Finite Dependent Types

Fancy Types For Systems Metaprogramming And Dependency Management

Alan Jeffrey
Mozilla Research
ajeffrey@mozilla.com

1 INTRODUCTION

Applications such as web browsers often have issues of scale and complexity of the code base. For example, the Servo [11] next generation web engine contains more than 250KLoC:

```
$ loc servo/components/
--------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
--------------------------------------------------------------------------------
 Rust                   811       323350        28930        35208       259212
...
--------------------------------------------------------------------------------
 Total                  961       353834        33406        37472       282956
--------------------------------------------------------------------------------
```

That is just the Servo codebase itself. Servo also makes use of the Cargo [7] software ecosystem, and has over 200 transitive dependencies with more than a 1MLoC in Rust, and 9MLoC in total:

```
$ loc servo/.cargo/
...
 Rust                  2274      1541124        65910       111065      1364149
...
--------------------------------------------------------------------------------
 Total                58295     11784796      1274681      1179475      9330640
--------------------------------------------------------------------------------
```

Building Servo generates even more source code:

```
$ loc servo/target/
...
 Rust                   611       893414        74200        13194       806020
...
--------------------------------------------------------------------------------
 Total                 3901      1660507       174703       107729      1378075
--------------------------------------------------------------------------------
```

Much of this generated code comes from the `script` component, which generates Rust bindings from Web IDL [17]:

```
$ loc servo/target/debug/build/script-*/
...
 Rust                   579       781977        63352         6424       712201
...
--------------------------------------------------------------------------------
 Total                  592       800055        66936         9849       723270
--------------------------------------------------------------------------------
```

The code generator itself is 20KLoC in Python:

```
$ loc servo/components/script/dom/bindings/codegen/
...
 Python                  80        26919         3903         2112        20904
...
--------------------------------------------------------------------------------
 Total                   81        26932         3904         2112        20916
--------------------------------------------------------------------------------
```

There should be a more principled approach to dependency management and metaprogramming.

2 METAPROGRAMMING

Fortunately, metaprogramming is a well-explored area, notably in the Racket [6] programming language's `#lang` ecosystem. Much metaprogramming relies on dynamic checks, since the host language's type system is not usually expressive enough to encode object language types at compile time.

A notable exception is the use of *dependent types* (as implemented in, for example, Coq [3], Agda [18] or Idris [4]) which allow

$$
\begin{aligned}
(A \times B) &= (\textstyle\prod x \in A \cdot B) \\
(A \to B) &= (\textstyle\sum x \in A \cdot B)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{nothing} &\in \mathsf{FSet}(0) \\
\mathsf{unit} &\in \mathsf{FSet}(0) \\
\mathsf{bool} &\in \mathsf{FSet}(1) \\
(\textstyle\prod x \in A \cdot B(x)) &\in \mathsf{FSet}(m+n) \\
&\quad \text{when } B \in (A \to \mathsf{FSet}(n)) \text{ and } A \in \mathsf{FSet}(m) \\
(\textstyle\sum x \in A \cdot B(x)) &\in \mathsf{FSet}(n \ll m) \\
&\quad \text{when } B \in (A \to \mathsf{FSet}(n)) \text{ and } A \in \mathsf{FSet}(m) \\
\mathsf{FSet}(n) &\in \mathsf{FSet}(\mathsf{succ}(n))
\end{aligned}
$$

Figure 1: Type rules for simple finite dependent types, where $\mathsf{FSet}(n)$ is the type of types with at most 2^n elements

the compile-time computation of types which depend on data, Dependent types have already been proposed for low-level programming [9], generic programming [2] and metaprogramming [5].

Dependent metaprogramming. Metaprogramming includes the ability to interpret object languages such as Web IDL. For example:

```
interface Console { log(DOMString moduleURL); };
```

might be interpreted (using types from Figure 1):

$$
\begin{aligned}
\mathsf{Console} = \lambda\, \mathsf{ssize} \in \mathsf{word} \cdot \lambda\, \mathsf{DOMString} \in \mathsf{FSet}(\mathsf{ssize}) \\
\cdot \textstyle\prod \mathsf{csize} \in \mathsf{word} \cdot \textstyle\prod \mathsf{Console} \in \mathsf{FSet}(\mathsf{csize}) \\
\cdot \textstyle\prod \mathsf{log} \in \&((\mathsf{Console} \times \mathsf{DOMString}) \to \mathsf{IO}(\mathsf{unit})) \\
\cdot\, \mathsf{unit}
\end{aligned}
$$

The important point about this interpretation is that it is internal to the system, and can be typed. If we define:

$$\mathsf{CSize} = (\mathsf{WORDSIZE} \ll \mathsf{WORDSIZE}) \ll \mathsf{WORDSIZE}$$

then the typing of Console is internal to the language:

$$\mathsf{Console}(n)(S) \in \mathsf{FSet}(\mathsf{CSize}) \text{ when } S \in \mathsf{FSet}(n) \text{ and } n \in \mathsf{word}$$

Chlipala [5] has shown that dependent metaprogramming can give type-safe bindings for first-order languages like SQL schemas. Hopefully it scales to higher-order languages like Web IDL.

