# *The Next 700 Data Description Languages*

Kathleen Fisher
AT&T Labs Research

Yitzhak Mandelbaum, David Walker
Princeton

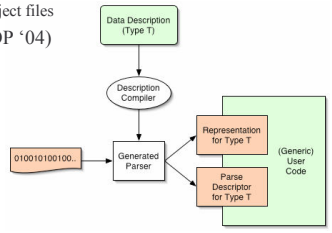
## Review: Technical Challenges of Ad Hoc Data

- Data arrives "as is."
- Documentation is often out-of-date or nonexistent.
  - Hijacked fields.
  - Undocumented "missing value" representations.
- Data is buggy.
  - Missing data, human error, malfunctioning machines, race conditions on log entries, "extra" data, …
  - Processing must detect *relevant* errors and respond in *application-specific* ways.
  - Errors are sometimes the *most* interesting portion of the data.
- Data sources often have high volume.
  - Data may not fit into main memory.



## Many Data Description Languages

- PacketTypes (SIGCOMM '00)
  - Packet processing
- DataScript (GPCE '02)
  - Java jar files, ELF object files
- Erlang Binaries (ESOP '04)
  - Packet processing
- PADS (PLDI '05)
  - General ad hoc data






## The Next 700 Programming Languages

The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc.). They also differ in physical appearance, and more important, in logical structure. *The question arises, do the idiosyncrasies reflect basic logical properties of the situation that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments?* This question is clearly important if we are trying to predict or influence language evolution.

Continued…



## The Next 700 Programming Languages, cont.

To answer it we must think in terms, not of languages, but families of languages. That is to say we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction.

— J. P. Landin
*The Next 700 Programming Languages*, 1965.



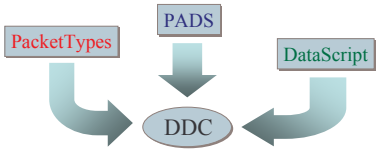
## The Next 700 Data Description Languages

- What is the family of data description languages?
- How do existing languages related to each other?
- What differences are crucial, which "accidents of history"?
- What do the existing languages mean, precisely?

To answer these questions, we introduce a semantic framework for understanding data description languages.

## Contributions

- A core data description calculus (DDC)
  - Based on dependent type theory
  - Simple, orthogonal, composable types
  - Types transduce external data source to internal representation.
- Encodings of high-level DDLs in low-level DDC






## Outline

- Introduction
- A Data Description "| Calculus" (DDC)
- But what does DDC mean?
  - Well-kinding judgment
  - Representation, parse descriptor, and parser generation
- But what do data description languages (DDLs) mean?
  - Idealized PADS (IPADS)
  - Features from other DDLs.
- Applications of the semantics



## A Data Description Calculus

?



## Candidate DDC Primitives

- Base types parameterized by expressions (Pstring(:`|`:))
  - Type constructor constants
- Pair of fields with cascading scope (Pstruct)
  - Dependent products
- Additional constraints (Ptypedef, Pwhere, field constraints).
  - Set types
- Alternatives (Punion, Popt)
  - Sums
- Open-ended sequences (Parray)
  - Some kind of list?
- User-defined parameterized types
  - Abstraction and application
- "Active types": compute, absorb, and scanning
  - Built-in functions



## Base Types and Sequences

- C(e): base type parameterized by expression e.
- $\Sigma x{:}\ \tau.\ \tau'$: dependent product describes sequence of values.
  - Variable $x$ gives name to first value in sequence.
  - Note syntactic sugar: $\tau * \tau'$ if $x$ not in $\tau'$ .
- Examples:

| | | |
|---|---|---|
| "123hello\|" | int * string('\|') * char | (123, "hello", '\|') |
| "3513" | $\Sigma$*width*:int_fw(1). int_fw(*width*) | (3,513) |
| ":hello:" | $\Sigma$*term*:char.string(*term*) * char | (':',"hello",':') |



