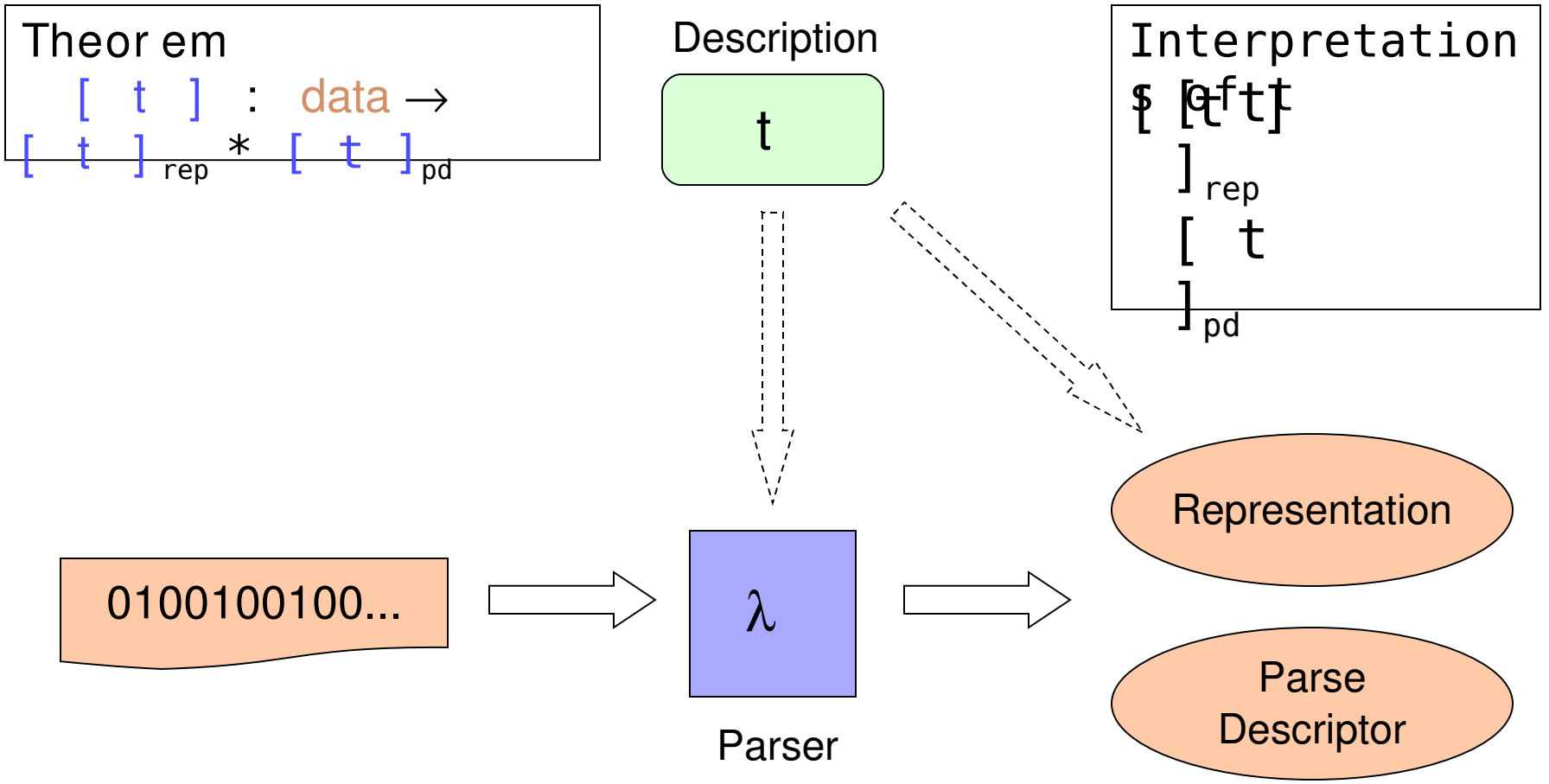
λ

Parser

Representation

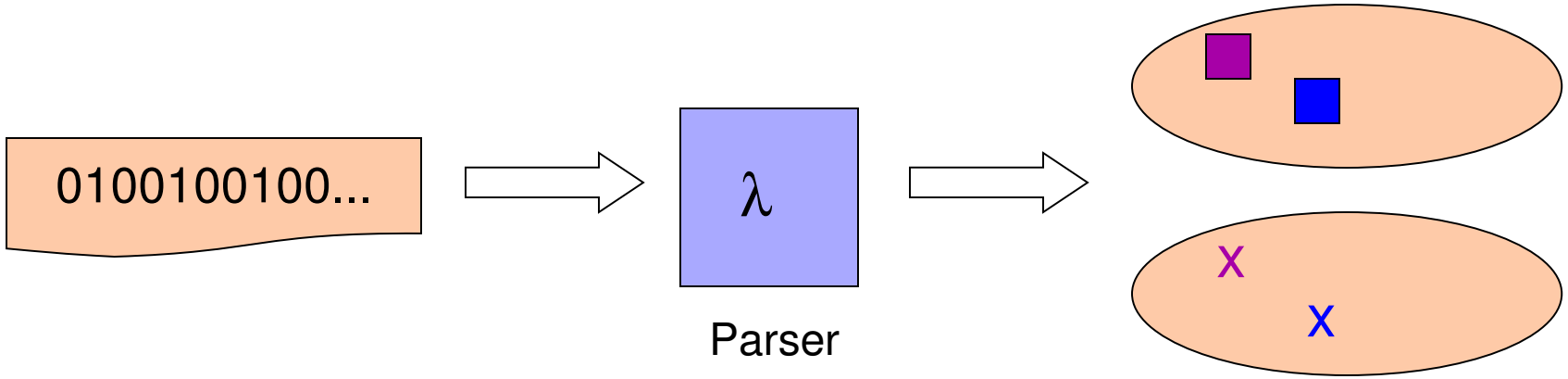
Parse Descriptor

Type Correctness



Error Correctness

- Theorem: Parsers report errors accurately.
  - Errors recorded in parse descriptor correspond to errors in data.
  - Parsers check all semantic constraints.
  - More ...



