A note on this paper draft:

This paper, originally written in 2014, presents an approach for reducing the quantity and size of type information in intermediate languages of a type-directed compiler. The technique presented works, and succeeds at that goal, and I believe it is a technique of interest. It is also accompanied by what I believe is an interesting proof that reconstructability of types is preserved by our closure conversion algorithm.

The snag is that the overarching goal of reducing the size of type information is to improve the *speed* of the compiler. I did not originally collect speed comparision numbers because producing apples-to-apples comparision numbers would have required duplicating much of the compiler with similar passes that preserved types in a more traditional way.

During review, we (rightly!) received pushback about this, and in 2016 I eventually sucked it up and built the passes needed to do a comparsion. Unfortunately, the results were that, when including the cost of reconstructing types at the end, the "forgetful" approach was consistently slower, often by a factor of two.

This is not necessarily a nail in the coffin of the idea, as there may well be substantial room to optimize, but it does not feel very promising to me anymore. Since this came after my main focus had moved on to other research, I didn't investigate further optimization potential. Since I felt that the prospects of publishing a paper about how I wrote a slow compiler were slim, I let the whole thing drop.

Because the compiler frontend contains solution code for a class that my co-author teaches, I haven't released the code, but I can probably share it upon request.

-Michael J. Sullivan

Forgetful Type-Directed Compilation

Reducing the overhead of type information through type reconstruction

Michael J. Sullivan Karl Crary

Carnegie Mellon University

{mjsulliv, crary}@cs.cmu.edu

Abstract

Type-preserving compilation, in which type information is preserved in the intermediate languages of a compiler, has a number of benefits, such as enabling generation of certified code, making it easier to catch compiler bugs, and facilitating type-directed optimization. However, type-preserving compilation is tricky to implement efficiently, as the manipulation of type information can become very costly, and many techniques rely on novel and complex type theory.

We present forgetful type-directed compilation, an alternate approach that focuses on vastly reducing the amount of type information rather than managing the cost of manipulating it. This is accomplished by removing much of the type information while always maintaining the property that we can recover the types through type reconstruction. Types are eliminated through a "type forgetting" algorithm and through the construction of translations that avoid introducing many new annotations. As implemented in our prototype compiler, forgetful type-directed compilation reduces the amount of type information by more than 95% on our test cases.

1. Introduction

Type-preserving compilation, in which type information is preserved in the intermediate languages of a compiler, has a number of benefits. It enables the generation of certified code [7], helps compiler development by catching bugs, and allows the implementation of type-directed optimizations.

A major difficulty in implementing type-preserving compilation is managing the size of type information. As high-level constructs such as algebraic datatypes, modules, polymorphism, and closures are compiled away into lower level constructs with less structure, more type information is needed in order to properly typecheck the intermediate language programs. This can lead to an explosion of type annotations, causing the cost of managing type information to dominate compilation time. In addition, Shao et al. argue that when compiling ML-like languages with features like polymorphic types and modules it is necessary to represent identical types by the same in-memory structure in order to get acceptable performance; maintaining and taking advantage of this sharing, however, pervasively influences the construction of translation passes. [12].

In forgetful type-directed compilation, a new methodology for implementing type-preserving compilers, we take a radically different approach. While many techniques (including Shao et al.'s) for type-preserving compilation seek to manage the cost of manipulating types, we instead attempt to *eliminate* most of the type information. In forgetful type-directed compilation, we "forget" many of the type annotations that appear in intermediate language programs and avoid introducing new ones during translations while maintaining the ability to recover all of the type information using type reconstruction. Additionally, while many techniques to manage the cost of handling types rely on novel intermediate languages with clever but sophisticated type theory, forgetful type-directed compilation does not; we use fairly conventional typed lambda calculi as our intermediate languages.

The main challenge is that type reconstruction for our intermediate languages is quite difficult (in fact, undecidable). Reconstruction is done using an algorithm based on higher-order pattern unification [11], and can correctly reconstruct a subset of well-typed programs while failing on others. We then maintain the invariant that at every stage during compilation, we can reproduce all of the elided type annotations in the program. One of the key insights of forgetful type-directed compilation is that we can restrict ourselves to the subset of programs that we can properly reconstruct without actually having a theory that tells us which programs fall into that subset.

The combination of the type forgetting algorithm and translations that do not introduce new type annotations is very effective at reducing the amount of type information: in our test cases, we reduce the amount of type information by more than 95%, more than is saved by ensuring physical sharing of identical types.

The main contributions of this paper are:

- We present a syntax-directed typed closure conversion algorithm that introduces very few new type annotations (Section 4.1), and prove that the translation preserves type reconstructability (Section 4.2). We focus on closure conversion because we feel it is the trickiest of the translations we implemented (in terms of reconstructability) and because it is a good showcase of avoiding the introduction of new annotations. We believe that the proof itself is an interesting contribution, as we are not aware of any previous results of this sort.
- While much of the benefit of forgetful type-directed compilation comes from not adding new annotations during compilation, we can also reduce the number of annotations at the beginning of compilation. We present an algorithm for removing type annotations from a program such that we can still reconstruct the types (Section 5). The algorithm is largely indepen-

dent of the structure of the language; the main requirement is that inference be expressible as a higher-order unification problem.

- We discuss our experience using forgetful type-directed compilation in an in-progress prototype compiler for Standard ML (Section 6).

2. Overview

We now present an informal introduction to our techniques in the context of a typed lambda calculus similar to what might be used as an intermediate language.

As a somewhat contrived example of a term with erased type annotations, consider the term (where $\circ$ represents an erased type annotation; not necessarily all the same one):

$$\begin{array}{l}\mathsf{let}\, f = \lambda x{:}\circ\,.\,\lambda y{:}\circ\,.\,\langle x, y\rangle\ \mathsf{in}\\ \mathsf{let}\, g = \lambda n{:}\circ\,.\, f\, n\, \mathsf{true}\ \mathsf{in}\\ \mathsf{let}\, h = \lambda b{:}\circ\,.\, f\, 1\, b\ \mathsf{in}\\ \langle g, h\rangle\end{array}$$

All of the the annotations in this term are reconstructable. The function g passes a boolean constant as f's second argument and h passes an integer as its first, which allows us to determine that the type annotation for x is int and y is bool. From that we can determine the annotations on n and b as well. (This example does not use any universal or existential types, and so can be easily reconstructed just using first-order unification.)

2.1 Closure Conversion

To demonstrate how type forgetting is used in the compiler, we present a forgetful closure-conversion translation. Closure conversion is an important part of compiling functional programming languages and poses some problems for type-directed compilers.

In closure conversion, we rewrite the program so that lambdas are closed terms. To do this, we modify each lambda to take, as an additional argument, a tuple containing its environment of free variables; we then pair this rewritten lambda with an environment tuple of the variables.

For example, the following simple function with free variables n and s

$$\lambda x{:}\mathsf{int}.\, x < (\mathsf{strlen}\, s) + n$$

could be rewritten to

$$\begin{array}{l}\langle(\lambda p{:}\mathsf{int} \times (\mathsf{string} \times \mathsf{int}).\, \mathsf{let}\, \langle x, \langle y, z\rangle\rangle = p\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\qquad x < \mathsf{strlen}\, y + z),\\ \langle s, n\rangle\rangle\end{array}$$

This, however, rewrites a lambda of type $\mathsf{int} \to \mathsf{bool}$ into one of type $\mathsf{int} \times (\mathsf{string} \times \mathsf{int}) \to \mathsf{bool}$. This is problematic because it implies that the translated type of a function depends on the context it appears in; to make the translation work out, we follow Minamide et al. [6] in wrapping functions in an existential type, rewriting functions of type $A \to B$ into existential packages of type $\exists\alpha_e.(A \times \alpha_e \to B) \times \alpha_e$. The example above would then be translated into

$$\begin{array}{l}\mathsf{pack}[\mathsf{string} \times \mathsf{int},\\ \quad\langle(\lambda p{:}\mathsf{int} \times (\mathsf{string} \times \mathsf{int}).\, \mathsf{let}\, \langle x, \langle y, z\rangle\rangle = p\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\qquad x < \mathsf{strlen}\, y + z),\\ \quad\langle s, n\rangle\rangle]\\ \mathsf{as}\ \exists\alpha_e.(\mathsf{int} \times \alpha_e \to \mathsf{bool}) \times \alpha_e\end{array}$$

This works, but we have introduced a number of new type annotations that were not present in the original. The lambda and the pack each contain an annotation for the environment, which doesn't correspond to anything in the source program. We also now annotate the return type of the function, when previously it was determined by the function body. In forgetful type-directed compilation, we want to produce something like

$$\begin{array}{l}\mathsf{pack}[\circ,\\ \quad\langle(\lambda p{:}\circ\,.\,\mathsf{let}\, \langle x, \langle y, z\rangle\rangle = p\ \mathsf{in}\\ \qquad\qquad\qquad\qquad x < \mathsf{strlen}\, y + z),\\ \quad\langle s, n\rangle\rangle]\\ \mathsf{as}\ \exists\alpha_e.(\mathsf{int} \times \alpha_e \to \circ) \times \alpha_e\end{array}$$

where $\circ$ represents an annotation that we have eliminated but wish to be able to reconstruct.

