Partial Evaluation for Typechecking

DANIEL WINOGRAD-CORT, University of Pennsylvania
HENGCHU ZHANG, University of Pennsylvania
BENJAMIN C. PIERCE, University of Pennsylvania

We study a small functional language in which programs are partially evaluated before typechecking, achieving some of the useful effects of preprocessors, template systems, and macros in a pleasantly straightforward way. We present the system three ways—a declarative formulation reflecting the programmer's view of its behavior, a nondeterministic algorithm uniformly capturing a wide range of possible heuristic choices about how an implementation might interleave partial evaluation and typechecking, and a concrete instance embodying one specific set of heuristics—and show that all three produce the same typings "in the limit." We also show that the system enjoys standard properties including unicity of types, progress, and preservation.

CCS Concepts: •**Theory of computation → Lambda calculus; Type theory;**

ACM Reference format:
Daniel Winograd-Cort, Hengchu Zhang, and Benjamin C. Pierce. 2017. Partial Evaluation for Typechecking. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 19 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Contemporary programming languages often support a bit of program transformation before typechecking. C++ has templates. Scala and Typed Racket have macros. GHC includes Template Haskell. Early implementations of Haskell used a preprocessor to desugar arrow notation before typechecking (Paterson 2001). Even C uses a preprocessor to replace identifiers by numeric constants. These features give programmers the freedom to use idioms that the typechecker would otherwise not be able to understand.

An even simpler way to achieve a similar effect is to replace the program transformation steps found in these systems with a bit of vanilla partial evaluation. Abstractly, we first perform a sequence of reductions (on the whole program, possibly at deeply nested points in its syntax tree), and then invoke a simple typechecker. If $\Gamma \vdash^{\bullet} t : \tau$ represents the underlying typing judgment and $s \Rightarrow^{*} t$ represents a sequence of reductions within s, we can articulate the programmer's declarative view of typing in such a system as $\Gamma \vdash^{\Rightarrow} s : \tau$, meaning there exists t such that $s \Rightarrow^{*} t$ and $\Gamma \vdash^{\bullet} t : \tau$.

Consider this implementation of `sprintf` in a simple functional language with base and function types:

```
let sprintf fmt =
  let rec loop fmt = match fmt with
    | "%d" ++ rest => fun (s : string) (x : int) => loop rest (s ++ int2str x)
    | "%f" ++ rest => fun (s : string) (x : float) => loop rest (s ++ float2str x)
    | "%s" ++ rest => fun (s : string) (x : string) => loop rest (s ++ x)
    | c:rest       => fun (s : string) => loop rest (s ++ [c])
    | ""           => fun (s : string) => s
  in loop fmt ""
```

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

$$e ::= () \mid x \mid e\ e \mid \lambda x{:}\tau.\, e \mid \mu x{:}\tau.\, e$$
$$\tau ::= 1 \mid \tau \to \tau$$

Fig. 1. Syntax

$$\textsc{Val-Unit}\ \frac{}{\mathtt{value}\ ()} \qquad \textsc{Val-Abs}\ \frac{}{\mathtt{value}\ \lambda x{:}\tau.\, e}$$

Fig. 2. Values

$$\textsc{Ev-App1}\ \frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \textsc{Ev-App2}\ \frac{\mathtt{value}\ e_1 \quad e_2 \mapsto e_2'}{e_1 e_2 \mapsto e_1 e_2'}$$
$$\textsc{Ev-Beta}\ \frac{\mathtt{value}\ v}{(\lambda x{:}\tau.\, e)v \mapsto [v/x]e} \qquad \textsc{Ev-Fix}\ \frac{}{\mu x{:}\tau.\, e \mapsto [\mu x{:}\tau.\, e/x]e}$$

Fig. 3. Call-By-Value Reduction

$$\textsc{Pr-Unit}\ \frac{}{() \Rightarrow ()} \qquad \textsc{Pr-Var}\ \frac{}{x \Rightarrow x}$$
$$\textsc{Pr-App}\ \frac{e_1 \Rightarrow t_1 \quad e_2 \Rightarrow t_2}{e_1\ e_2 \Rightarrow t_1\ t_2} \qquad \textsc{Pr-Abs}\ \frac{e \Rightarrow t}{\lambda x{:}\tau.\, e \Rightarrow \lambda x{:}\tau.\, t}$$
$$\textsc{Pr-Beta}\ \frac{e_1 \Rightarrow t_1 \quad \mathtt{value}\ e_2 \quad e_2 \Rightarrow t_2}{(\lambda x{:}\tau.\, e_1)e_2 \Rightarrow [t_2/x]t_1} \qquad \textsc{Pr-Fix1}\ \frac{e \Rightarrow t}{\mu x{:}\tau.\, e \Rightarrow \mu x{:}\tau.\, t}$$
$$\textsc{Pr-Fix2}\ \frac{e \Rightarrow t}{\mu x{:}\tau.\, e \Rightarrow [\mu x{:}\tau.\, t/x]t}$$

