Gradual Singleton Kinds

Johnson He[1]
[1]Computer Science and Engineering, Univeristy of Michigan, Ann Arbor, Michigan

Introduction

"Gradual typing is a [lambda calculus] type system… that allows parts of a program to be dynamically typed and other parts to be statically typed. The programmer controls which parts are which by either leaving out type annotations or by adding them in"[1] (Siek, 2014). Singleton kinds are an extension to a non-dependent kind system to encode definitional type equivalence. Singleton kinds are used to handle type variables and definitions in typed lambda calculi in a principled manner. This SURE supported work aims to bring gradual typing and singleton kinds together for incorporation in Hazel, a live programming language developed at Michigan's FP Lab, headed by Prof. Cyrus Omar, who also advised this project.

Motivating Example

Example of a Hazel program that should become runnable:

```
type T a = a * a in
let foo : T Int = ((|false|), 5) in
(\x : (|T|).
  (\y : T Int.
    case y
    | (l, r) => r
    end
  ) x) foo
```

Result: 5
`x` is not ascribed at a proper type, thus `T` should be placed in a type hole. `false` is not of expected type `Int`, thus should be placed in a term hole. `foo` is cast from `T Int` to `(|T|)` to `T Int`. There should be no failed casts encountered during evaluation.

As we have seen, Hazel is a gradually typed language[4]. Thus, enabling type definitions requires integrating singleton kinding with gradual typing.

Objectives

1. Lay out declarative kinding rules– formalizes which judgments should be provable (eg what is a well formed type?)
2. Prove expected properties of the declarative rules– ensures declarative rules behave as we wish (eg whenever a type is well formed at a kind, the kind itself is well formed)
3. Prove admissiblity of absent "natural" rules– declarative rules we may have wanted are absent; need to prove these rules are already derivable with the current rules
4. Develop algorithmic kinding rules and prove soundness and completeness with respect to the declarative kinding rules– the declarative rules as is are not "codeable" (eg many rules going bottom up require guessing arbitrary premisses), thus we create a second set of rules that are algorithmic and show it is sound and complete with respect to the earlier (well behaved) declarative rules
5. Implement the algorithmic kinding rules in Reason (a OCaml like language with a bent for web browsers) and patch Hazel with support for parameterized type definitions to support the motivating example

6? With the support of the above, develop the ML module theory for Hazel-- originally my objective for the summer, but required the development of higher order kinds

Results and Future Work

The first several weeks were spent digesting the central literature on singleton kinds ([2] [3] [6] [7]) and gradual subtyping ([5] [8]), as well as strengthening my understanding of Hazel's theory ([4]). No previous work on singleton kinds I am aware of has been with a system with gradual typing. A notable technical difficulty was the decision to adopt higher order (HO) singletons (also known as labeled singletons[7]) as primitive, as in Aspinall[2], rather than syntactically defining HO singletons to primitive non-kind-predicative singletons, as in Stone and Harper[7]. This was necessitated by the (new) presence of kind holes (the gradual 'Unknown' type of terms lifted to the kind and type level), which also had the consequence that, unlike Aspinall, type equality could not be syntactically defined to singleton kinding, due to resultant undesirable interaction with consistent subkinding[8] and kind holes.
Intuitively, non-kind-predicative singletons are "the kind of all proper types equivalent to [given type]", while HO singletons are the extension to "the kind of all types equivalent to [given type], where [given type] has kind [given kind]".

The declarative rules have mostly been ironed out, but modifications still arise as I work through objectives 2 and 3. The difficulty with the metatheory stems from the rampant mutual definition of the rules, making inductive hypotheses rather unwieldy. As singletons are inherently circular, I left out as many rules as possible. Therefore, the admissibility of many expected "natural" rules still needs to be shown.
Further in the future, algorithmic rules need to be developed both so that the judgments can be decided in code, but also because some properties of the declarative system are very difficult to prove directly. It can be easier to prove the analogous properties about the algorithmic system and use soundness and completeness to show it for the declarative system in a somewhat backwards fashion. This is done in Stone, 2000 ([6]). For example, this is my plan for showing that principal kinds are indeed principal (that the principal kind of any given type is always a consistent subkind of any kind that type is well formed against).