Intuitively, it seems like we should be able to reconstruct all of the elided type annotations. We know that $\langle s, n\rangle$ has the type $\mathsf{string} \times \mathsf{int}$; noting that this lines up with the final α_e in the annotation on the pack should tell us that $\mathsf{string} \times \mathsf{int}$ must be the type being packed. Knowing this, from the annotation on the pack we can deduce that the lambda must have the argument type $\mathsf{int} \times (\mathsf{string} \times \mathsf{int})$. From that we can determine that the result type of the function must be bool, which lets us reconstruct the last annotation. In Section 4.1 we will formalize our description of closure conversion and prove that this intuition that it preserves reconstructability is correct.

2.2 Type Forgetting

While much of the benefit of type-erased intermediate languages comes from being able to elide new type annotations introduced by translations, it is also helpful to remove unnecessary annotations at an early stage of compilation. To do this, we use a simple "type forgetting" algorithm. The key insight of the algorithm is that we can determine what is inferrable by trying to infer the program with all the annotations removed and seeing what we can recover. We then find an annotation we could not infer, add it back in, re-solve our constraints with the additional information, and repeat until all of the annotations can either be inferred or have been added back in.

This strategy is very general — it can be applied to any language for which type inference is expressible as a higher-order unification problem. It is also quite brute force: it does not take advantage of any knowledge or theory about what type annotations might be necessary.

3. Language and Type Reconstruction

We formalize forgetful type-directed compilation in the context of System F_ω, the higher-order polymorphic lambda calculus. Our variant of the language has higher kinds, universal and existential types, and n-ary products. Type inference on our intermediate languages is done by generating higher-order unification problems. To that end, the representation of type constructors is chosen to facilitate higher-order unification.

In this section, we discuss our representation of type constructors, the formalization of and algorithm for higher-order unification we use, and how we generate higher-order unification problems to implement type inference. The material in this section is not novel, but is included to make the presentation self-contained.

3.1 The Type System

The type constructor level of the language is a simply typed lambda calculus. The constructor level does not have any special forms for the types of the language; instead it uses type constructor constants to form types. For example, $(\to)$ is a constant of kind $\mathsf{T} \to \mathsf{T} \to \mathsf{T}$ and $A \to B$ is expressed as $(\to)\, A\, B$. Likewise, there are constants $(\forall_\kappa)$ of kind $(\kappa \to \mathsf{T}) \to \mathsf{T}$ for every kind κ; thus $\forall\alpha{:}\kappa.\, A$ is expressed as $(\forall_\kappa)(\lambda\alpha.A)$.

Two notable features of the constructor level of our language are chosen to match the formalism of the unification algorithm that we use. First, we restrict attention to type constructors in *canonical*

form, which are constructors that are fully η-expanded and have no β-redexes. We take advantage of the observation by Watkins et al. [14] that a *hereditary substitution* operation can be defined that reduces any β-redexes that would be introduced by a regular substitution.

Second, we take advantage of *contextual modal type theory's* [8] notion of modal variables to represent metavariables (also called unification variables). A modal variable X acts as a placeholder where some other type, possibly containing free variables, can be substituted. Each modal variable is associated with a type context Δ that indicates what variables can occur free in it. Modal variables and the type contexts associated with them are tracked in a modal context Ξ. When modal variables appear in types, it is always in the form $X[\sigma]$, where σ is a substitution that maps from the ambient context to the context of X. Substitutions can contain both type-for-variable substitutions and variable-for-variable renamings; renamings will be used to define the key notion of *pattern substitutions*. We will write id_Δ to mean the identity substitution for all variables that appear in Δ. To reduce clutter, we will usually simply write X to mean $X[id_\Delta]$ when the surrounding context is Δ.

The syntax is as follows:

$$\begin{array}{rlll}
\text{Kinds} & \kappa & ::= & \mathsf{T} \mid \kappa \to \kappa \\
\text{Normal Types} & A, B, C & ::= & \lambda\alpha.A \mid R \\
\text{Atomic Types} & R & ::= & H \mid R\,A \mid X[\sigma] \\
\text{Heads} & H & ::= & \alpha \mid c \\
\\
\text{Substitutions} & \sigma, s & ::= & \cdot \mid \sigma, (\alpha/\beta) \mid \sigma, (A/\alpha) \\
\text{Type Contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
\text{Modal Contexts} & \Xi & ::= & \cdot \mid \Xi, X :: (\Delta \vdash \mathsf{T})
\end{array}$$

Types are checked under a type context Δ and a modal context Ξ. The main kinding judgments are

$$\Xi; \Delta \vdash A \Leftarrow \kappa \qquad \Xi; \Delta \vdash R \Rightarrow \kappa$$

"A checks against kind κ" and "R synthesizes kind κ". To reduce clutter, we generally elide the Ξ.

We present the kinding rules below. One key rule is the checking rule for the case when an atomic type appears as a normal type: this is allowed, but only at base kind. This enforces eta-long form by requiring a constant or variable to be fully applied before it can be included into a normal type.

The other key rule is the rule for checking modal variables. This rule depends on the substitution checking judgement $\Delta \vdash \sigma : \Delta'$ which states that the substitution σ will map terms under the context Δ' to terms under the context Δ. A use of a modal variable $X[\sigma]$ is well typed if X exists in the modal context Ξ at some context Δ' and σ maps from Δ' to the ambient context Δ.

$$\frac{X :: (\Delta' \vdash \mathsf{T}) \in \Xi \quad \Delta \vdash \sigma : \Delta'}{\Xi; \Delta \vdash X[\sigma] \Rightarrow \mathsf{T}} \qquad \frac{\Delta \vdash R \Rightarrow \mathsf{T}}{\Delta \vdash R \Leftarrow \mathsf{T}}$$

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha \Rightarrow \kappa} \qquad \frac{\Delta \vdash R \Rightarrow \kappa \to \kappa' \quad \Delta \vdash A \Leftarrow \kappa}{\Delta \vdash R\,A \Rightarrow \kappa'}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash A \Leftarrow \kappa_2}{\Delta \vdash \lambda\alpha.A \Leftarrow \kappa_1 \to \kappa_2}$$

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash A \Leftarrow \kappa \quad \Delta \vdash \sigma : \Delta'}{\Delta \vdash \sigma, (A/\alpha) : (\Delta', \alpha{:}\kappa)}$$

$$\frac{\beta{:}\kappa \in \Delta \quad \Delta \vdash \sigma : \Delta'}{\Delta \vdash \sigma, (\beta/\alpha) : (\Delta', \alpha{:}\kappa)}$$

We define the hereditary substitution operation $[B/\alpha]A$ which substitutes the canonical type B for the variable α in a canonical type A, producing a canonical type. To handle the case where α is the head, we define the auxiliary operation $[A \mid B_1, \cdots, B_n]$ which gives the β-reduction of A when applied to a list of arguments.

$$\begin{array}{rcl}
[B/\alpha](\alpha\ A_1 \cdots A_n) & = & [B \mid [B/\alpha]A_1, \ldots, [B/\alpha]A_n] \\
[B/\alpha](H\ A_1 \cdots A_n) & = & H\ ([B/\alpha]A_1) \cdots ([B/\alpha]A_n) \\
 & & \text{where } H \neq \alpha \\
[B/\alpha](\lambda\beta.A) & = & \lambda\beta.[B/\alpha]A \\
[\lambda\alpha.A \mid B_1, B_2, \ldots, B_n] & = & [[B_1/\alpha]A \mid B_2, \ldots, B_n] \\
[R \mid \cdot] & = & R
\end{array}$$

As shown by Watkins [14] using an induction metric reminiscent of cut elimination, because the inputs are well kinded, these functions are total.

In order to handle the modal variable case, we also define $[B/\alpha]\sigma$, which acts on a substitution.

$$\begin{array}{rcll}
[B/\alpha](X[\sigma]) & = & X[[B/\alpha]\sigma] & \\
[B/\alpha](\sigma, (\alpha/\beta)) & = & [B/\alpha]\sigma, (B/\beta) & \\
[B/\alpha](\sigma, (\gamma/\beta)) & = & [B/\alpha]\sigma, (\gamma/\beta) & \text{where } \gamma \neq \alpha \\
[B/\alpha](\sigma, (A/\beta)) & = & [B/\alpha]\sigma, ([B/\alpha]A/\beta) &
\end{array}$$

We also need the operation $[\sigma]A$, which applies each individual substitution in σ to A, and the operation $[R/X]A$, which finds each subterm $X[\sigma]$ of A and replaces it with $[\sigma]R$.

3.2 Unification

We borrow both our formalization of higher-order unification problems and our constraint simplification algorithm for solving it from Reed [11]. Much of the presentation is directly adapted from Reed as well. A unification problem is a system $\exists\,\Xi \Vdash P$ where

$$\begin{array}{rlll}
\text{Equation Sets} & P & ::= & \top \mid P \wedge Q \\
\text{Equations} & Q & ::= & A \doteq A' \mid R \doteq R' \\
 & & & \mid X \doteq R \mid X \leftarrow R
\end{array}$$

Intuitively, $\exists\,\Xi \Vdash P$ asks whether there exist instantiations of the metavariables in Ξ such that all of the equations in P are satisfied. Equation sets are considered up-to the reordering of the equations in them. The equation $X \leftarrow R$ indicates that X has been solved for R and instantiated in the rest of the problem, while in $X \doteq R$, R may contain uses of X, preventing instantiation. When it is unambiguous, we will often write P instead of $\exists\,\Xi \Vdash P$. We freely drop solved variables from unification problems if we are no longer interested in them.