Fig. 4. CBV Parallel Reduction

Here, the inner function `loop` takes a format string and returns a function whose arguments match the expected types of values to be formatted. For example, the expression `sprintf "%s=%d"` partially evaluates to `fun (s : string) (x : int) => s ++ "=" ++ int2str x`, which is typeable using just simple types.

We define and study a simple system embodying this idea of partial evaluation for typechecking, which we call PETS: the Partial Evaluation Typing System. The core language is a call-by-value simply typed lambda-calculus extended with recursion (which makes the system much more interesting algorithmically because it means that evaluating terms to normal form before typechecking is not an option). Our main contributions are:

- We propose a straightforward *declarative presentation*—the programmer's view of PETS—in terms of parallel reduction (§2).
- We present a *generalized algorithm* for evaluation and typechecking that uniformly describes a wide range of specific implementation possibilities and prove that it is equivalent to the declarative presentation (§3).
- We establish fundamental *properties* of PETS, including unicity of typing, progress, and preservation (§4).
- We define a simple *deterministic instance* of the generalized algorithmic presentation and show that it is complete "in the limit" as well as sound (§5).

§6 illustrates programming in PETS with some additional examples, and §7 and 8 discuss related and future work. Proofs omitted from this short version can be found in the full version.

2 DECLARATIVE PRESENTATION

The principle of PETS is that a potentially untypeable term s is allowed to parallel reduce for some number of steps. If this reduction yields a term t that is typeable under the standard simply typed lambda calculus (STLC), then the original term is typeable under PETS. We begin with some basic definitions of typing, deterministic call-by-value reduction, and parallel reduction.

The syntax of terms is given in Figure 1. The standard simple typing relation, sans partial evaluation, is written $\Gamma \vdash^{\bullet} e : \tau$. (The definition is elided.) The deterministic step relation (Figures 2 and 3) is the standard small-step, call-by-value one, which we refer to as CBV reduction later. Two important details about CBV reduction are that variables are not values, and it can reduce open terms. For example, the term $(\lambda x{:}1.\, y)\,()$ steps to y.

We conjecture that the exact choice of evaluation strategy does not affect our main results significantly; indeed we have verified that most of the following holds for call-by-name reduction as well (with slightly different proofs). We choose call-by-value due to its ubiquity and straightforward interaction with computational effects.

Figure 4 shows a standard, call-by-value *parallel reduction* relation (Church and Rosser 1936). In addition to the beta and fixpoint reductions of plain CBV reduction, parallel reduction can perform multiple reductions in one pass (as in the Pr-App and Pr-Beta rules) as well as act under binders (as in the Pr-Abs and Pr-Fix1 rules). Essentially, parallel reduction is a technical tool that allows any subset of the current (CBV) redexes in a term to be contracted all at once.

Lemma 2.1. *If $t \mapsto s$, then $t \Rightarrow s$.*

We write $t \Rightarrow^{k} s$ to indicate that t reduces to s in k steps of parallel reduction. (This means the same as "t parallel-reduces to s in at most k steps," since parallel reduction is reflexive.)

Definition 2.2 (PETS). If $e \Rightarrow^{*} e'$ and $\Gamma \vdash^{\bullet} e' : \tau$, then e has the PETS type τ, written $\Gamma \Vdash e : \tau$.

Parallel reduction is confluent: if multiple paths of evaluation can be taken, they can always be brought back together.

Lemma 2.3 (Confluence). *If $e \Rightarrow^{*} t_1$ and $e \Rightarrow^{*} t_2$, then there exists t such that $t_1 \Rightarrow^{*} t$ and $t_2 \Rightarrow^{*} t$.*

Because call-by-value reduction preserves observational equivalence between terms (Crary 2009), and observational equivalence is a congruence, call-by-value parallel reductions naturally inherit the property of preserving observational equivalence as well. That is, if a term s is given a type τ by partially evaluating it to a term t that directly has type τ, we also know that s is "observably of type τ."