Encoding DDLs in DDC

- We distill idealized version of PADS (iPADS) and formalize its encoding in DDC.
  - PADS manual: ~350 pages
  - iPADS encoding: 1/2 page
- Majority of other languages encoded in DDC.
  - Either has direct parallel in iPADS,
  - Or, we present encodings directly.
- Remaining cases could be handled with straightforward extension to DDC.

Semantics: Practical Benefits

- Crystallized invariants - uncovered bugs.
  - Semantics forced us to formalize the error counting methodology.
- Guided our design decisions in the subsequent implementation efforts.
  - Added recursion in PADS.
  - Designing version of PADS for O'Caml.
- Helped us revisit design decisions from new perspective.

Existing Approaches

- Most people use C,Perl, or shell scripts.
- Time consuming & error prone to hand code parsers.
  - Binary data particularly so: Programs break in subtle and machine-specific ways (endien-ness, word-sizes).
- Difficult to maintain (worse than the ad hoc data itself in some cases!).
- Often incomplete, particularly with respect to errors.
  - Error code, if written, swamps main-line computation. If not written, errors can corrupt “good” data.

Type Kinding

- Kinding ensures types are well formed.

$$\frac{\Gamma \vdash t : \text{type} \qquad \Gamma, x{:}s \vdash t' : \text{type}}{\Gamma \vdash \Sigma\, x{:}t.t' : \text{type}}$$

$$\frac{\Gamma \vdash t : \text{type} \qquad \Gamma \vdash t' : \text{type}}{\Gamma \vdash t + t' : \text{type}}$$

$$\frac{\Gamma \vdash t : \text{type} \qquad \Gamma, x{:}s \vdash e : \text{bool} \qquad (s = \ldots)}{\Gamma \vdash \{x{:}t \mid e\} : \text{type}}$$

Parsing Semantics

- Each type interpreted as a parsing function in the polymorphic λ-calculus.
  - $[\bullet]$: DDC Type → Function
  - Input data and offset, output new offset, value and parse descriptor

$$[\Sigma x{:}t_1.t_2] = \lambda(bits,offset_0).$$
$$\quad \text{let } (offset_1,data_1,pd_1) = [\,t_1\,](bits,offset_0) \text{ in}$$
$$\quad \text{let } x = (data_1,pd_1) \text{ in}$$
$$\quad \text{let } (offset_2,data_2,pd_2) = [\,t_2\,](bits,offset_1) \text{ in}$$
$$\quad (offset_2,\ R_\Sigma(data_1,data_2),\ P_\Sigma(pd_1,pd_2))$$

Representation Semantics

- Each type interpreted as a two types in the host language (polymorphic λ -calculus).
  - $[\bullet]_{rep}$: type of parsed data.
  - $[\bullet]_{pd}$: type of parse descriptor (meta-data).
- Examples:
  - $[\ C(e)\ ]_{rep} = HostType(C)$
    $[\ C(e)\ ]_{pd} = hdr * unit$
  - $[\ \Sigma\ x{:}t.t'\ ]_{rep} = [\ t\ ]_{rep} * [\ t'\ ]_{rep}$
    $[\ \Sigma\ x{:}t.t'\ ]_{pd} = hdr * [\ t\ ]_{pd} * [\ t'\ ]_{pd}$

Properties of the Calculus

- Theorem: If $\Gamma \vdash t : k$ then
  - $[\![t]\!] = F$ *well formed types yield parsers*

    $\Gamma \vdash F : \text{bits} * \text{offset} \rightarrow \text{offset} * [t]_{rep} * [t]_{pd}$

    a t-Parser returns values with types that correspond to t.
- Theorem: Parsers report errors accurately.
  - Errors in parse descriptor correspond to errors found in data.
  - Parsers check all semantic constraints.
  - More …

Representation Semantics

- Parsers produce values with following type in the host language:

DDC	*Host Language*
Base Types $[C(e)]_{rep}$	I(C) + none
$[unit]_{rep}$	unit
Pairs $[\Sigma\ x{:}T.T']_{rep}$	$[T]_{rep}$ * $[T']_{rep}$
Union $[T + T']_{rep}$	$[T]_{rep}$ + $[T']_{rep}$
Set types $[\{x{:}T \mid e\}]_{rep}$	$[T]_{rep}$ + $[T]_{rep}$
$[absorb(T)]_{rep}$	unit + none

unrecoverable error

dependency erased

error

ok

Representation Semantics

- Parse descriptors have the following type in the host language:

	DDC	*Host Language*
Base Types	$[C(e)]_{pd}$	hdr * unit
	$[unit]_{pd}$	hdr * unit
Pairs	$[\Sigma\ x{:}T.T']_{pd}$	hdr * $[T]_{pd}$ * $[T']_{pd}$
Union	$[T + T']_{pd}$	hdr * ($[T]_{pd}$ + $[T']_{pd}$)
Set types	$[\{x{:}T \mid e\}]_{pd}$	hdr * $[T]_{pd}$
	$[absorb(T)]_{pd}$	hdr*unit