The key difficulty in higher-order unification is in handling constraints of the form $X[\sigma] \doteq R$. In general, it is difficult (undecidable) to directly solve for X. If the substitution had an inverse σ^{-1} such that $[\sigma^{-1}]\sigma = id$, then we could apply the inverse substitution to both sides of the equation to yield $X \doteq [\sigma^{-1}]R$. While substitutions do not, in general, have inverses, there exists a class of them, called *pattern substitutions*, that do.

The key feature of this constraint simplification algorithm for higher-order unification, then, is the inversion of pattern substitutions: a pattern substitution (written ξ) maps each variable in its domain to a *distinct* bound variable. Because each variable is mapped to a different replacement variable, we can invert a pattern substitution simply by reversing all of the mappings. We define the inversion operation ξ_Γ^{-1} such that it contains (y/x) for every $(x/y) \in \xi$ and $(\_/x)$ for any x in Γ not covered by the above. (We extend the syntax of normal types with $\_$, which represents an undefined type that must not appear in a solution.)

We write $\hat{Z}\{A\}$ to indicate a member of some syntactic class Z (in particular, we will use Q) with a hole for a type A to be plugged in.

$$
\begin{array}{rcl}
 & & \text{(we assume that } \Gamma = \Delta; \Psi) \\
V(\Gamma, x, A) & = & A \doteq \Gamma(x) \qquad \text{(provided that } x \in \Gamma) \\
 & & \\
V(\Gamma, \lambda x{:}B.\, e, A) & = & \exists\, Y :: (\Delta \vdash \mathsf{T}) \;\Vdash (A \doteq B \to Y) \wedge V(\Gamma \oplus x{:}B, e, Y) \\
V(\Gamma, e_1\, e_2, A) & = & \exists\, X :: (\Delta \vdash \mathsf{T}) \;\Vdash V(\Gamma, e_1, X \to A) \wedge V(\Gamma, e_2, X) \\
 & & \\
V(\Gamma, \Lambda\alpha{:}\kappa.e, A) & = & \exists\, F :: (\Delta, \alpha{:}\kappa \vdash \mathsf{T}) \;\Vdash (A \doteq (\forall_\kappa)\,(\lambda\alpha.F)) \wedge V(\Gamma \oplus \alpha{:}\kappa, e, F) \\
V(\Gamma, e\,[B]_\kappa, A) & = & \exists\, F :: (\Delta, \alpha{:}\kappa \vdash \mathsf{T}) \;\Vdash (A \doteq F[id_\Delta, (B/\alpha)]) \wedge V(\Gamma, e, (\forall_\kappa)(\lambda\alpha.F)) \\
 & & \\
V(\Gamma, \langle e_1, \ldots\, e_n\rangle, A) & = & \exists\, X_1, \ldots, X_n :: (\Delta \vdash \mathsf{T}) \;\Vdash (A \doteq (\times_n)\, X_1 \ldots X_n) \wedge V(\Gamma, e_1, X_1) \wedge \cdots \wedge V(\Gamma, e_n, X_n) \\
V(\Gamma, \mathsf{let}\, \langle x_1, \ldots\, x_n\rangle = e_1\ \mathsf{in}\ e_2, A) & = & \exists\, X_1, \ldots, X_n :: (\Delta \vdash \mathsf{T}) \;\Vdash \\
 & & V(\Gamma, e_1, (\times_n)\, X_1 \ldots X_n) \wedge \\
 & & V(\Gamma \oplus x_1{:}X_1 \oplus \cdots \oplus x_n{:}X_n, e_2, A)
\end{array}
$$

Figure 1. Type inference

$$
\begin{array}{rcl}
 & & \text{(we assume that } \Gamma = \Delta; \Psi) \\
V(\Delta; \Psi, \hat{\lambda} x{:}B.\, e, A) & = & \exists\, Y :: (\Delta \vdash \mathsf{T}) \;\Vdash (A \doteq B \to Y) \wedge V(\Delta; x{:}B, e, Y) \\
 & & \\
V(\Gamma, \mathsf{let}\, x = e_1\ \mathsf{in}\ e_2, A) & = & \exists\, X :: (\Delta \vdash \mathsf{T}) \;\Vdash V(\Gamma, e_1, X) \wedge V(\Gamma \oplus x{:}X, e_2, A) \\
 & & \\
V(\Gamma, \mathsf{pack}_\kappa[B, e]\, \mathsf{as}\, C, A) & = & \exists\, X :: (\Delta, \alpha{:}\kappa \vdash \mathsf{T}) \;\Vdash \\
 & & (A \doteq C) \wedge (C \doteq (\exists_\kappa)\,(\lambda\alpha.X)) \wedge V(\Gamma, e, X[id_\Delta, (B/\alpha)]) \\
V(\Gamma, \mathsf{unpack}_\kappa(e_1, \alpha.x.e_2), A) & = & \exists\, X :: (\Delta, \alpha{:}\kappa \vdash \mathsf{T}) \;\Vdash V(\Gamma, e_1, (\exists_\kappa)\,(\lambda\alpha.X)) \wedge V(\Gamma \oplus \alpha{:}\kappa \oplus x{:}X, e_2, A) \\
 & & \\
V(\Gamma, e\, \mathsf{as}\, B, A) & = & (B \doteq A) \wedge V(\Gamma, e, B)
\end{array}
$$

Figure 2. Type inference for new target language constructs

The algorithm is specified as a reduction relation $P \mapsto\!\!\!\!\twoheadrightarrow P'$ that reduces a unification problem to a simpler one; the algorithm proceeds by repeatedly applying the reduction rules below until no rules are applicable. Reed [11] proved that this algorithm is correct and terminating. It is correct in the sense that every transition preserves the set of solutions to the unification problem and terminating in that every chain of reductions will eventually reach a state where no rule applies. This final state will either be a solution, or $\bot$, or a stuck state that represents an answer of "maybe".

We present only the rules critical to our present discussion, leaving out rules for the occurs check, for handling constraints of the form $X \doteq X[\xi]$, and for pruning the contexts of metavariables to eliminate underscores from substitutions. Full details are available in Reed [11].

1. **Decomposition.**

$$(\lambda\alpha.A \doteq \lambda\alpha.A') \wedge P \mapsto\!\!\!\!\twoheadrightarrow (A \doteq A') \wedge P$$

$$(H\, A_1 \cdots A_n \doteq H\, A'_1 \cdots A'_n) \wedge P \mapsto\!\!\!\!\twoheadrightarrow (A_1 \doteq A'_1) \wedge \cdots \wedge (A_n \doteq A'_n) \wedge P$$

$$(H\, A_1 \cdots A_n \doteq H'\, A'_1 \cdots A'_n) \wedge P \mapsto\!\!\!\!\twoheadrightarrow \bot \quad (\text{if } H \neq H')$$

2. **Inversion.**

$$(X[\xi] \doteq R) \wedge P \mapsto\!\!\!\!\twoheadrightarrow (X \doteq [\xi^{-1}]R) \wedge P$$

3. **Instantiation.**

$$(X \doteq R) \wedge P \mapsto\!\!\!\!\twoheadrightarrow (X \leftarrow R) \wedge [R/X]P \qquad (\text{if } X \notin FV(R))$$

3.3 Expression Language

The expression syntax of our language is mostly standard. We write Ψ for term contexts and reserve Γ for a pair $\Delta; \Psi$ of a type and term context. We use a pattern matching let as our elimination form for products, as it permits a more elegant definition of closure conversion.

$$
\begin{array}{rcll}
\text{Expressions} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\, e' \mid \\
 & & & \mid \Lambda\alpha{:}\kappa.e \mid e\,[A]_\kappa \\
 & & & \mid \langle e_1, \ldots\, e_n\rangle \\
 & & & \mid \mathsf{let}\, \langle x_1, \ldots\, x_n\rangle = e\ \mathsf{in}\ e' \\
\text{Constants} & c & ::= & (\to) \mid (\times_n) \mid (\forall_\kappa) \mid \cdots \\
\text{Term Contexts} & \Psi & ::= & \cdot \mid \Psi, x{:}A \\
\text{Contexts} & \Gamma & ::= & \Delta; \Psi
\end{array}
$$

We could give a standard set of typing rules for our term language. We elide those and instead present type inference rules for the language as Figure 1. Following Pfenning [10], we define type inference as a function $V(\Gamma, e, A) = P$ over contexts, terms, and expected types that generates a higher-order unification problem. Because type inference relies on renamings, we implicitly eta-

contract types down to variables when constructing substitutions if appropriate.