Since parallel reduction is nondeterministic, a given term may have many different typing derivations. One might therefore worry that some of these could be inconsistent—i.e., that it might be possible to show both $\Gamma \Vdash e : \tau$ and $\Gamma \Vdash e : \tau'$, where $\tau \neq \tau'$. In §4, we prove this cannot happen because PETS has the Unicity (Theorem 4.3) property.

3 GENERALIZED ALGORITHMIC PRESENTATION

We next present an alternative definition of PETS that replaces parallel-reduction-before-typing with interleaved-typing-and-CBV-steps. This system can be thought of as a unified nondeterministic presentation embodying all possible deterministic strategies for interleaving typechecking with evaluation. We refer to this as the Generalized Algorithmic (GA) system. The main result in this section is that GA-PETS is equivalent to the presentation of PETS based on parallel reduction.

GA is represented by a set of typing rules, shown in Figure 5. In addition to providing a type, the judgments express the *evolution* of one term to another. Thus, a typing judgment $\Gamma \vdash t \rightsquigarrow_k s : \tau$ can be read as, "With the context Γ, the term t evolves to s with type τ and a depth of at most k." The context Γ maps variables to types as usual, and the *depth* is a positive integer that bounds the depth of the derivation tree. (Requiring the depth to be

$$\textsc{Ty-Unit}\ \frac{k > 0}{\Gamma \vdash () \rightsquigarrow_k () : 1} \qquad \textsc{Ty-Var}\ \frac{k > 0}{\Gamma, x : \tau \vdash x \rightsquigarrow_k x : \tau}$$

$$\textsc{Ty-App}\ \frac{\Gamma \vdash t_1 \rightsquigarrow_k s_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 \rightsquigarrow_k s_2 : \tau_2}{\Gamma \vdash t_1\ t_2 \rightsquigarrow_{k+1} s_1\ s_2 : \tau} \qquad \textsc{Ty-Abs}\ \frac{\Gamma, x : \tau_2 \vdash t \rightsquigarrow_k s : \tau}{\Gamma \vdash \lambda x{:}\tau.\, t \rightsquigarrow_{k+1} \lambda x{:}\tau.\, s : \tau_2 \to \tau}$$

$$\textsc{Ty-Fix}\ \frac{\Gamma, x : \tau \vdash t \rightsquigarrow_k s : \tau}{\Gamma \vdash \mu x{:}\tau.\, t \rightsquigarrow_{k+1} \mu x{:}\tau.\, s : \tau} \qquad \textsc{Ty-Eval}\ \frac{e \mapsto t \quad \Gamma \vdash t \rightsquigarrow_k e' : \tau}{\Gamma \vdash e \rightsquigarrow_{k+1} e' : \tau}$$

Fig. 5. Typing Rules

$$\frac{\dfrac{\varnothing, x{:}1 \vdash x \rightsquigarrow_1 x : 1}{\varnothing \vdash \lambda x{:}1.\, x \rightsquigarrow_2 \lambda x{:}1.\, x : 1 \to 1} \quad \varnothing \vdash () \rightsquigarrow_1 () : 1}{\varnothing \vdash (\lambda x{:}1.\, x)\,() \rightsquigarrow_3 (\lambda x{:}1.\, x)\,() : 1} \qquad \frac{(\lambda x{:}1.\, x)\,() \mapsto () \quad \varnothing \vdash () \rightsquigarrow_1 () : 1}{\varnothing \vdash (\lambda x{:}1.\, x)\,() \rightsquigarrow_2 () : 1}$$

Fig. 6. Two different ways to derive a type for $(\lambda x{:}1.\, x)\,()$

positive, as opposed to nonnegative, makes some proofs and definitions simpler below.) When the exact value of the depth bound is irrelevant, we sometimes elide it.

Most of the rules in Figure 5 are unsurprising. For instance, Ty-App says that, if t_1 evolves to s_1 with a function type and t_2 evolves to s_2 with the type of the argument to that function, then $t_1\ t_2$ evolves to $s_1\ s_2$ with the type of the result (and the depth of the derivation tree is one greater). The last rule, Ty-Eval, is the interesting one. It says that, if e evaluates to t and t evolves to e' with type τ, then e also evolves to e' with type τ.

