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 [10] 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 [6] 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 [15]:

```
$ 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
```

Is there a more principled approach to code generation?

2 METAPROGRAMMING

Fortunately, metaprogramming is a well-explored area, notably in the Racket [5] 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 [2], Agda [16] or Idris [3]) which allow

$$(A \times B) = (\textstyle\prod x \in A \cdot B)$$
$$(A \to B) = (\textstyle\sum x \in A \cdot B)$$

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

Figure 1: Type rules for simple finite dependent types

the compile-time computation of types which depend on data, Dependent types have already been proposed for low-level programming [8], generic programming [1] and metaprogramming [4].

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

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

might be interpreted as:

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

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

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

then the typing of Console is internal to the language:

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

Chlipala [4] 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 [17]. 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 $z \in T$:

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

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

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

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

$$A[1,1] = \prod \text{size} \in \text{word} \cdot \prod T \in \text{FSet}(\text{size})$$
$$\cdot \prod z \in T \cdot \prod s \in \&(T \rightarrow T) \cdot \text{unit}$$

which can be implemented by setting T to be word:

$$\text{tsucc} = (\lambda x \cdot \text{truncate}(1+x))$$
$$a[1,1,0] = (\text{WORDSIZE}, \text{word}, 0, \text{tsucc}, \epsilon)$$

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

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

with matching implementation:

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

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 [11, 12, 14]. In some ways, dependency management is simpler because the dependency graph is acyclic, but it does introduce interface evolution complexity, for example Rust's [7, #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 in1finite 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 [9, §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).

A simple language of finite types is given in Figure 1. In this language, all types are finite, in particular $\text{FSet}(n) \in \text{FSet}(1+n)$ (the size is slightly arbitrary, we could have chosen any increasing size, for instance $\text{FSet}(n) \in \text{FSet}(1 \ll n)$).

The system is based on a theory of binary arithmetic, but even that is definable within the language, for example the type of bitstrings is definable by induction:

$$\text{binary} = \text{indn}(\text{FSet})(\text{unit})(\lambda n \cdot \lambda A \cdot (\text{bool} \times A))$$

For example:

$$\text{WORDSIZE} \in \text{binary}(\text{WORDSIZE})$$

Some issues finite types raise include:

Induction: 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. An induction principal is needed, even if it is extra-logical, similar to Agda's universe polymorphism [16]. We hypothesize that a theory of irrelevant natural numbers might suffice.

Path types: 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 [18], then the size of $A \equiv_{\text{FSet}(n)} B$ is at most the size of $A \rightarrow B$,

which would suggest considering $(a \equiv_A b) \in \text{FSet}(n \ll n)$ when $A \in \text{FSet}(n)$. 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.

Theory of binary arithmetic: The theory of finite types is very dependent on the theory of natural numbers, and it is very easy to end up type checking dependent on properties such as associativity and commutativity of addition. Such theorems could be handled by an SMT solver, but this has its own issues, such as the interaction between the SMT solver and type inference, and the complexity of an SMT solver being visible to the end programmer.

Pointers In systems programming languages such as Rust, cyclic data structures are mediated by pointers. In a finite type system, we could allow a type of pointers $\&(A)$ where $\&(A) \in \text{FSet}(\text{WORDSIZE})$ when $A \in \text{FSet}(n)$. Pointer creation can fail in low-memory situations, so should be encapsulated in a monad, similar to Haskell's ST monad. Care needs to be taken about pointers to data which includes sets, in particular $\&(\text{FSet}(\text{WORDSIZE})) \in \text{FSet}(\text{WORDSIZE})$ is very close to introducing unsoundness.

More research is needed!

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] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proc. IFIP TC2/WG2.1 Generic Programming*, pages 1–20, 2003.
[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.
[3] E. Brady. *Type-Driven Development with Idris.* Manning, 2017.
[4] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. ACM Programming Language Design and Implementation*, pages 122–133, 2010.
[5] The Racket Community. Racket: Solve problems, make languages. https://racket-lang.org/.
[6] The Rust Community. Rust package registry. https://crates.io/.
[7] The Rust Community. Rust RFCs. http://rust-lang.github.io/rfcs/.
[8] 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.
[9] The Rust Project Developers. The Rust programming language, 1st ed. https://doc.rust-lang.org/book/first-edition/, 2011.
[10] The Servo Project Developers. Servo, the parallel browser engine project. https://servo.org/.
[11] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, 1993.
[12] 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.
[13] A. S. A. Jeffrey. Thoughts on finite dependent types. https://github.com/asajeffrey/finite-dtypes.
[14] D. B. MacQueen. Using dependent types to express modular structure. In *Proc. ACM Sympo. Principles of Programming Languages*, pages 277–286, 1986.
[15] C. McCormack, B. Zbarsky, T. Langel, et al. Web IDL. https://heycam.github.io/webidl/.
[16] U. Norell, N. A. Danielsson, A. Abel, et al. Agda. http://wiki.portal.chalmers.se/agda.
[17] T. Preston-Werner. Semantic versioning 2.0.0. http://semver.org/.
[18] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.