In the definitions of type inference, we adopt the notational convention that the modal contexts that are output implicitly percolate upwards. That is, $\exists\,\Xi \Vdash (\exists\,\Xi_1 \Vdash P_1) \wedge (\exists\,\Xi_2 \Vdash P_2)$ is synonymous with $\exists\,\Xi, \Xi_1, \Xi_2 \Vdash P_1 \wedge P_2$.

Most of the rules are quite straightforward. A common pattern is that introduction forms create new variables for their subcomponents and unify a type constructed from them with the expected type. Elimination forms construct the type being eliminated and use it as the expected type for the expression being operated on. The trickiest is for a type application $e\,[B]_\kappa$; a new type variable F is created under an extended context, and e is checked against $(\forall_\kappa)(\lambda\alpha.F)$ – that is, against $\forall\alpha.F$. The result type, then, is B substituted for α in F, so we unify A with $F[id_\Delta, (B/\alpha)]$.

In order to omit a type annotation, metavariables are used. For example, in order to write a lambda without annotating its domain, we would write $\lambda x{:}X.\, e$ for a fresh metavariable X[1]. Somewhat unusually, we permit *partial* type annotations: metavariables may occur inside of a type annotation, not just at the top level. This is to allow translation phases freedom to include enough of the annotation to preserve reconstructability without including the entire type. In particular, this is quite important for closure conversion.

4. Translation

4.1 Definition of Closure Conversion

The target language for the translation is the source language extended as follows:

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \cdots \\ & & & \mid \mathsf{let}\, x = e_1\ \mathsf{in}\ e_2 \\ & & & \mid \hat{\lambda} x{:}A.\, e \\ & & & \mid \mathsf{pack}_\kappa[B, e]\,\mathsf{as}\, C \\ & & & \mid \mathsf{unpack}_\kappa(e_1, \alpha.x.e_2) \\ & & & \mid e\,\mathsf{as}\, A \\ \text{Constants} & c & ::= & \cdots \mid (\exists_\kappa) \end{array}$$

where pack and unpack are the introduction and elimination forms for existential types, $\hat{\lambda}$ defines a closed function, and $e\,\mathsf{as}\,A$ is a type annotation that we will need for technical reasons. Type inference rules for these constructs are shown in Figure 2.

Since many of the type annotations may have been removed prior to closure conversion, the closure conversion translation cannot rely on having full typing information. Thus we define it as a syntax-directed [2] translation. Figure 3 defines closure conversion as three syntax-directed functions $\overline{A}, \overline{\sigma}$, and $\overline{e}$. (We will also write $\overline{\Gamma}$ and $\overline{P}$ for the lifting of type conversion over contexts and unification problems.)

Unsurprisingly, most cases are trivial compatibility cases. The action happens in the type translation case for arrows and in the term translation case for application and lambda. Type translation for arrows works by rewriting $(\rightarrow)\,A_1\,A_2$ into the existential package

$$(\exists_\mathsf{T})(\lambda\alpha_e.((\overline{A_1} \times \alpha_e) \rightarrow \overline{A_2}) \times \alpha_e).$$

Translation of application translates the subparts, binds them to variables, then unpacks the closure and applies the function to the argument and the environment. A type annotation is placed on the variable env: this is necessary to preserve type reconstructability.

The translation of lambdas is a little trickier and involves generating variables to fill in annotations. The three variables generated are Z, which is the type of environment of the function, Y, the codomain of the function, and I, the domain of the function. The pack is given an annotation detailed enough to make it clear that the result is a translated function, including whatever annotations on the domain already existed in the source program.

4.2 Correctness of Closure Conversion

The main interesting property that we require of our closure conversion translation (other than correctness) is preservation of type reconstructability: if the input program can be reconstructed by our algorithm, then it can be reconstructed after closure conversion as well. We prove the following theorem:

Preservation of reconstructability: If $V(\Gamma, e, A) \mapsto\!\!\!\!\twoheadrightarrow^* S$, where S is a solved form, then $V(\overline{\Gamma}, \overline{e}, \overline{A}) \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$. [3]

We prove this in two steps: one pertaining to types and one to terms. First we show that solvability of unification problems is preserved under type translation. Second we show that the inference constraints of a translated term reduce to the translation of the inference constraints of the original term.

Preservation of solvability: If $P \mapsto\!\!\!\!\twoheadrightarrow^* S$, where S is a solved form, then $\overline{P} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$.

Proof. By induction over the length of the reduction derivation, case analyzing on the rule used to derive the first step.

For **Decomposition** of lambdas, we observe that by induction, $(\overline{A} \doteq \overline{A'}) \wedge \overline{P} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$. Then, by reapplying **Decomposition** we find $(\lambda\alpha.\overline{A} \doteq \lambda\alpha.\overline{A'}) \wedge \overline{P} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$ which is equivalent to $(\overline{\lambda\alpha.A} \doteq \overline{\lambda\alpha.A'}) \wedge \overline{P} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$.

The case for **Decomposition** of applications where the head is something other than $(\rightarrow)$ is similar.

For **Decomposition** of applications where the head is an arrow, we need to show that $\overline{((\rightarrow)\,A_1\,A_2) \doteq ((\rightarrow)\,B_1\,B_2)} \wedge \overline{P} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$. Expanding the equation we are looking at gives us

$$(\exists_\kappa)(\lambda\alpha.((\overline{A_1}\times\alpha) \rightarrow \overline{A_2})\times\alpha) \doteq (\exists_\kappa)(\lambda\alpha.((\overline{B_1}\times\alpha) \rightarrow \overline{B_2})\times\alpha)$$

Applying **Decomposition** seven times tells us that

$$\overline{P} \wedge \overline{((\rightarrow)\,A_1\,A_2) \doteq ((\rightarrow)\,B_1\,B_2)} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{P} \wedge \overline{A_1} \doteq \overline{B_1} \wedge \overline{A_2} \doteq \overline{B_2}$$

Then we apply induction to finish up this case.

For **Invert**, we first note that since translation is the identity on variables a translated pattern substitution is still a pattern substitution. Furthermore, hereditary substitution commutes with type translation (the induction metric for this proof, like all proofs about hereditary substitution, is complicated—but the proof cases themselves are very simple).

For **Instantiate**, we have that $P \wedge X \doteq A \mapsto\!\!\!\!\rightarrow [A/X]P \wedge X \leftarrow A$ and $[A/X]P \wedge X \leftarrow A \mapsto\!\!\!\!\twoheadrightarrow^* S$. We note that modal substitution commutes with type translation (this is easily established by induction), and so $[\overline{A}/X]\overline{P} = \overline{[A/X]P}$. Then we can reapply **Instantiate** and appeal to induction to show that $\overline{P} \wedge X \doteq \overline{A} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$.

For this case we have that $P \wedge X[\xi] \doteq B \mapsto\!\!\!\!\rightarrow P \wedge X \doteq [\xi^{-1}]B$ and $P \wedge X \doteq [\xi^{-1}]B \mapsto\!\!\!\!\twoheadrightarrow^* S$. By induction we have that $\overline{P \wedge X \doteq [\xi^{-1}]B} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$. From the two observations above, we have that $\overline{[\xi^{-1}]B} = [\overline{\xi}^{-1}]\overline{B}$.

Now we can apply **Invert**, yielding $\overline{P \wedge X[\xi] \doteq B} \mapsto\!\!\!\!\rightarrow \overline{P} \wedge X \doteq \overline{[\xi^{-1}]B}$ and thus $\overline{P \wedge X[\xi] \doteq B} \mapsto\!\!\!\!\twoheadrightarrow^* \overline{S}$.

The cases for the rules we elided are straightforward. □

[1] It is important that an implementation be able to elide the substitutions on metavariables in most cases, or else we will eliminate type annotations only to be swamped by substitution annotations.

[2] Technically the translation also depends on the kind environment, though representation choices in the implementation eliminate this dependency

[3] Technically, $V(\overline{\Gamma}, \overline{e}, \overline{A})$ can step to some extension of $\overline{S}$. We sweep this under the rug and identify equations up to ignoring solved metavariables that were introduced during translation.

