Soft Typing

Ben Greenman, HOPL 2017

Abstract

A *soft type checker* for a language L analyzes L programs and inserts casts to ensure runtime type safety. A soft type checker does not reject programs as "ill-typed", nor does it require annotations that would not appear in typical L programs.

Soft typing: An approach to type checking for dynamically typed languages

```
@phdthesis{f-thesis-1991,
  author = {Mike Fagan},
  title = {Soft typing: An approach to type checking for
           dynamically typed languages},
  school = {Rice University},
  year = {1991}
}
```

Defines a soft type system as a source-to-source translator that:

- does not reject any syntactically-correct programs,
- assures the run-time safety of its output via inserted assertions,
- infers and checks types "unobtrusively".

To clarify "unobtrusive", Fagan defines *minimal text* and *minimal failure* principles:

Minimal Text Principle ... the type checking system should function in the absence of programmer-supplied type declarations.

Minimal Failure Principle The checker must pass "a large fraction" of dynamic programs that cannot be verified.

In this spirit, Fagan defines a type algebra with untagged unions and recursive types, along with a set of type inference rules that derive a valid typing for any program written in a small functional language.

Significance: this dissertation introduced soft typing. The idea that a type checker could guarantee the *safety* of programs it could not prove type-correct was novel.

Typechecking records and variants in a natural extension of ML

```
@inproceedings{r-popl-1989,
  author = {Didier R\'{e}my},
  title = {Typechecking records and variants in a natural
           extension of ML},
  booktitle = {POPL},
  pages = {77 -- 88},
  year = {1989}
}
```

Rémy introduces a technique for encoding structural subtyping on record types through polymorphism. The key idea is to represent record types as a sequence of labeled *and flagged* types. Flags indicate whether a given label is present or absent. By being polymorphic over the flags it accepts, a function can, e.g., accept records with more present fields that it explicitly accesses.

Significance (to soft typing): introduced the encoding that Fagan and Wright later used.

Practical Soft Typing

```
@phdthesis{w-thesis-1994,
  author = {Andrew K. Wright},
  title = {Practical Soft Typing},
  school = {Rice University},
  year = {1994}
}
```

Adapts Fagan's soft type system (for a small functional language) to R4RS Scheme, both in theory and with an implementation. On the theoretical side, Wright's type system represents types more compactly and accomodates features including user-defined types, mutable state (with strong updates), `call/cc`, and recursive definitions. On the practical side, Wright demonstrates that the implementation can infer usefully-precise types for small programs and is often able to improve programs' performance by removing runtime checks.

Significance: Wright's dissertation describes the first (and possibly, only) implementation of soft typing for a widely-used language.

Soft Typing with Conditional Types

```
@inproceedings{awl-popl-1994,
  author = {Alexander Aiken
            and Edward L. Wimmers
            and T. K. Lakshman},
  title = {Soft Typing with Conditional Types},
  booktitle = {POPL},
  pages = {163 -- 173},
  year = {1994}
}
```

Describes a soft type system where types are sets and type inference builds a collection of set inclusion constraints. On one hand, this approach infers precise types for unannotated programs and the type system can model control flow with set-union and set-intersection operations. On the other hand, a program can generate a very large collection of constraints and the authors' constraint solving algorithm takes exponential time in the worst case. The paper reports low time and memory overhead (at most 30 seconds and 5 MB) on programs with "hundreds of lines", but it is not clear whether the technique will work at scale.

Significance: this work explores soft typing with explicit constraints, as opposed to unification over equations with slack variables. The *conditional types* later influenced Sam Tobin-Hochstadt's work on occurrence typing.

Quasi-Static Typing

```
@inproceedings{t-popl-1990,
  author = {Satish Thatte},
  title = {Quasi-Static Typing},
  booktitle = {POPL},
  pages = {367 -- 381},
  year = {1990}
}
```

Thatte's quasi-static type system is a variant of the simply-typed lambda calculus with implicit, unrestricted dynamic typing. In particular, the type system includes a "Dynamic" type representing a lack of type information. Programmers may use "Dynamic" as a type annotation; the quasi-static type checker implicitly adds casts to and from the dynamic type. Additionally, Thatte defines a "plausibility checker" that removes redundant casts and replaces impossible casts (e.g., from Int to $(\mathsf{Int} \to \mathsf{Int})$) with an error term.

Significance: Thatte's idea that "possible dynamic typing as a property should be inherited" was novel, and appears later in Siek's work on gradual typing.

Global Tagging Optimization by Type Inference

```
@inproceedings{h-lfp-1992,
  author = {Fritz Henglein},
  title = {Global Tagging Optimization by Type Inference},
  booktitle = {LFP},
  pages = {205 -- 215},
  year = {1992}
}
```

Dynamically typed programs ensure type safety at runtime via implicit, first-order casts. For some of these casts, one can show that they never fail because they never receive ill-typed data. Henglein does so with a type inference algorithm. The algorithm makes these casts (implied by data destructors) and first-order datatypes (implied by data constructors) explicit. Together, these explicit casts describe a system of constraints that Henglein solves in almost-linear time to identify unnecessary casts.

Significance: the paper demonstrates that a first-order type system designed to remove first-order casts can quickly remove a number of casts in Scheme programs.