Predictable Macros for Hindley-Milner

(Extended Abstract)

Langston Barrett
Galois, Inc.
Portland, Oregon, USA
langston@galois.com

David Thrane Christiansen
Galois, Inc.
Portland, Oregon, USA
dtc@galois.com

Samuel Gélineau
SimSpace Corporation
Montréal, Québec, Canada
gelisam@gmail.com

We present "Klister", an ML-like language with hygienic type-driven macros that have a predictable yet procedural programming model. Klister's macros are *hygienic* in that they prevent variable capture by default, and they are *type-driven* in that macros have access to the type that is expected for the expression that is to be produced. Being *procedural* means that macros are written in the full Klister language, rather than a restricted subset in the style of Scheme's `syntax-rules` [Kelsey et al. 1998, Sec. 4.3], and *predictable* means that macro developers can be blissfully unaware of the type checker's implementation details, in particular the order in which it traverses expressions.

The `const*` macro (below) is type-driven: it uses the type expected at its expansion site to determine how many arguments should be ignored by the curried constant function that it generates. For simplicity's sake, Klister's syntax is currently Lisp-style, and `the` is the syntax for type annotations.

```
(the (-> Bool Bool Bool Bool Unit)
     (const* unit))
```

The definition of `const*` pattern-matches explicitly on an expected type `t`. Even if the expression `e` contains free references to a name `arg`, the resulting code will not capture them.

```
(define-macro (const* e)
  ...
    (type-case t
      [(-> a b)
       (pure `(lambda (arg) (const* ,e)))]
      [(else x) (pure e)]))
```

We expect this essentially unrestricted *ad hoc* polymorphism to be useful when implementing embedded domain-specific languages that rely on the host language's type checker for correctness. Once Klister is expressive enough, we hope to apply techniques from Chang et al. [2020, 2017], with the added benefit of a trusted host-language type system to catch mistakes in the embedded type system.

1 Macros and Types

Klister's macro expander achieves hygiene via a variant of Flatt's set-of-scopes algorithm [Flatt 2016]. The effects available in Klister's macro monad are primarily a subset of those available in Racket's macro expander, such as comparing identifiers based on their bindings or signaling errors.

In Klister, *problems* correspond to judgments in the type system that user syntax is expected to derive: to construct an expression with a given type, to construct a valid type, to constuct a valid declaration, and so forth. Klister macros can query which problem the current expansion task must solve. Indeed, these problems serve as the source of types to be inspected for metaprogramming:

```
(define-macro (const* e)
  (do (<- prob (which-problem))
      (case prob
        [(expression t)
         (type-case t ...)])))
```

The `which-problem` operator connects type-checking problems in the core typed language to the user's instructions for solving them. The primitive forms in Klister serve dual roles: like Racket's primitives, they propagate the lexical context information that enables lexical scope and macro hygiene, and like ML's syntax, they give rise to types and type equalities.

2 The Problem of Predictability

Principal typing gives ML programmers the freedom to write programs without having to understand the implementation of their type checker. If a program is well typed, then it has the same type, no matter the type checker's traversal order.

This predictability disappears as soon as the antecedent no longer holds: when a program is not *yet* well typed, the type checker's implementation is of urgent importance to users. The order in which the input program is traversed and type constraints are generated and solved governs the display of user feedback, and improving type error diagnosis from ML-like languages is an important area of research (Heeren [2005, Chap. 3] has a good overview). Without attention from language implementors, users can wind up with radically different error messages after simple refactorings or when using different versions of the programming language.

The presence of type-driven procedural macros requires that macro expansion be interleaved with type checking. The position of a macro invocation in a program determines the type that is expected for it, and the expansion of one macro can solve unification constraints that provide information about another invocation's type. This mirrors the problem of type error diagnosis: details of the type checker's implementation leak into both the order in which macros are executed

and the amount of type information available when a given macro is executed. After all, programs that have not yet been expanded are not yet well-typed.

For example, suppose the programmer decides to give a local name to the function generated by `const*`, by introducing a `let`[1] expression:

```
(let ([returns-unit (const* unit)])
  (the (-> Bool Bool Bool Bool Unit)
       returns-unit))
```

This refactoring should preserve the meaning of the program.

Unfortunately, under a straightforward implementation of Hindley-Milner with added macros, this may not be the case. It is possible that `const*` now receives a unification variable instead of a concrete type, so the `(-> a b)` pattern does not match. If the catch-all branch were selected instead, `const*` would generate `unit`, and the program would no longer be well typed.

To make type-driven macros predictable, the language must guarantee that even if the type checker and macro expander traverses the program in a random order, its result is still deterministic. In Klister, when a macro requests information that is not yet available, it gets temporarily *stuck*, blocking until the required information is available.