$$
\begin{array}{rcl}
\overline{\lambda\alpha.A} & = & \lambda\alpha.\overline{A} \\
\overline{(\to)\,A_1\,A_2} & = & (\exists_{\mathsf{T}})(\lambda\alpha_{env}.((\overline{A_1} \times \alpha_{env}) \to \overline{A_2}) \times \alpha_{env}) \\
\overline{H\,A_1\,\cdots\,A_n} & = & H\ \overline{A_1} \cdots \overline{A_n} \qquad \text{where } H \neq (\to) \\
\overline{X[\sigma]} & = & X[\overline{\sigma}] \\
\overline{(\_)} & = & (\_) \\
\\
\overline{\cdot} & = & \cdot \\
\overline{\sigma, (\alpha/\beta)} & = & \overline{\sigma}, (\alpha/\beta) \\
\overline{\sigma, (M/\alpha)} & = & \overline{\sigma}, (\overline{M}/\beta) \\
\\
\overline{x} & = & x \\
\overline{e_1\,e_2} & = & \mathsf{let}\, x = \overline{e_1}\ \mathsf{in} \\
& & \mathsf{let}\, y = \overline{e_2}\ \mathsf{in} \\
& & \mathsf{unpack}_{\mathsf{T}}\,[\alpha_{env}, clos] = x\ \mathsf{in} \\
& & \mathsf{let}\,\langle f, env\rangle = clos\ \mathsf{in} \\
& & f\,\langle y, env\,\mathsf{as}\,\alpha_{env}\rangle \\
\overline{\lambda x{:}B.\,e} & = & \mathsf{pack}_{\mathsf{T}}[Z, \\
& & \quad \langle(\hat{\lambda} i{:}I.\,\mathsf{let}\,\langle x, env\rangle = i\ \mathsf{in} \\
& & \qquad \mathsf{let}\,\langle x_1, \ldots\, x_n\rangle = env\ \mathsf{in} \\
& & \qquad \overline{e}), \\
& & \quad \langle x_1, \ldots\, x_n\rangle\rangle] \\
& & \mathsf{as}\,(\exists_{\mathsf{T}})(\lambda\alpha_{env}.((\overline{B} \times \alpha_{env}) \to Y[id_{\overline{\alpha}}]) \times \alpha_{env}) \\
& & (\text{where } FV(e) \setminus \{x\} = \{x_1, \ldots, x_n\}, \\
& & \text{and } \overline{\alpha} \text{ are the type variables in scope} \\
& & \text{and } I, Y, Z \text{ are fresh}) \\
\overline{\langle e_1, \ldots\, e_n\rangle} & = & \langle\overline{e_1}, \ldots\, \overline{e_n}\rangle \\
\overline{\mathsf{let}\,\langle x_1, \ldots\, x_n\rangle = e_1\ \mathsf{in}\ e_2} & = & \mathsf{let}\,\langle x_1, \ldots\, x_n\rangle = \overline{e_1}\ \mathsf{in}\ \overline{e_2} \\
\\
\overline{\Lambda\alpha{:}\kappa.e} & = & \Lambda\alpha{:}\kappa.\overline{e} \\
\overline{e\,[B]_\kappa} & = & \overline{e}\,[\overline{B}]_\kappa
\end{array}
$$

Figure 3. Closure conversion

Equivalence of inferred constraints under term translation: For all expressions e, $V(\overline{\Gamma}, \overline{e}, \overline{A}) \mapsto\!\!\!\!\to^* \overline{V(\Gamma, e, A)}$. This theorem essentially states that translation and constraint generation commute modulo reduction of the unification problem. (It is somewhat surprising that the left hand side can reduce directly to the right hand side, instead of just being able to reduce to a common reduct!)

Proof. By induction over the structure of the term e. The proof cases are all fairly straightforward, but the cases for lambda and application are unsurprisingly quite involved. Since the translation of lambda and application are fairly large, the unification problems generated by their translations introduce a number of new constraints that need to be unrolled. Once the constraints are collected, it is a fairly straightforward procedure to decompose constraints and instantiate variables to arrive at something we can apply the inductive hypothesis to.

Application First, we consider the case where $e = e_1\,e_2$. We assume that $\Gamma = \Delta; \cdots$ and we let $\Delta' = \Delta, \alpha_e{:}\mathsf{T}$. The first step is to unroll the definition of $\overline{e_1\,e_2}$ in order to compute the constraints generated by $V(\overline{\Gamma}, \overline{e_1\,e_2}, \overline{A})$:

$$
\begin{array}{l}
\mathsf{let}\, x = \overline{e_1}\ \mathsf{in} \\
\mathsf{let}\, y = \overline{e_2}\ \mathsf{in} \\
\mathsf{unpack}_{\mathsf{T}}\,[\alpha_e, clos] = x\ \mathsf{in} \\
\mathsf{let}\,\langle f, env\rangle = clos\ \mathsf{in} \\
f\,\langle y, env\,\mathsf{as}\,\alpha_e\rangle
\end{array}
$$

Some tedious but straightforward computation tells us that

$$
\begin{array}{rcl}
V(\overline{\Gamma}, \overline{e_1\,e_2}, \overline{A}) & = & \exists\, X, Y :: (\Delta \vdash \mathsf{T}), \\
& & \quad C, F, E, D, Y', E' :: (\Delta, \alpha_e{:}\mathsf{T} \vdash \mathsf{T})\ \Vdash \\
& & V(\overline{\Gamma}, \overline{e_1}, X) \wedge \\
& & V(\overline{\Gamma} \oplus x{:}X, \overline{e_2}, Y) \wedge \\
(1) & & X \doteq (\exists_{\mathsf{T}})\,(\lambda\alpha_e.C) \wedge \\
(2) & & C \doteq F \times E \wedge \\
(3) & & F \doteq D \to \overline{A} \wedge \\
(4) & & D \doteq Y' \times E' \wedge \\
(5) & & Y'[id_{\Delta'}] \doteq Y[id_\Delta] \wedge \\
(6) & & E \doteq \alpha_e \wedge \\
(7) & & E' \doteq \alpha_e
\end{array}
$$

The unification variables X, Y, C, F, and E correspond to the types of the term variables x, y, $clos$, f, and env. Constraint (1) comes from unpacking x, (2) comes from destructuring the variable $clos$ bound by the unpack, (3) comes from applying f, (4) comes the argument to f being a pair, (5) comes from the first pair element being y, and (6) and (7) come from the second pair element being $env\,\mathsf{as}\,\alpha_e$.

Inverting (5) gives us $Y' \doteq Y[id_\Delta]$. We then instantiate D, Y', E, and E' (constraints (4) through (7)) and substitute into (2) and (3), giving us $F \doteq (Y[id_\Delta] \times \alpha_e) \to \overline{A}$ and $C \doteq F \times \alpha_e$. Then instantiating F and C and substituting into (1) gives us that $X \doteq (\exists_\mathsf{T})(\lambda\alpha_e.((Y[id_\Delta] \times \alpha_e) \to \overline{A}) \times \alpha_e)$. Finally we instantiate X.

Putting all of these reductions together, eliminating the temporary bindings, and taking advantage of x not appearing free in $\overline{e_2}$ gives us

$$
\begin{array}{lll}
V(\overline{\Gamma}, \overline{e_1\, e_2}, \overline{A}) & & \\
\quad \twoheadrightarrow^* & \exists Y :: (\Delta \vdash \mathsf{T}) \Vdash & \\
& \quad V(\overline{\Gamma}, \overline{e_1}, & \\
& \qquad (\exists_\mathsf{T})(\lambda\alpha_e.((Y[id_\Delta] \times \alpha_e) \to \overline{A}) \times \alpha_e)) \wedge & \\
& \quad V(\overline{\Gamma}, \overline{e_2}, Y) & \\
\quad = & \exists Y :: (\Delta \vdash \mathsf{T}) \Vdash & \\
& \quad V(\overline{\Gamma}, \overline{e_1}, \overline{Y \to A}) \wedge & \\
& \quad V(\overline{\Gamma}, \overline{e_2}, \overline{Y}) &
\end{array}
$$

By induction we have that $V(\overline{\Gamma}, \overline{e_1}, \overline{Y \to A}) \twoheadrightarrow^* \overline{V(\Gamma, e_1, Y \to A)}$ and $V(\overline{\Gamma}, \overline{e_2}, \overline{Y}) \twoheadrightarrow^* \overline{V(\Gamma, e_2, Y)}$. Then we have

$$
\begin{array}{lll}
V(\overline{\Gamma}, \overline{e_1\, e_2}, \overline{A}) & \twoheadrightarrow^* & \exists Y :: (\Delta \vdash \mathsf{T}) \Vdash \\
& & \quad \overline{V(\Gamma, e_1, Y \to A)} \wedge \\
& & \quad \overline{V(\Gamma, e_2, Y)} \\
& = & \overline{V(\Gamma, e_1\, e_2, A)}
\end{array}
$$

As an aside, to see why the annotation on the use of the variable env was necessary, note that if we did not have that annotation, instead of constraints (6) and (7) we would have the single constraint $E \doteq E'$. We then would have no way to determine that $E \doteq \alpha_e$, and would only be able to get $X \doteq (\exists_\mathsf{T})(\lambda\alpha_e.((Y[id_\Delta] \times E) \to \overline{A}) \times E)$, which is not what we need to match our inductive hypothesis for e_1.