There is a simple weakening lemma for the depth:

Lemma 3.1 (Depth weakening). *If $\Gamma \vdash s \rightsquigarrow_k t : \tau$, then for all $k' > k$, it is also true that $\Gamma \vdash s \rightsquigarrow_{k'} t : \tau$.*

Consider the term $(\lambda x{:}1.\, x)\,()$. With GA, this term has two distinct typing derivations, since both the Ty-App and Ty-Eval rules apply at the top level—one corresponding to reducing first and the other to typechecking without reduction (Figure 6).

Relation to STLC. Before proving that evolutions are equivalent to parallel reduction followed by simple typing, it is worth formally recognizing the relationship between this system and simple typing itself. Specifically, evolutions without the Ty-Eval rule are equivalent to simple typing. Therefore, the proof of a simple type for a term can be used to show its type in PETS. That is, the judgment $\Gamma \vdash^\bullet e : \tau$ implies $\Gamma \vdash e \rightsquigarrow e : \tau$ by using the same derivation tree. The other direction is only true of the evolved term:

Lemma 3.2 (Typing without partial evaluation). *If $\Gamma \vdash t \rightsquigarrow s : \tau$, then $\Gamma \vdash^\bullet s : \tau$.*

To prove that our two definitions are equivalent, we will show that a sequence of parallel reductions can be extracted from an evolution and that parallel reduction can be integrated into an evolution. We begin with extraction:

Lemma 3.3 (Extraction). *If $\Gamma \vdash t \rightsquigarrow_k s : \tau$, then $t \Rightarrow^k s$.*

Using extraction along with Lemma 3.2 yields a sequence of parallel reductions followed by a simple typing, just as required. For the other direction, we must show that parallel reduction can be "embedded" into an evolution. That is, if a term s, in 0 or more parallel steps, becomes t, and $\Gamma \vdash^\bullet t : \tau$, then $\Gamma \vdash s \rightsquigarrow t : \tau$. Proving this statement

requires a few lemmas about parallel reductions as well as its relation to CBV reductions. First, we state two properties about parallel reductions.

Lemma 3.4 (Parallel reduction preserves values). *If* `value` e *and* $e \Rightarrow t$, *then* `value` t.

Lemma 3.5 (Substitution preserves parallel reduction). *If* $e \Rightarrow e'$ *and* $t \Rightarrow t'$, *then* $[t/x]e \Rightarrow [t'/x]e'$.

Next, we state a few properties about the interaction between parallel reductions and CBV reductions. To begin, we show that if a parallel step yields a particular syntactic form, then multiple CBV steps (which we write $\mapsto^*$) will yield that form too. Precisely, for each syntactic form, we show that if s parallel reduces to a term t of that form, then s will eventually CBV reduce to a related term $\hat{t}$ of the same syntactic form, and $\hat{t}$ and t are related by another parallel reduction that doesn't involve a beta reduction at the top level. We use this to additionally show that if s parallel steps to a term t, and t further CBV steps to e, then s will eventually CBV reduce to a term $\hat{t}$ such that $\hat{t}$ parallel steps to e. Lemma 3.7 formalizes this second fact, and it is the critical lemma that helps prove embedding.

Note that in the proofs of Parallel Forms (Lemma 3.6) and Lemma 3.7, only parallel and CBV steps are considered. Since type annotations do not affect how a term steps in PETS, we omit the type annotations in these proofs for cleaner syntax.

Lemma 3.6 (Parallel Forms).

- *If* $s \Rightarrow x$, *then* $s \mapsto^* x$.
- *If* $s \Rightarrow ()$, *then* $s \mapsto^* ()$.
- *If* $s \Rightarrow \lambda x.\, t$, *then there is a term* $\hat{t}$ *such that* $s \mapsto^* \lambda x.\, \hat{t}$ *and* $\hat{t} \Rightarrow t$.
- *If* $s \Rightarrow t_1 t_2$, *then there are terms* $\hat{t}_1$ *and* $\hat{t}_2$ *such that* $s \mapsto^* \hat{t}_1 \hat{t}_2$ *and* $\hat{t}_1 \Rightarrow t_1$ *and* $\hat{t}_2 \Rightarrow t_2$.
- *If* $s \Rightarrow \mu x.\, t$, *then there exists* $\hat{t}$ *such that* $s \mapsto^* \mu x.\, \hat{t}$ *and* $\hat{t} \Rightarrow t$.

The proofs of these statements all follow a similar pattern. We present just the abstraction and application cases.

Statement. *If* $s \Rightarrow \lambda x.\, t$, *then there is a term* $\hat{t}$ *such that* $s \mapsto^* \lambda x.\, \hat{t}$ *and* $\hat{t} \Rightarrow t$.