Conclusions

I spent several weeks this summer learning the theory of singleton kinds, gradual typing, and Hazel. Being the first time I have seriously studied programming language theory, the work was difficult, but the experience was but worthwhile. The inherent circularity of singletons makes the metatheory of a gradually typed language with singleton kinds and type holes tricky, but the payoff is wellfounded machinery that supports type variables and definitions, as well as related concepts like ML modules. So far, declarative rules have been developed that formalize the required machinery, but the metatheory work is still ongoing. I will be continuing this work indefinitely while I am a student.

Acknowledgement

SURE funding enabled me to work on this project full time over the summer, for which I am much oblidged. This project is an extention of Brandon Kase's (bkase.dev) work on singleton kinds without type constructors. Many thanks to Prof. Omar for heading the genial FP Lab and insight and assistance on technical matters.
Lastly, I am indebted to Prof. Richmond Thomason for introducing me to mathematical logic in his intimately taught PHIL 414.

References

[1] https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/
[2] Aspinall, David. 1995. Subtyping with Singleton Types. Proc. Computer Science Logic, CSL'94, Kazimierz, Poland. LNCS 933, p.1-15.
[3] Harper, Robert. *Practical Foundations for Programming Languages*, Second Edition. Cambridge University Press, 2016.
[4] Omar, Cyrus, Voysey, Ian, Chugh, Ravi, Mathew, Hammer A.. 2019. Live Functional Programming with Typed Holes. PACMPL 3, POPL (2019). https://doi.org/10.1145/3290327
[5] Siek, Jeremy, Taha, Walid. 2007. Gradual Typing for Objects. In: Ernst E. (eds) ECOOP 2007 – Object-Oriented Programming. ECOOP 2007. Lecture Notes in Computer Science, vol 4609. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73589-2_2
[6] Stone, C.A. (2000). *Singleton Kinds and Singleton Types*. [Doctoral Dissertation, Carnegie Mellon University].
[7] Stone, Christopher, Harper, Robert. 2006. Extensional Equivalence and Singleton Types. ACM Trans. Comput. Log., 7(4):676-722.
[8] Xie, Ningning, Bi, Xuan, Oliveira, Bruno C. D. S., Schrijvers, Tom. 2019. Consistent Subtyping for All. ACM Trans. Program. Lang. Syst. 42, 1, Article 2 (January 2020), 79 pages. DOI:https://doi.org/10.1145/3310339

SURE SUMMER UNDERGRADUATE RESEARCH IN ENGINEERING
UNIVERSITY OF MICHIGAN

[Poster Code ###]

Introduction

“Gradual typing is a [lambda calculus] type system… that allows parts of a program to be dynamically typed and other parts to be statically typed. The programmer controls which parts are which by either leaving out type annotations or by adding them in”[1] (Siek, 2014). Singleton kinds are an extension to a non-dependent kind system to encode definitional type equivalence. Singleton kinds are used to handle type variables and definitions in typed lambda calculi in a principled manner. This SURE supported work aims to bring gradual typing and singleton kinds together for incorporation in Hazel, a live programming language developed at Michigan’s FP Lab, headed by Prof. Cyrus Omar, who also advised this project.

Motivating Example

Example of a Hazel program that should become runnable:

```
type T a = a * a in
let foo : T Int = ((|false|), 5) in
(\x : (|T|).
  (\y : T Int.
    case y
    | (l, r) => r
    end
  ) x) foo
```

Result: 5
\`x\` is not ascribed at a proper type, thus \`T\` should be placed in a type hole. \`false\` is not of expected type \`Int\`, thus should be placed in a term hole. \`foo\` is cast from \`T Int\` to \`(|T|)\` to \`T Int\`. There should be no failed casts encountered during evaluation.

As we have seen, Hazel is a gradually typed language[4]. Thus, enabling type definitions requires integrating singleton kinding with gradual typing.

Objectives

1. Lay out declarative kinding rules– formalizes which judgments should be provable (eg what is a well formed type?)
2. Prove expected properties of the declarative rules– ensures declarative rules behave as we wish (eg whenever a type is well formed at a kind, the kind itself is well formed)
3. Prove admissiblity of absent “natural” rules– declarative rules we may have wanted are absent; need to prove these rules are already derivable with the current rules
4. Develop algorithmic kinding rules and prove soundness and completeness with respect to the declarative kinding rules– the declarative rules as is are not “codeable” (eg many rules going bottom up require guessing arbitrary premisses), thus we create a second set of rules that are algorithmic and show it is sound and complete with respect to the earlier (well behaved) declarative rules
5. Implement the algorithmic kinding rules in Reason (a OCaml like language with a bent for web browsers) and patch Hazel with support for parameterized type definitions to support the motivating example

