TypOS: An Operating System for Typechecking Actors

Guillaume Allais[1], Malin Altenmüller[2], Conor McBride[2], Georgi Nakov[2], Fredrik Nordvall Forsberg[2], and Craig Roy[3]

[1] University of St Andrews, Fife, United Kingdom
[2] University of Strathclyde, Glasgow, United Kingdom
[3] Quantinuum & Cambridge Quantum, Cambridge, United Kingdom

Introduction We report work in progress on TypOS, a domain-specific language for experimenting with typecheckers and elaborators. Our remit is similar to those of other domain-specific languages such as Andromeda [1], PLT Redex [3], or Turnstyle+ [2]. However, we try to minimise demands on the order in which subproblems are encountered and constraints are solved: when programs contain holes, and constraints are complex, it helps to be flexible about where to make progress. TypOS tackles typing tasks by spawning a network of concurrent actors [4], each responsible for one node of the syntax tree, communicating with its parent and child nodes via channels. Constraints demanded by one actor may generate enough information (by solving metavariables) for other actors to unblock. Metavariables thus provide a secondary and asynchronous means of communication, creating subtle difficulties modelling actor resumptions as host-language functions, because metavariables unsolved at the time of a resumption's creation may be solved by the time it is invoked. Thus forced into a more syntactic representation of suspended processes, we decided to explore what opportunities a new language could bring.

Term syntax and judgement forms TypOS supports a generic LISP-style syntax for terms, including atoms ('a), cons lists ([a b c]), and an α-invariant notion of term variables (x) and binding ($\backslash x.t$). Users can restrict the shape of such terms using context-free grammars, for example defining the syntactic categories of simple types ('`Type`), synthesisable ('`Synth`), and checkable terms ('`Check`). The notion of *judgement form* is recast as channel *interaction protocol* [5], specifying what to communicate, and in which direction. We declare an actor by giving its name and protocol. For example, we may declare type checking and synthesis actors:

```
check : ?'Type. ?'Check. -- receives a type, then a checkable term
synth : ?'Synth. !'Type. -- receives a synthesisable term, sends a type back
```

Information about free variables is kept in contexts, which must also be declared, for example:

```
ctxt |- 'Synth -> 'Type  -- declare ctxt to map 'Synth variables to types
```

Typing rules For each judgement form, we define an actor as the server for all the rules whose conclusion takes that form. Where rules have premises, a typing actor spawns children, each with its own channel. Actors may fork parallel subactors, generate fresh metavariables, declare constraints, bind local variables, match on terms using a scope-aware pattern language, and extend and query contexts. For example, here are actors for bidirectional type checking and type synthesis of the simply typed lambda calculus, following the interaction protocol above:

```
check@p = p?ty. p?tm. case tm -- receive type and term via p, match on tm
  { ['Lam \x. body] ->        -- lambda case:
     'Type?S. 'Type?T.        --   make fresh type metavars S and T
     ( ty ~ ['Arr S T]        --   in parallel: constrain ty as arrow type
```

```
    | \x.                    --   and bring fresh x into scope
      ctxt { x -> S }.       --   then extend ctxt to map x to S
      check@q. q!T. q!body.) --   then spawn child on channel q to check body
  ; ['Emb e] ->              -- embedded synthesisable term e:
    synth@q. q!e. q?S.       --   spawn child to synthesise type S for e
    S ~ ty }                 --   constrain S as ty

synth@p = p?tm. if tm in ctxt -- receive term tm via p, query ctxt for tm
  { S -> p!S. }               -- if tm is a variable in ctxt, send its type
  else case tm                -- otherwise match on tm
  { ['Ann t T] ->             -- type annotated term case: in parallel
    ( type@q. q!T.            --   spawn child to validate type T
    | check@r. r!T. r!t.      --   spawn child to check t has type T
    | p!T. )                  --   send type T
  ; ['App f s] ->             -- application case:
    'Type?S. 'Type?T.         --   make fresh type metavars S and T
    ( synth@q. q!f. q?F.      --   spawn child to synthesise a type F for f
      F ~ ['Arr S T]          --   constrain F as arrow type
    | check@r. r!S. r!s.      --   spawn child to check argument
    | p!T. ) }                --   send the target type back
```

Schematic variables and scope management The notion of *schematic variable* in a typing rule is recast as the ordinary notion of program variable (e.g. `ty`, `tm`, `S`, `T` above) within an actor. We are careful to distinguish these 'actor variables' from both 'term variables' of the syntax being manipulated (e.g. `x`), and 'channel variables' (e.g. `p`, `q`, `r`). Schematic variables in typing rules are usually thought of as implicitly universally quantified: by contrast, each of our actor variables has one explicit binding site in an actor process, either at an input action, or in a pattern-match. As in Delphin [8], the term variables in scope at each point in an actor are determined by explicit binding constructs. The scope for the signals on a channel is bounded by the scope at its creation, ensuring that parents never encounter variables bound locally by their children. Internally, we use a precisely scoped co-deBruijn representation of terms [7].

Metavariables and constraints The actor model principle of sharing by message not memory lets us guarantee that each actor has direct knowledge of only those term variables it has bound itself. Actors cannot learn the names of any term variables free when they were spawned, and are thus less likely to violate stability under substitution. Nonetheless, we do support the one form of memory that distributed concurrent processes may safely share [6]: the metavariables actors create and share mutate monotonically only by becoming more defined. Actors cannot detect that a metavariable is unsolved, but block when matching strictly on one. Decisions already taken on the basis of less information need never be retracted when more arrives.

Executing actors In the future, we will implement a concurrent runtime, but for the moment, we use a stack-based virtual machine. Each actor runs until it blocks, then the machine refocuses on the next place progress can be made, until execution stabilises. The derivation thus constructed can readily be extracted from the final configuration of the stack.

Try it yourself TypOS is available at https://github.com/msp-strath/TypOS, together with more examples of actors.

References

[1] Andrej Bauer, Philipp G. Haselwarter, and Anja Petkovic. Equality checking for general type theories in andromeda 2. In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff, editors, *ICMS '20*, volume 12097 of *Lecture Notes in Computer Science*, pages 253–259. Springer, 2020.

[2] Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent type systems as macros. *Proc. ACM Program. Lang.*, 4(POPL):3:1–3:29, 2020.

[3] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[4] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI '73*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[5] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93*, pages 509–523. Springer, 1993.

[6] Lindsey Kuper. *Lattice-Based Data Structures For Deterministic Parallel And Distributed Programming.* PhD thesis, Indiana University, 2015.

[7] Conor McBride. Everybody's got to be somewhere. In Robert Atkey and Sam Lindley, editors, *MSFP '18*, volume 275, pages 53–69, 2018.

[8] Adam Poswolsky and Carsten Schürmann. System description: Delphin – A functional programming language for deductive systems. *Electron. Notes Theor. Comput. Sci.*, 228:113–120, 2009.