## Constraints

- $\{x{:}\ \tau \mid e\}$: set types add constraints to the type $\tau$ and express relationships between elements of the data.
- Examples:

| | | |
|---|---|---|
| 'a' | $\{c : \text{char} \mid c = \text{'a'}\}$ *(abbrev: $S_c$('a'))* | inl 'a' |
| "101", "82" | $\{x : \text{int} \mid x > 100\}$ | inl 101, inr 82 |
| "43\|105\|67" | $\Sigma min{:}\text{int}.S_c(\text{'\|'})$ * $\Sigma max{:}\{m{:}\text{int} \mid min \le m\}.S_c(\text{'\|'})$ * $\{mid{:}\text{int} \mid min \le mid \,\&\, mid \le max\}$ | (43, inl '\|', inl 105, inl '\|', inl 67) |

## Unions and the Empty String

- $\tau + \tau'$ : deterministic, exclusive or
  - try $\tau$; on failure, try $\tau'$.
- unit: matches the empty string.
- Examples:

| | | |
|---|---|---|
| "54", "n/a" | int + $S_s$("n/a") | inl 54, inr "n/a" |
| "2341", "" | int + unit | inl 2341, inr () |



## Array Features

- What features do we need to handle data sequences?
  - Elements
  - Separator between elements
  - Termination condition ("Are we done yet?")
  - Terminator *after* sequence
- Examples:
  "192.168.1.1"
  "Harry|Ron|Hermione|Ginny;"



## Bottom and Arrays

- $\tau$ seq($\tau_s$, e, $\tau_t$) specifies:
  - Element type $\tau$
  - Separator types $\tau_s$.
  - Termination condition e.
  - Terminator type $\tau_t$.
- bottom: reads nothing, flagging an error.
- Example: IP address.

| | | |
|---|---|---|
| "192.168.1.1" | int seq($S_c$('.'); len 4, bottom) | [192,168,1,1] |



## Abstraction and Application

- Can parameterize types over values: $\lambda x.\ \tau$
- Correspondingly, can apply types to values: $\tau$ e
- Example: IP address with terminator

| | | |
|---|---|---|
| *none* | $\lambda$*term*.int seq($S_c$('.'); len 4, $S_c$(*term*)) | *none* |
| "1.2.3.4\|" | IP_addr '\|' * $S_c$('\|') | ([1,2,3,4], inl '\|') |



## Absorb, Compute and Scan

- Absorb, Compute and Scan are *active* types.
  - absorb($\tau$) : consume data from source; produce nothing.
  - compute(e:$\sigma$) : consume nothing; output result of computation e.
  - scan($\tau$) : scan data source for type $\tau$.
- Examples:

| | | |
|---|---|---|
| "\|" | absorb($S_c$('\|')) | () |
| "10\|12" | $\Sigma$*width*:int.$S_c$('\|') *<br>$\Sigma$*length*:int.<br>*area*:compute(*width* × *length*:int) | (10,12,120) |
| "^%$!&_\|" | scan($S_c$('\|')) | (6,inl '\|') |



## DDC Example: Idealized Web Server Log

S = $\lambda st.\{s$ : string | $s = st\}$

authid_t = S("-") + string(' ')

response_t = $\lambda x.\{y$ : int16_FW($x$) | 100 <= $y$ and $y$ < 600 }

entry_t =
  $\Sigma$*client* : ip. S(" ") *
  $\Sigma$*remoteid* : authid_t. S(" ") *
  $\Sigma$*response* : response_t 3.
  compute(getdomain *client* = "edu" : bool)

entry_t seq(S("\n"). $\lambda x$.false, bottom)

124.207.15.27 - 234
12.24.20.8 kfisher 208

## A data description calculus

| | |
|---|---|
| C(e) | Atomic type parameterized by expression e |
| $\Sigma x{:}\ \tau.\ \tau'$ | Field sequence with cascading scope |
| $\{x{:}\ \tau \mid e\}$ | Adding constraints to existing descriptions |
| $\tau + \tau'$ | Alternatives |
| $\tau$ seq($\tau_s$; e, $\tau_t$) | Open ended sequences |
| $\lambda x.\ \tau$ | Parameterizing types by expressions. |
| $\tau$ e | Supplying values to parameterized types. |
| unit/bottom | Empty strings: ok/error |
| absorb, compute, scan | "Active types" |



## Semantics Overview

- Well formed DDC type: $\Gamma \vdash \tau : \kappa$
- Representation for type $\tau$ : $[\tau]_{rep}$
- Parse descriptor for type $\tau$ : $[\tau]_{PD}$
- Parsing function for type $\tau$ : $[\tau]$
  - $[\tau]$ : bits * offset → offset * $[\tau]_{rep} * [\tau]_{PD}$