Lambda Now we consider the case for $e = \lambda x{:}B.\, e$. Suppose that the free variables of e are $\{x_1, \ldots, x_n\}$. Then we will have $\Gamma = \Delta; x_1{:}A_1; \ldots x_n{:}A_n; y_1{:}B_1; \ldots y_m{:}B_m$ (where $y_1, \ldots, y_m$ are in the context but not used in e). Because only the free variables are used, we have
$V(\Gamma, \lambda x{:}B.\, e, A) = V(\Gamma', \lambda x{:}B.\, e, A)$ where
$\Gamma' = \Delta; x_1{:}A_1; \ldots x_n{:}A_n$. The translation of the expression is

$$
\begin{array}{l}
\mathsf{pack}_\mathsf{T}[Z, \\
\quad \langle(\hat{\lambda} i{:}I.\, \mathsf{let}\, \langle x, env\rangle = i\, \mathsf{in} \\
\qquad \mathsf{let}\, \langle x_1, \ldots\, x_n\rangle = env\, \mathsf{in} \\
\qquad \overline{e}), \\
\quad \langle x_1, \ldots\, x_n\rangle\rangle] \\
\mathsf{as}\, (\exists_\mathsf{T})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_{\overline{\alpha}}]) \times \alpha_e) \\
(\text{for fresh } I, Y, Z)
\end{array}
$$

After more tedious but uncomplicated computation, we get:

$$
\begin{array}{ll}
V(\overline{\Gamma}, \overline{\lambda x{:}B.\, e}, \overline{A}) = & \\
\quad \exists\, F :: (\Delta, \alpha_e{:}\mathsf{T} \vdash \mathsf{T}), & \\
\qquad B_1, B_2, V_1, \ldots, V_n, Y', & \\
\qquad X, E, X_1, \ldots, X_n :: (\Delta \vdash \mathsf{T}) \Vdash & \\
(1) \quad \overline{A} \doteq (\exists_\mathsf{T})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \wedge & \\
(2) \quad (\exists_\mathsf{T})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \doteq & \\
\quad (\exists_\mathsf{T})\,(\lambda\alpha_e.F) \wedge & \\
(3) \quad F[id_\Delta, (Z/\alpha_e)] \doteq B_1 \times B_2 \wedge & \\
(4) \quad B_2 \doteq (\times_n)\, V_1 \ldots V_n \wedge & \\
(5) \quad V_1 \doteq \overline{A_1} \wedge \cdots \wedge V_n \doteq \overline{A_n} \wedge & \\
(6) \quad B_1 \doteq I \to Y' \wedge & \\
(7) \quad I \doteq X \times E \wedge & \\
(8) \quad E \doteq (\times_n)\, X_1 \ldots X_n \wedge & \\
\quad V(\Delta; x{:}X; x_1{:}X_1; \cdots x_n{:}X_n, \overline{e}, Y') &
\end{array}
$$

With the exception of F, all modal variables that appear in this case are under the context Δ. The unification variables $X, E, X_1, \ldots, X_n$ correspond to the term variables $x, env, x_1, \ldots, x_n$. Constraints (1) and (2) come directly from the outer pack; (3) comes from the term inside of the pack being a pair; (4) comes from the second pair element being an n-tuple, (5) comes from each element of the environment n-tuple being a variable x_i that has type $\overline{A_i}$; (6) comes from the first element of the pair being a lambda with annotation I; (7) and (8) come from deconstructing the argument p. In the recursive invocation of V we elide i and env because they do not occur free in $\overline{e}$.

Performing all of the immediately available instantiations (constraints (4) through (8)) gives us

$$
\begin{array}{ll}
V(\overline{\Gamma}, \overline{\lambda x{:}B.\, e}, \overline{A}) \twoheadrightarrow^* & \\
\quad \exists\, F :: (\Delta, \alpha_e{:}\mathsf{T} \vdash \mathsf{T}), & \\
\qquad Y', X, E, X_1, \ldots, X_n :: (\Delta \vdash \mathsf{T}) \Vdash & \\
(1) \quad \overline{A} \doteq (\exists_\mathsf{T})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \wedge & \\
(2) \quad (\exists_\mathsf{T})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \doteq & \\
\quad (\exists_\mathsf{T})\,(\lambda\alpha_e.F) \wedge & \\
(3) \quad F[id_\Delta, (Z/\alpha_e)] \doteq ((X \times ((\times_n)\, X_1 \ldots X_n)) \to Y') & \\
\qquad\qquad \times ((\times_n)\, \overline{A_1} \ldots \overline{A_n}) \wedge & \\
\quad V(\Delta; x{:}X; x_1{:}X_1; \cdots x_n{:}X_n, \overline{e}, Y') &
\end{array}
$$

Using Decomposition on (2) twice gives us

$$F \doteq ((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e,$$

which we instantiate. The right hand side of this equation contains a free α_e, which will be replaced by Z when F is substituted into constraint (3), yielding the constraint

$$
\begin{array}{l}
((X \times ((\times_n)\, X_1 \ldots X_n)) \to Y') \times ((\times_n)\, \overline{A_1} \ldots \overline{A_n}) \\
\quad \doteq ((\overline{B} \times Z) \to Y) \times Z
\end{array}
$$

Decomposing this repeatedly gives us that $\overline{B} \doteq X$, $Z \doteq (\times_n)\, X_1 \ldots X_n$, $Y \doteq Y'$ and $Z \doteq (\times_n)\, \overline{A_1} \ldots \overline{A_n}$. We can then instantiate X and Y'; instantiating Z and decomposing then gives us $X_i \doteq \overline{A_i}$.

Putting all of these reductions together and eliminating the temporary bindings gives us

$$\begin{aligned} & V(\overline{\Gamma}, \overline{\lambda x{:}B.\, e}, \overline{A}) \twoheadrightarrow^* \\ & \quad \overline{A} \doteq (\exists_{\mathsf{T}})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \wedge \\ & \quad V(\Delta; x{:}\overline{B}; x_1{:}\overline{A_1}; \cdots; x_n{:}\overline{A_n}, \overline{e}, Y) \\ = \quad & \overline{A} \doteq (\exists_{\mathsf{T}})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \wedge \\ & \quad V(\overline{\Gamma' \oplus x{:}B}, \overline{e}, \overline{Y}) \end{aligned}$$

By induction we have that $V(\overline{\Gamma' \oplus x{:}X}, \overline{e}, \overline{Y}) \twoheadrightarrow^* \overline{V(\Gamma' \oplus x{:}X, e, Y)}$. Thus

$$\begin{aligned} & V(\overline{\Gamma}, \overline{\lambda x{:}B.\, e}, \overline{A}) \twoheadrightarrow^* \\ & \quad \overline{A} \doteq (\exists_{\mathsf{T}})(\lambda\alpha_e.((\overline{B} \times \alpha_e) \to Y[id_\Delta]) \times \alpha_e) \wedge \\ & \quad \overline{V(\Gamma' \oplus x{:}X, e, Y)} \\ = \quad & \overline{A \doteq B \to Y[id_\Delta] \wedge V(\Gamma' \oplus x{:}X, e, Y)} \\ = \quad & \overline{V(\Gamma', \lambda x{:}B.\, e, A)} \\ = \quad & \overline{V(\Gamma, \lambda x{:}B.\, e, A)} \end{aligned}$$

The other cases in the proof are straightforward applications of induction. □

Unfortunately, the theorem we proved is only *almost* the theorem we want. In particular, the theorem states that there *exists* an evaluation trace of the algorithm that yields the solution. However, the algorithm is non-deterministic: there are a number of rules that can be applied at every step. It is conceivably possible that even if there existed one execution path that produced a solution, another one might get stuck. We believe that this cannot occur and express this with the following "restricted confluence" conjecture: If $P \twoheadrightarrow^* P_1$, $P \twoheadrightarrow^* P_2$, and $P_1 \twoheadrightarrow^* S$, where S is a solved form, then $P_2 \twoheadrightarrow^* S'$, where S' is a solved form.

Although our testing and this algorithm's long-time use in Twelf leaves us reasonably confident of this conjecture's validity, a proof has stubbornly evaded our attempts so far.

However, since *some* path is guaranteed to exist, even if this conjecture is false, we could still produce a working algorithm by backtracking when a stuck state is reached.

5. Type Forgetting

5.1 Algorithm

As discussed earlier, in order to remove annotations from an early stage intermediate language, we employ a "type forgetting" algorithm. The key idea of the type forgetting algorithm is that in order to determine what annotations we need, we simply run the type reconstruction algorithm and see where it gets stuck. We then iteratively add back in type information until we can solve all of the constraints.

Given some program e, such that $\cdot \vdash e : A$, we construct a detyped expression $e_{detyped}$ as follows

1. Fix some ordering on the type annotations so each annotation has some unique identifier i, $1 \leq i \leq n$, where n is the number of type annotations.
2. Construct $e_{unannotated}$ by replacing each type annotation A_i in e with a fresh modal variable X_i.
3. Compute $P = V(\cdot, e_{unannotated}, A)$ and find irreducible P_0 such that $P \twoheadrightarrow^* P_0$.
4. For each annotation location i we construct a new unification state P_i and a type annotation C_i to use in $e_{detyped}$:
   (a) Find a constraint of the form $X_i \leftarrow B_i$ in P_{i-1}. If no such constraint exists, let B_i be X_i.
   (b) If B_i contains no modal variables, the annotation is reconstructable and so we do not need to include it. Let $C_i = X_i$ (that is, omit the annotation) and $P_i = P_{i-1}$.
   (c) If it does contain modal variables, we can not fully reconstruct the annotation, so we restore it by letting $C_i = A_i$. Then find irreducible P_i such that $P_{i-1} \wedge A_i \doteq B_i \twoheadrightarrow^* P_i$.