Dependent dependencies. Dependencies are usually versioned, for instance Cargo uses semantic versioning [19]. Semantic versions are triples $[x, y, z]$, where the interface for a package only depends on $[x, y]$, and interfaces with the same x are required to be upwardly compatible. For example an interface at version [1,0] might consist of a sized type T together with an element $\mathsf{z} \in \mathsf{T}$:

$$
\begin{aligned}
\mathsf{A}[1,0] = \textstyle\prod \mathsf{size} \in \mathsf{word} \cdot \textstyle\prod \mathsf{T} \in \mathsf{FSet}(\mathsf{size}) \\
\cdot \textstyle\prod \mathsf{z} \in \mathsf{T} \cdot \mathsf{unit}
\end{aligned}
$$

Submitted for publication, November 2017,

This paper [15] is written in Literate Agda, and typechecked with Agda v2.4.2.5.

One implementation sets T to be unit, but the next sets T to be bool:

$$\mathsf{a}[1,0,1] = (0, \mathsf{unit}, \epsilon, \epsilon) \quad \mathsf{a}[1,0,2] = (1, \mathsf{bool}, \mathsf{false}, \epsilon)$$

Bumping the minor version requires an implementation with a compatible interface, for example:

$$\begin{aligned} \mathsf{A}[1,1] = \textstyle\prod \mathsf{size} \in \mathsf{word} \cdot \prod \mathsf{T} \in \mathsf{FSet}(\mathsf{size}) \\ \cdot \textstyle\prod \mathsf{z} \in \mathsf{T} \cdot \prod \mathsf{s} \in \&(\mathsf{T} \to \mathsf{T}) \cdot \mathsf{unit} \end{aligned}$$

which can be implemented by setting T to be word:

$$\begin{aligned} \mathsf{tsucc} &= (\lambda\, x \cdot \mathsf{truncate}(1 + x)) \\ \mathsf{a}[1,1,0] &= (\mathsf{WORDSIZE}, \mathsf{word}, 0, \mathsf{tsucc}, \epsilon) \end{aligned}$$

Implementations may be dependent, for example B might depend on $\mathsf{A}[1, y]$ for any $y \geq 1$:

$$\mathsf{B}[1,0](\mathsf{size}, \mathsf{A}, \mathsf{z}, \mathsf{s}, \ldots) = \textstyle\prod \mathsf{ss} \in \&(\mathsf{A} \to \mathsf{A}) \cdot \mathsf{unit}$$

with matching implementation:

$$\mathsf{b}[1,0,1](\mathsf{size}, \mathsf{A}, \mathsf{z}, \mathsf{s}, \ldots) = ((\lambda\, x \cdot \mathsf{s}(\mathsf{s}(x))), \epsilon)$$

In summary, an interface $A[x, y]$ is interpreted as family of types where if $y \leq y'$ then $A[x, y] :> A[x, y']$ for an appropriate definition of *interface evolution*. Dependent packages are treated in the style of Kripke semantics, as functions $\forall A[x, y'] <: A[x, y] \cdot B[m, n]$.

There has been much attention paid to dependent types for module systems [13, 14, 16]. In some ways, dependency management is simpler because dependencies are acyclic, but it does introduce interface evolution complexity, for example Rust's [8, #1105].

Finite dependencies. One feature that all of these examples have in common is that they do not require any infinite data. Existing dependent type systems encourage the use of infinite types such such as lists or trees. The prototypical infinite types are $\mathbb{N}$ (the type of natural numbers) and Set (the type of types). This is a mismatch with systems programs, where types are often *sized* (for example in Rust, types are `Sized` by default [10, §3.31]). In particular, systems programs are usually parameterized by WORDSIZE, and assume that data fits into memory (for example that arrays are indexed by a word, not by a natural number).