## Type Kinding

- Kinding ensures types are well formed.

$$\frac{\Gamma \vdash \tau : \sigma \rightarrow \kappa \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash \tau\ e : \kappa}$$

$$\frac{\Gamma \vdash \tau : \text{type} \qquad \Gamma \vdash \tau' : \text{type}}{\Gamma \vdash \tau + \tau' : \text{type}}$$

$$\frac{\Gamma \vdash \tau : \text{type} \qquad \Gamma, x{:}[\tau]_{rep} * [\tau]_{pd} \vdash e : \text{bool}}{\Gamma \vdash \{x{:}\ \tau \mid e\} : \text{type}}$$



## Selected Representation Types

| ***DDC*** | ***Host Language*** |
|---|---|
| $[C(e)]_{rep}$ | $\mathbb{I}(C)$ + noval |
| $[\Sigma x{:}\ \tau.\ \tau']_{rep}$ | $[\tau]_{rep} * [\tau']_{rep}$ |
| $[\{x{:}\ \tau \mid e\}]_{rep}$ | $[\tau]_{rep}$ + ($[\tau]_{rep}$ error) |
| $[\tau + \tau']_{rep}$ | $[\tau]_{rep} + [\tau']_{rep}$ + noval |
| [ $\tau$ seq($\tau_s$; e, $\tau_t$) ]$_{rep}$ | int * ($[\tau]_{rep}$ seq) |
| $[\lambda x.\ \tau]_{rep}$, $[\tau\ e]_{rep}$ | $[\tau]_{rep}$ |
| $[unit]_{rep}$ | unit |

unrecoverable error

semantic error

Note that we erase all dependencies.



## Selected Parse Descriptor Types

| ***DDC*** | ***Host Language*** |
|---|---|
| $[C(e)]_{PD}$ | pd_hdr |
| $[\Sigma x{:}\ \tau.\ \tau']_{PD}$ | pd_hdr * $[\tau]_{PD}$ * $[\tau']_{PD}$ |
| $[\{x{:}\ \tau \mid e\}]_{PD}$ | pd_hdr * $[\tau]_{PD}$ |
| $[\tau + \tau']_{PD}$ | pd_hdr * ($[\tau]_{PD}$ + $[\tau']_{PD}$ ) |
| [ $\tau$ seq($\tau_s$; e, $\tau_t$) ]$_{PD}$ | pd_hdr * int * int * ( $[\tau]_{PD}$ seq ) |
| $[\lambda x.\ \tau]_{PD}$, $[\tau\ e]_{PD}$ | $[\tau]_{PD}$ |
| $[unit]_{PD}$ | pd_hdr |

pd_hdr = int * errcode * span



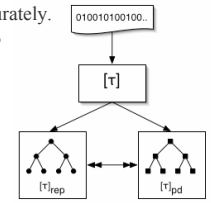
## Parsing Semantics of Types

- Semantics expressed as parsing functions written in the polymorphic $\lambda$-calculus.
  - $[\tau]$ : bits * offset → offset * $[\tau]_{rep} * [\tau]_{PD}$
- Product case:

$[\Sigma x{:}\ \tau.\ \tau'] =$
$\lambda(B, \omega).$
let $(\omega_1, r_1, p_1) = [\tau]\ (B, \omega)$ in
let $x = (r_1, p_1)$ in
let $(\omega_2, r_2, p_2) = [\tau']\ (B, \omega_2)$ in
$(\omega_2, R_\Sigma(r_1, r_2), P_\Sigma(p_1, p_2))$

## Properties of the Calculus

- Theorem: If $\Gamma \vdash \tau : \kappa$ then
  - $\Gamma \vdash [\tau] :$ bits * offset → offset * $[\tau]_{rep}$ * $[\tau]_{pd}$
    "*Well-formed type τ yields a parser that returns values with types corresponding to τ.*"
- Theorem: Parsers report errors accurately.
  - Errors in parse descriptor correspond to errors in representation.
  - Parsers check all semantic constraints.