6? With the support of the above, develop the ML module theory for Hazel-- originally my objective for the summer, but required the development of higher order kinds

Results and Future Work

The first several weeks were spent digesting the central literature on singleton kinds ([2] [3] [6] [7]) and gradual subtyping ([5] [8]), as well as strengthening my understanding of Hazel's theory ([4]). No previous work on singleton kinds I am aware of has been with a system with gradual typing. A notable technical difficulty was the decision to adopt higher order (HO) singletons (also known as labeled singletons[7]) as primitive, as in Aspinall[2], rather than syntactically defining HO singletons to primitive non-kind-predicative singletons, as in Stone and Harper[7]. This was necessitated by the (new) presence of kind holes (the gradual 'Unknown' type of terms lifted to the kind and type level), which also had the consequence that, unlike Aspinall, type equality could not be syntactically defined to singleton kinding, due to resultant undesirable interaction with consistent subkinding[8] and kind holes.

Intuitively, non-kind-predicative singletons are "the kind of all proper types equivalent to [given type]", while HO singletons are the extension to "the kind of all types equivalent to [given type], where [given type] has kind [given kind]".

The declarative rules have mostly been ironed out, but modifications still arise as I work through objectives 2 and 3. The difficulty with the metatheory stems from the rampant mutual definition of the rules, making inductive hypotheses rather unwieldy. As singletons are inherently circular, I left out as many rules as possible. Therefore, the admissibility of many expected “natural” rules still needs to be shown.

Further in the future, algorithmic rules need to be developed both so that the judgments can be decided in code, but also because some properties of the declarative system are very difficult to prove directly. It can be easier to prove the analogous properties about the algorithmic system and use soundness and completeness to show it for the declarative system in a somewhat backwards fashion. This is done in Stone, 2000 ([6]). For example, this is my plan for showing that principal kinds are indeed principal (that the principal kind of any given type is always a consistent subkind of any kind that type is well formed against).

Conclusions

I spent several weeks this summer learning the theory of singleton kinds, gradual typing, and Hazel. Being the first time I have seriously studied programming language theory, the work was difficult, but the experience was but worthwhile. The inherent circularity of singletons makes the metatheory of a gradually typed language with singleton kinds and type holes tricky, but the payoff is wellfounded machinery that supports type variables and definitions, as well as related concepts like ML modules. So far, declarative rules have been developed that formalize the required machinery, but the metatheory work is still ongoing. I will be continuing this work indefinitely while I am a student.

Acknowledgement

SURE funding enabled me to work on this project full time over the summer, for which I am much oblidged. This project is an extention of Brandon Kase's (bkase.dev) work on singleton kinds without type constructors. Many thanks to Prof. Omar for heading the genial FP Lab and insight and assistance on technical matters.
Lastly, I am indebted to Prof. Richmond Thomason for introducing me to mathematical logic in his intimately taught PHIL 414.

References

[1] https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/
[2] Aspinall, David. 1995. Subtyping with Singleton Types. Proc. Computer Science Logic, CSL'94, Kazimierz, Poland. LNCS 933, p.1-15.
[3] Harper, Robert. *Practical Foundations for Programming Languages*, Second Edition. Cambridge University Press, 2016.
[4] Omar, Cyrus, Voysey, Ian, Chugh, Ravi, Mathew, Hammer A.. 2019. Live Functional Programming with Typed Holes. PACMPL 3, POPL (2019). https://doi.org/10.1145/3290327
[5] Siek, Jeremy, Taha, Walid. 2007. Gradual Typing for Objects. In: Ernst E. (eds) ECOOP 2007 – Object-Oriented Programming. ECOOP 2007. Lecture Notes in Computer Science, vol 4609. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73589-2_2
[6] Stone, C.A. (2000). *Singleton Kinds and Singleton Types*. [Doctoral Dissertation, Carnegie Mellon University].
[7] Stone, Christopher, Harper, Robert. 2006. Extensional Equivalence and Singleton Types. ACM Trans. Comput. Log., 7(4):676-722.
[8] Xie, Ningning, Bi, Xuan, Oliveira, Bruno C. D. S., Schrijvers, Tom. 2019. Consistent Subtyping for All. ACM Trans. Program. Lang. Syst. 42, 1, Article 2 (January 2020), 79 pages. DOI:https://doi.org/10.1145/3310339