5.2 Example

To illustrate the workings of the algorithm, consider the example term

$$\mathsf{let}\, id = \Lambda\alpha{:}\mathsf{T}.\lambda x{:}\alpha.\, x\ \mathsf{in}\ id\,[\mathsf{int}]_{\mathsf{T}}\,5$$

To detype this, we first construct $e_{unannotated}$

$$\mathsf{let}\, id = \Lambda\alpha{:}\mathsf{T}.\lambda x{:}X_1.\, x\ \mathsf{in}\ id\,[X_2]_{\mathsf{T}}\,5$$

We then compute the inference constraints for this term and reduce them until we reach the irreducible equation P_0

$$X_1[X_2/\alpha] \doteq int$$

We now process the first type annotation. There are no constraints of the form $X_i \leftarrow B_i$, so we output α as the annotation to use in place of X_i and find irreducible P_1 such that

$$X_1[X_2/\alpha] \doteq int \wedge \alpha \doteq X_1 \twoheadrightarrow^* P_1$$

We find P_1 as

$$X_1 \leftarrow \alpha \wedge X_2 \leftarrow \mathsf{int}$$

When we go to process the second annotation, we see that the unification state contains $X_2 \leftarrow \mathsf{int}$; since X_2 is fully solved (is instantiated with a type that contains no modal variables), we know we don't need to annotate it, and emit X_2 as the annotation, leaving us with the detyped term $e_{detyped}$

$$\mathsf{let}\, id = \Lambda\alpha{:}\mathsf{T}.\lambda x{:}\alpha.\, x\ \mathsf{in}\ id\,[X_2]_{\mathsf{T}}\,5$$

While the correctness of the algorithm does not depend on the order in which annotations are processed, the number of annotations removed is sensitive to the order. In the above example, if we process the int annotation first, we would add back in that annotation but be left with the irreducible constraint $X_1[\mathsf{int}/\alpha] \doteq \mathsf{int}$ and would need to add back the α annotation as well. In our implementation, the syntax tree is traversed in an order that was found to work well, but nothing particularly clever is done. Investigating heuristics for what order to process annotations is a potential area for future work.

5.3 Correctness

The important correctness property of type forgetting is that the term produced by the algorithm can actually be reconstructed.

Correctness of type forgetting: Given a term $e_{detyped}$ produced by running type forgetting on $\cdot \vdash e : A$, then $V(\cdot, e_{detyped}, A) \twoheadrightarrow^* S$, for some solution S.

Proof. There are two snags that make correctness non-obvious. First, type forgetting adds constraints in piecemeal as it progresses while reconstruction of detyped term adds all of the constraints at the beginning. Second, the constraints added during forgetting do not exactly match how annotations appear during type reconstruction: forgetting adds constraints $(A_i \doteq B_i)$ while reconstruction of the detyped term directly replaces X_i with A_i.

The proof, then, procedes in two stages. First, we show that the problem generated by adding to P a constraint $(X_i \doteq B_i)$ for each annotation we added back can be solved. Second, we show that this implies that $V(\cdot, e_{detyped}, A)$ be solved.

First, in order to talk about the annotations added by the algorithm, we let $Q_i = (A_i \doteq B_i)$, $Q'_i = (X_i \doteq B_i)$ if we added an

annotation for i, and otherwise let $Q_i = \top$, $Q'_i = \top$. Thus, Q_i is the constraint that was added during forgetting and Q'_i represents the type annotation that was added. Let $R = Q'_1 \wedge \cdots \wedge Q'_n$. That is, R represents all of the annotations added to construct $e_{detyped}$.

Recall that the first stage of this proof consists of showing that adding the constraints Q'_i at the beginning of unification is equivalent to adding in the constraints Q_i piecemeal as type forgetting proceeds. Recall that $P = V(\cdot, e_{unannotated}, A)$. We show that $P \wedge R \mapsto\!\!\!\!\twoheadrightarrow^* P_n$. We construct a trace of this that follows the trace from the forgetting process.

First, since $P \mapsto\!\!\!\!\twoheadrightarrow^* P_0$, we have $P \wedge R \mapsto\!\!\!\!\twoheadrightarrow^* P_0 \wedge R_0$, where R_0 is R with the substitutions induced by P_0 applied. Then, for each i (looking only at the case where we have to add an annotation): We have $P_i \wedge R_{i-1}$, where P_i contains $X_i \leftarrow B_i$. This means that R_{i-1} contains $A_i \doteq B_i$. We know that $P_{i-1} \wedge A_i \doteq B_i \mapsto\!\!\!\!\twoheadrightarrow^* P_i$, so then $P_{i-1} \wedge R_{i-1} \mapsto\!\!\!\!\twoheadrightarrow^* P_i \wedge R_i$. Note that for every i, $R_i = Q_{i+1} \wedge \cdots \wedge Q_n$ with the substitutions induced by P_i applied.

Now it remains to be shown that this implies that $V(\cdot, e_{detyped}, A) \mapsto\!\!\!\!\twoheadrightarrow^* S$. The term $e_{detyped}$ is simply $e_{unannotated}$ with certain modal variables X_i replaced with annotations B_i; thus, $V(\cdot, e_{detyped}, A)$ is likewise simply $V(\cdot, e_{unannotated}, A)$ (that is, P) with certain modal variables substituted for. This substitution can be simulated by adding constraints representing the substitutions (that is, R) to P and evaluating. Thus, we have that $P \wedge R \mapsto\!\!\!\!\twoheadrightarrow^* V(\cdot, e_{detyped}, A)$. Since we also have that $P \wedge R \mapsto\!\!\!\!\twoheadrightarrow^* P_n$, if P_n is a solved form, by restricted confluence we have that $V(\cdot, e_{detyped}, A) \mapsto\!\!\!\!\twoheadrightarrow^* S$, for some S.

It remains to show, then, that P_n is a solved form. We show this by observing that if we add in *all* of the annotations ($X_i \doteq B_i$ for each i) to P, this can still reduce to P_n and also (since it is fully annotated) to a solution. Then, by restricted confluence, we know than P_n can reduce to a solution, and because it is irreducible, must be one. □

6. Prototype Implementation

6.1 Implementation

We have tested our technique in a prototype compiler for ML that we are building. In our testbed, we use an ML frontend that elaborates ML to a module calculus based on singleton kinds, and then, through phase splitting [5] and singleton elimination [3], compiles to a fairly conventional intermediate language that we call "IL-Core". IL-Core is a System F_ω variant with all the features needed to compile ML reasonably: existential types, products, sums, recursive types, references, mutually recursive function definitions, exceptions (and the accompanying extensible type they carry), and a collection of base types. To allow more efficient closure conversion and to permit detupling/decurrying optimizations, IL-Core functions are n-ary.

It is at IL-Core that we begin applying the forgetful type-directed methodology. Through the method described in Section 5, we remove much of the type information prior to doing any further translations. The primary compilation stages implemented by the compiler are conversion to A-normal form, closure conversion, and hoisting of closure converted functions. These are all syntax-directed and believed or proved to preserve type reconstructability. We also have some fairly straightforward optimizations: detupling/decurrying of function arguments and a fairly simple optimization pass that eliminates reducible expressions, propagates constants and copies, and eliminates dead code. This optimization pass is run after each of the other passes to clean up after them as well as before type forgetting.

In practice, the performance of the type forgetting algorithm seems quite sensitive to when in the compilation pipeline it is run. Type forgetting is able to remove many more types if our simple optimization pass is run first. One reason for this is that the phase splitting approach to compiling away the ML module system introduces many packs and unpacks of existential types, which inhibit inference; the vast majority of these can be eliminated with simple optimizations. While converting to A-normal form early would enable more optimization opportunities, converting to A-normal form before forgetting yields a large performance penalty: by introducing let bindings for all intermediate values, type inferring code in A-normal form generates many more unification variables and constraints.

While the theory presented above keeps type constructors in canonical form and allows substitutions to appear only on modal variables, the implementation represents types in a calculus of explicit substitutions in which explicit substitutions can be applied to any type constructor [1]. During unification, types are weak head normalized as needed to determine what rules can fire. Variables are represented as De Bruijn indexes, which allows a very compact representation of most substitutions that come up in practice. While notionally we have a number of different intermediate languages, in the implementation, all of the stages we have described following conversion to IL-Core use the same internal representation in order to facilitate reuse of type checking/reconstruction and optimization passes. The type checker/reconstructor, then, is parametric over a set of restrictions on the language.

6.2 Evaluation

To evaluate the effectiveness of this technique in reducing the size of type annotations we measured the size (as measured by the number of kind and type nodes inside of type annotations). We do not present time measurements because we don't have a way to do an apples-to-apples comparison against a similar compiler differing only in how the cost of type information is handled.

We measured the type size of the initial IL-Core program, after running the type forgetting algorithm, after performing all of our compilation stages, and after reconstructing the annotations for the last stage. As tests we used an implementation of infinite dimensional vectors, mutable doubly-linked lists, a collection of monads and assorted utilities, and a miniature version of the ML basis library. The type sizes are presented in Figure 4.

Forgetful type-directed compilation dramatically reduces the size of the types. With all annotations in place, our compilation process increases the amount of type information by nearly a factor of ten. With forgetful type-directed compilation, the sizes of the output types are reduced by over 85%. In fact, they are actually smaller in the translated code than in the original annotated input!

In Figure 5, we show the actual size of the types in memory. Here, we also show the size of the final stage if the type-forgetting algorithm was not used (but forgetful translations were) and of the final stage reconstructed types if all identical types are structurally shared (using a feature of the MLton ML compiler to enforce sharing). While sharing substantially reduces the memory usage of types compared to the fully reconstructed unshared version, it still uses several times more memory than our type-forgotten version.