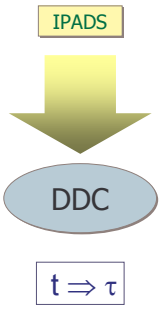



## Making Use of the Calculus

IPADS

t ::= **C**(e) | **Pfun**(x:s) = t | t e
| **Pstruct**{fields} | **Punion**{fields}
| **Pswitch** e **of** {alts $t_{def}$;} | **Popt** t
| t **Pwhere** x.e | **Palt**{fields}
| t **Parray** [t, t] | **Pcompute** e | **Plit** c

fields ::= | fields x : t;
alts ::= | alts e => t;






## IPADS Example

```
authid_t = Punion { unauth : "-";   id  : Pstring(" ")};

response_t = Pfun(x:int)
             Puint16_FW(x) Pwhere y.100 <= y and y < 600;

entry_t = Pstruct {
            client : Pip;                " ";
            remoteid : authid_t;         " ";
            response : response_t 3;
            academic :  Pcompute(getdomain client = "edu" : bool);
          };

entry_t Parray(Peor, Peof)
```

124.207.15.27 - 234
12.24.20.8 kfisher 208



## Example: Popt and Plit

unit
$\tau_1 + \tau_2$

$$\frac{t \Rightarrow \tau}{\textbf{Popt}\ t \Rightarrow \tau + \text{unit}}$$

C(e)
{x:τ | e}
absorb(τ)
scan(τ)

$$\frac{c : \text{char}}{\textbf{Plit}\ c \Rightarrow \text{scan}(\text{absorb}(\{x{:}\text{char} \mid x = c\}))}$$



## Example: Pswitch

τ + τ'
λx.τ
{x:τ|e}

$$\frac{t_i \Rightarrow \tau_i \ \ (i = 1 \ldots n) \qquad t_{def} \Rightarrow \tau_{def}}{\textbf{Pswitch}\ e\ \textbf{of}\ \{e_1 \Rightarrow t_1;\ e_2 \Rightarrow t_2;\ \ldots\ t_{def}\} \Rightarrow (\lambda c.\{x{:}\tau_1 \mid c = e_1\} + \{x{:}\tau_2 \mid c = e_2\} + \ldots + \tau_{def})\ e}$$



## Example: Pswitch

τ + τ'
λx.τ
{x:τ|e}

$$\frac{t_i \Rightarrow \tau_i \ \ (i = 1 \ldots n) \qquad t_{def} \Rightarrow \tau_{def}}{\textbf{Pswitch}\ e\ \textbf{of}\ \{e_1 \Rightarrow t_1;\ e_2 \Rightarrow t_2;\ \ldots\ t_{def}\} \Rightarrow (\lambda c.\{x{:}\tau_1 \mid c = e_1\} + \{x{:}\tau_2 \mid c = e_2\} + \ldots + \tau_{def})\ e}$$

But this encodin- g isn't exactly right, as it parses the data as each branch until it reaches the matching tag.

## Encoding Conditionals

if e then $t_1$ else $t_2$

$$\frac{t_1 \Rightarrow \tau_1 \qquad t_2 \Rightarrow \tau_2}{\text{if e then } t_1 \text{ else } t_2 \;\Rightarrow\; (\{x{:}\text{unit} \mid !e\} + \tau_1) * (\{x{:}\text{unit} \mid e\} + \tau_2)}$$



## Pswitch Revisted

- Encode **Pswitch** as a sequence of conditionals

```
Pswitch e {
  e1 => x1 : t1
  ...
  en => xn : tn
  tdef
}
```

=

```
( Pfun (x : int) =
  if x = e1 then t1 else
  ...
  if x = en then tn else
  tdef )  e
```



## Other Features

- PacketTypes: arrays, where clauses, structures, overlays, and alternation.
- DataScript: set types (enumerations and bitmask sets), arrays, constraints, value-parameterized types, and (monotonically increasing labels).



## Other Uses of the Semantics

- Bug hunting!
  - Non-termination of array parsing if no progress made.
  - Inconsistent parse descriptor construction.
- Principled extensions
  - Adding recursion (done)
  - Adding polymorphism (done in PADS/ML)
- Distinguishing the essential from the accidental
  - Highlights places where PADS/C sacrifices safety.
  - Pomit and Pcompute : much more useful than originally thought
  - Punion : what if correct branch has an error?



## Summary

- Data description languages are well-suited to describing ad hoc data.
- No one DDL will ever be right. Different domains and applications will demand different languages with differing levels of expressiveness and abstraction.
- Our work defines the first semantics for data description languages.
- For more information, visit www.padsproj.org.