An overview of approaches to finite sets in dependent type theory is given in [12], which defines a type constructor $\mathcal{K}(A)$ (the Kuratowski-finite subsets of A), and defines the finite types to be a type A together with an $X \in \mathcal{K}(A)$ such that every $x : A$ has $x \in X$. Type constructors such as dependent functions and pairs, preserve finiteness, but the type of finite types is not itself finite.

In Figure 1, we posit a type system with a type-of-types $\mathsf{FSet}(n)$, containing types with at most 2^n elements. In this system, all types are finite, in particular $\mathsf{FSet}(n) \in \mathsf{FSet}(1 + n)$. We expect that this explicit bound on the size of $\mathsf{FSet}(n)$ cannot be expressed internally inside an off-the-shelf dependent type theory, but we conjecture that an extension like this is consistent.

This system is based on a theory of binary arithmetic, but even that is definable within the language, in the presence of an appropriate induction principal for binary arithmetic. For example the type of bitstrings is definable by induction:

$$\begin{aligned} \mathsf{binary} &= \mathsf{indn}(\mathsf{FSet})(\mathsf{unit})(\lambda\, n \cdot \lambda\, A \cdot (\mathsf{bool} \times A)) \\ \mathsf{WORDSIZE} &\in \mathsf{binary}(\mathsf{WORDSIZE}) \end{aligned}$$

As the type for WORDSIZE suggests, this is all spookily cyclic. In particular, binary numbers are parameterized by their bitlength, which is itself a binary number. A bootstrapping induction principal is needed, similar to Agda universe polymorphism [18]. We hypothesize that irrelevant natural numbers might suffice.

Potential advantages of explicitly finite dependent types include:

- A better fit with systems programming languages such as Rust, where sized types are the default.
- Domain specific techniques for arithmetic can better support type inference.
- Simpler induction schemes, such as over the irrelevant naturals, similar to the use of sized types in taming coinduction [1], but without the use of ordinals.

Dependent type systems usually come with an identity type $a \equiv_A b$ where $a : A$ and $b : A$. If identity types are interpreted as paths as in Homotopy Type Theory [20], then the size of $A \equiv_{\mathsf{FSet}(n)} B$ is at most the size of $A \to B$, which would suggest considering $(a \equiv_A b) \in \mathsf{FSet}(n \ll n)$ when $A \in \mathsf{FSet}(n)$. This makes the type of identities over A much larger than the type of A, which may give problems with, for example, codings of indexed types.

3 CONCLUSIONS

Dependent types are a good fit for some of the more difficult problems with programming in the large: metaprogramming and dependency management. However, their focus on infinite types is a mismatch. Systems are finite, and are better served by systems which encourage the use of finite types.

REFERENCES

[1] A. Abel and B. Pientka. Well-founded recursion with copatterns and sized types. *J. Functional Programming*, 26, 2016.
[2] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proc. IFIP TC2/WG2.1 Generic Programming*, pages 1–20, 2003.
[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
[4] E. Brady. *Type-Driven Development with Idris*. Manning, 2017.
[5] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. ACM Programming Language Design and Implementation*, pages 122–133, 2010.
[6] The Racket Community. Racket: Solve problems, make languages. `https://racket-lang.org/`.
[7] The Rust Community. Rust package registry. `https://crates.io/`.
[8] The Rust Community. Rust RFCs. `http://rust-lang.github.io/rfcs/`.
[9] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. Euro. Symp. Programming*, pages 520–535, 2007.
[10] The Rust Project Developers. The Rust programming language, 1st ed. `https://doc.rust-lang.org/book/first-edition/`, 2011.
[11] The Servo Project Developers. Servo, the parallel browser engine project. `https://servo.org/`.
[12] D. Frumin, H. Geuvers, L. Gondelman, and N. van der Weide. Finite sets in homotopy type theory. 2018. Submitted for publication.
[13] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, 1993.
[14] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Proc. ACM Symp. Principles of Programming Languages*, pages 341–354, 1990.
[15] A. S. A. Jeffrey. Thoughts on finite dependent types. `https://github.com/asajeffrey/finite-dtypes`.
[16] D. B. MacQueen. Using dependent types to express modular structure. In *Proc. ACM Sympo. Principles of Programming Languages*, pages 277–286, 1986.
[17] C. McCormack, B. Zbarsky, T. Langel, et al. Web IDL. `https://heycam.github.io/webidl/`.
[18] U. Norell, N. A. Danielsson, A. Abel, et al. Agda. `http://wiki.portal.chalmers.se/agda`.
[19] T. Preston-Werner. Semantic versioning 2.0.0. `http://semver.org/`.
[20] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.