7. Related Work

A number of techniques have been used to manage type information in typed intermediate languages. Shao et al. "combine hash-consing, memoization, and advanced lambda encoding" in order to preserve physical sharing of type representations in SML/NJ [12]. Their techniques are effective for their uses but require passes to be carefully constructed to take advantage of it. Furthermore, SML/NJ throws away types prior to CPS and closure conversion and so it is unclear whether this technique is sufficient for well performing typed CPS and closure conversion.

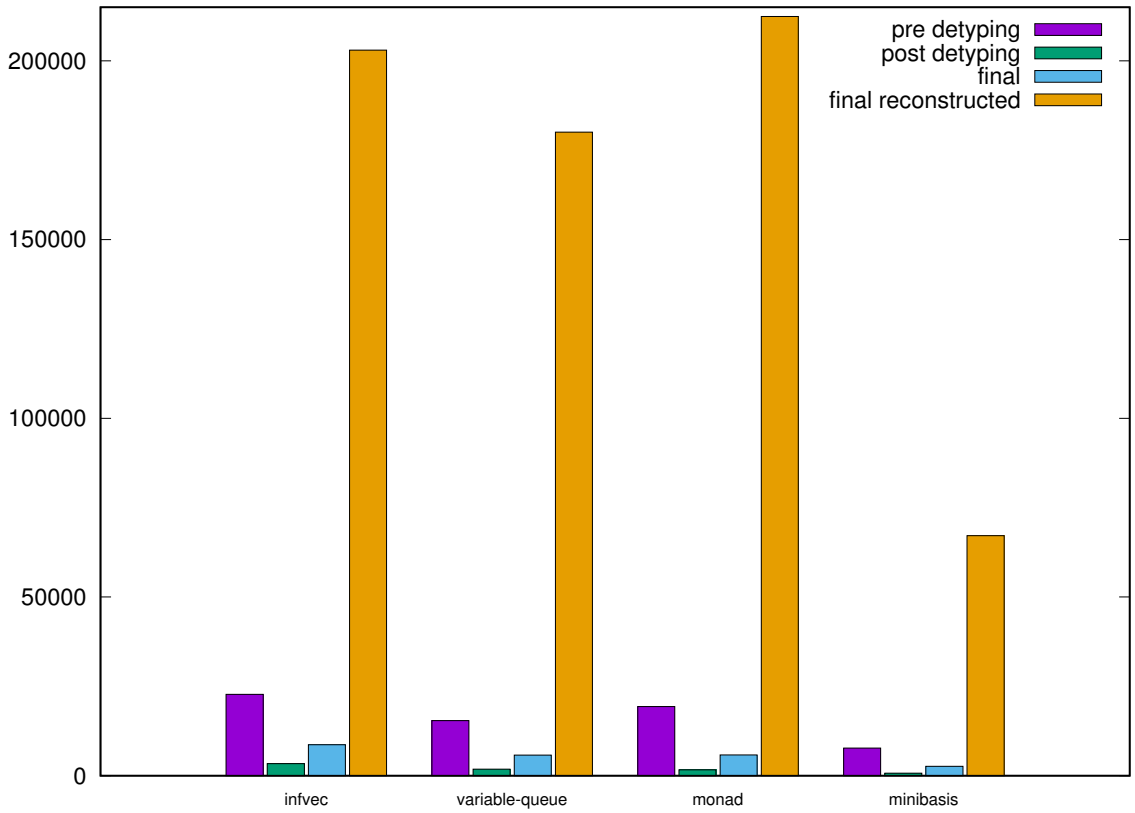

Figure 4. Size of type information in various stages (node counts)

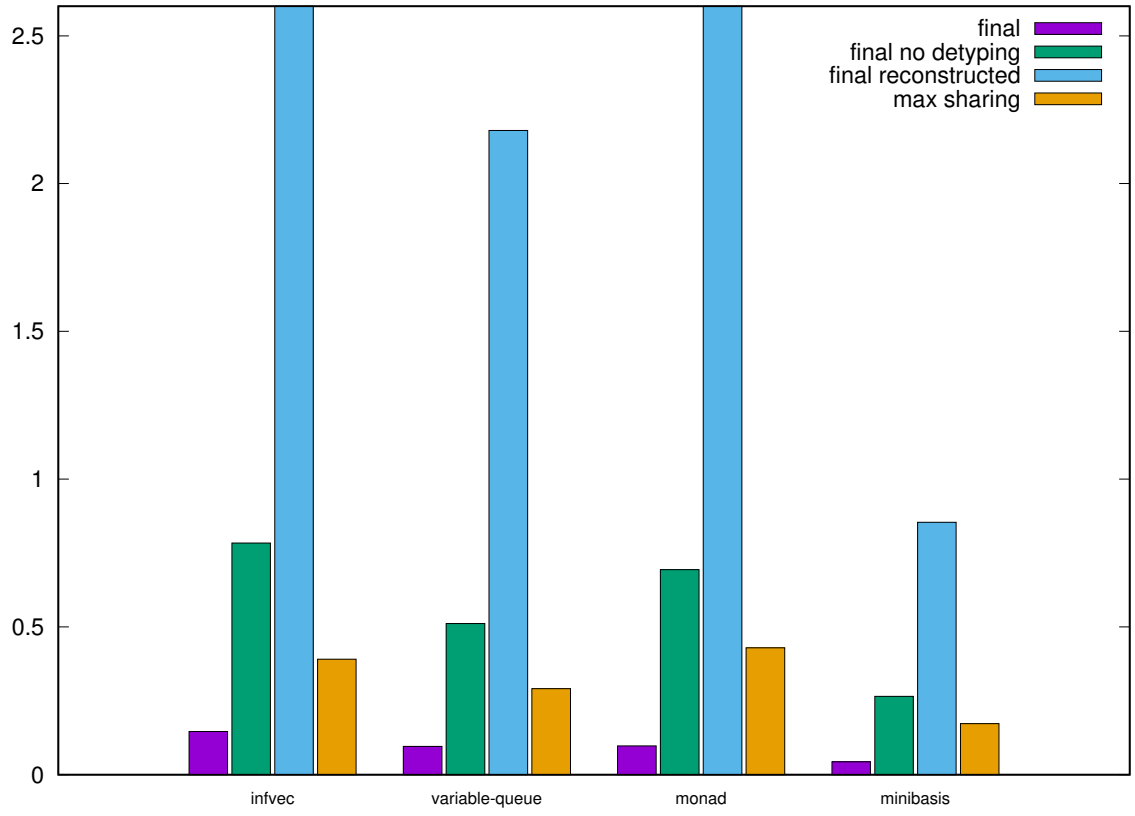

Figure 5. Size of type information in various stages (in megabytes)

The TILT compiler uses sophisticated type theory (such as unlabeled singleton kinds) and a lettype construct to manage its type information [9]. Chlipala et al. proposed Strict Bidirectional Type Checking, which uses strict logic in order to preserve the benefits of bidirectional type checking across sequentialization into A-normal form [2].

Tate et. al present iTalX [13], a system for certifying compilation of object-oriented languages with a similar approach to ours: the target is an inferable typed assembly language, and inference is done after all the compilation stages. Unlike our system, iTalX restricts quantification in order to guarantee that inference is decidable. iTalX avoids type annotations on instructions but needs to annotate function signatures and object layouts; this works well for object oriented imperative languages but is unlikely to scale well to representing functional languages after closure conversion, since all closures would need to be fully annotated.

Our technique, in contrast to all of the above, uses intermediate languages that are fairly standard and unsurprising typed lambda calculi.

While higher-order unification is undecidable, in our setting we are inferring the types in compiler intermediate languages, which should always be well typed. This suggests we could in principle use a semi-decision procedure such as Huet's algorithm [4]; we have not pursued this because we felt that a backtracking algorithm is likely to be too expensive and because it would make catching compiler bugs much more difficult.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, USA, 1990.

[2] A. Chlipala, L. Petersen, and R. Harper. Strict bidirectional type checking. In *2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, Long Beach, California, USA, 2005.

[3] K. Crary. Sound and complete elimination of singleton kinds. In *Third Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 1–25. Springer, Sept. 2000. Extended version published as CMU technical report CMU-CS-00-104.

[4] G. Dowek, T. Hardin, and C. Kirchner. Higher Order Unification via Explicit Substitutions. Rapport de recherche RR-2709, INRIA, 1995.

[5] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, Jan. 1990.

[6] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, 1996.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[8] A. Nanevski, F. Pfenning, and B. Pientka. A contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3), 2008.

[9] L. Petersen, P. Cheng, R. Harper, and C. Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, School of Computer Science, 2000.

[10] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *1998 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988.

[11] J. Reed. Higher-order constraint simplification in dependent type theory. In *Fourth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Montreal, Quebec, Canada, 2009.

[12] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *1998 ACM International Conference on Functional Programming*, pages 313–323, Baltimore, Maryland, Sept. 1998.

[13] R. Tate, J. Chen, and C. Hawblitzel. Inferable object-oriented typed assembly language. In *2010 SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, 2010.

[14] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, School of Computer Science, 2002. Revised May 2003.