3 Getting Stuck, and Unstuck

Traditional macro expanders work from the outside of the program towards the inside, from left to right, trampolining between the expander and each individual macro. Macros are run in a predictable order, allowing them to communicate via predictable side effects. Klister's architecture, on the other hand, is more like an elaborator for a dependently-typed language such as Idris [Brady 2013], Agda [Norell 2007], or Lean [de Moura et al. 2015].

Klister's macro expander maintains a set of tasks to be performed together with a partially-constructed program in the core ML dialect and a union-find data structure that represents the type checker's current knowledge about the program's types. Instead of type checking or expanding subexpressions immediately, Klister saves them as future tasks to be performed, along with appropriate type constraints and dependency information. Some examples of tasks are expanding user macros, type checking primitive ML syntactic forms, adding a top-level declaration to the current module, and generalizing a definition's type. Dependency information prevents *e.g.* generalizing a definition's type before its body has been fully type checked and invoking a macro before its definition has been expanded, type checked, and evaluated.

When a macro attempts to analyze a type that has not yet become known, it gets temporarily stuck. The macro's continuation is captured and saved as a task to be reinvoked with the solution. Unlike Agda [Agda 2020] and Lean [Ullrich and de Moura 2020], stuck Klister macros need not re-run from scratch when unstuck.

Since different tasks process different parts of the syntax tree, our goal of ensuring determinism regardless of the order in which syntax tree is traversed reduces to ensuring determinism regardless of the order in which the tasks are executed. Thus, whenever there are several tasks which are no longer blocked on any dependency, their order of execution must not affect the end result.

Both example uses of `const*` result in equivalent behavior: in the original version, `const*` does not get stuck, as the outermost type constructor `->` is already known. In the refactored version, `const*` now gets stuck on the unification variable that represents the type, so `returns-unit` is bound to a currently-unknown expression. When type-checking the body of the `let`, `returns-unit`'s type is fixed by the annotation. Then, `const*` gets unstuck, it produces a function, and the type checker then validates that this function indeed has type `(-> Bool Bool Bool Bool Unit)`.

If two macros are blocked on type information that would be provided by the other, then no progress can be made. When the task set is non-empty and all tasks are stuck, type checking and macro expansion are terminated and an error is signalled. We view this situation as being analogous to an underdetermined unification problem in a system like Agda, and expect users to resolve it by adding annotations.

4 Future work

Klister is in the early phases of implementation, and we have not yet developed a large corpus of examples. We aim to discover the smallest extensions necessary to implement features such as type classes [Wadler and Blott 1989] as libraries, guided by the principle of macro expansion leading to monotonic increases program definedness, and of macros' queries about their environment coming from threshold read operations in the style of LVars [Kuper and Newton 2013].

Proven designs from Racket, like access to compile-time value bindings [Flatt et al. 2012], are not always immediately adaptable to a typed system. We intend to explore alternatives to these features that accomplish the same goals within our constraints. Additionally, we hope to implement a version of `local-expand` that can be used recursively without a quadratic overhead by providing an Edinburgh LCF-style interface [Milner 1979] to the results of expansion instead of treating core terms as if they were user syntax.

Finally, we hope to explore richer type systems than Hindley-Milner, beginning with extensions to higher-kinded polymorphism. The Lean team has already implemented hygienic macros for the upcoming Lean 4 prover [Ullrich and de Moura 2020], and we hope to build on their successes with a design oriented towards dependently typed programming rather than proof automation and mathematical notations.

[1] `let`-generalization normally prevents type information from flowing from the body to the definition, but this `let` macro does not generalize types.

References

Agda 2020. Agda's Documentation. https://agda.readthedocs.io/en/v2.6.1/

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2020. Dependent Type Systems as Macros. *Proc. ACM Program. Lang.* 4, POPL, Article 3 (Jan. 2020), 29 pages. https://doi.org/10.1145/3371071

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems as Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 694705. https://doi.org/10.1145/3009837.3009886

Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. 2015. Elaboration in Dependent Type Theory. arXiv:cs.LO/1505.04324

Matthew Flatt. 2016. Binding As Sets of Scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 705–717. https://doi.org/10.1145/2837614.2837620

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that Work Together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming* 22, 2 (2012), 181216. https://doi.org/10.1017/S0956796812000093

Bastiaan Heeren. 2005. *Top Quality Type Error Messages*. Ph.D. Dissertation. Utrecht University.

Richard Kelsey, William Clinger, and Jonathan Rees (eds.). 1998. Revised^5 Report on the Algorithmic Language Scheme. *Higher Order and Symbolic Computation* 11, 1 (August 1998).

Lindsey Kuper and Ryan R Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. 71–84.

Robin Milner. 1979. LCF: A way of doing proofs with a machine. In *Mathematical Foundations of Computer Science 1979*, Jiří Bečvář (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–159.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Sebastian Ullrich and Leonardo de Moura. 2020. Beyond Notations: Hygienic Macro Expansion forTheorem Proving Languages. arXiv:cs.LO/2001.10490

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 60–76.