Proof. By induction.

- Pr-Unit, Pr-Var, Pr-App, Pr-Fix1: These cases are impossible.
- Pr-Beta: In this case, we are given that $s = (\lambda z.\, s_{11}) s_2$ with $s_{11} \Rightarrow \overline{s_{11}}$ and $s_2 \Rightarrow \overline{s_2}$, and we know $[\overline{s_2}/z]\overline{s_{11}} = \lambda x.\, t$. Furthermore, s_2 must be a value and $\overline{s_{11}}$ must be either a variable or another abstraction.
  - If $\overline{s_{11}} = y \neq z$, then $[\overline{s_2}/z]y = y$. But we already know that $[\overline{s_2}/z]\overline{s_{11}} = \lambda x.\, t$, a contradiction.
  - If $\overline{s_{11}} = z$, then $[\overline{s_2}/z]\overline{s_{11}} = \overline{s_2} = \lambda x.\, t$. Since $s_2 \Rightarrow \overline{s_2}$, the induction hypothesis yields reductions $s_2 \mapsto^* \lambda x.\, \hat{t}$ and $\hat{t} \Rightarrow t$. In fact, since s_2 is a value, $s_2 = \lambda x.\, \hat{t}$, so $s = (\lambda z.\, s_{11})(\lambda x.\, \hat{t})$. Also, since $s_{11} \Rightarrow \overline{s_{11}} = z$, the variable case of Parallel Forms (Lemma 3.6) tells us that $s_{11} \mapsto^* z$. Putting all of this together,

$$\begin{aligned} s &= (\lambda z.\, s_{11})(\lambda x.\, \hat{t}) \\ &\mapsto [\lambda x.\, \hat{t}/z]s_{11} \\ &\mapsto^* [\lambda x.\, \hat{t}/z]z \\ &= \lambda x.\, \hat{t} \\ &\Rightarrow \lambda x.\, t. \end{aligned}$$

- If $\overline{s_{11}} = \lambda y.\, \overline{s_{12}}$, then

$$\begin{aligned}[\overline{s_2}/z]\overline{s_{11}} &= [\overline{s_2}/z]\lambda y.\, \overline{s_{12}} \\ &= \lambda y.\, [\overline{s_2}/z]\overline{s_{12}} \\ &= \lambda x.\, t.\end{aligned}$$

  So it must be that $y = x$ and $t = [\overline{s_2}/z]\overline{s_{12}}$. Next, recall that $s_{11} \Rightarrow \overline{s_{11}}$. The induction hypothesis yields $s_{11} \mapsto^* \lambda y.\, \hat{s}_{12}$ and $\hat{s}_{12} \Rightarrow \overline{s_{12}}$. Thus,

$$\begin{aligned}s &= (\lambda z.\, s_{11})s_2 \\ &\mapsto [s_2/z]s_{11} \\ &\mapsto^* [s_2/z]\lambda y.\, \hat{s}_{12} \\ &= \lambda y.\, [s_2/z]\hat{s}_{12} \\ &\Rightarrow \lambda y.\, [\overline{s_2}/z]\overline{s_{12}} \\ &= \lambda x.\, t.\end{aligned}$$

  Letting $\hat{t} = [s_2/z]\hat{s}_{12}$ concludes this case.
- Pr-Fix2: Similar. □

Statement. *If $s \Rightarrow t_1 t_2$, then there are terms $\hat{t}_1$ and $\hat{t}_2$ such that $s \mapsto^* \hat{t}_1\hat{t}_2$ and $\hat{t}_1 \Rightarrow t_1$ and $\hat{t}_2 \Rightarrow t_2$.*

Proof. By induction.

- Pr-Unit, Pr-Var, Pr-Abs, Pr-Fix1: These cases are impossible.
- Pr-App: In this case, we are given that $s = s_1 s_2$, and we know that $s_1 \Rightarrow t_1$ and $s_2 \Rightarrow t_2$. In this case, $\hat{t}_1$ and $\hat{t}_2$ are just t_1 and t_2 themselves.
- Pr-Beta: In this case, we are given that $s = (\lambda x.\, s_{11})s_2$, where s_2 is a value, and we know that $s_{11} \Rightarrow \overline{s_{11}}$ and $s_2 \Rightarrow \overline{s_2}$ and that $[\overline{s_2}/x]\overline{s_{11}} = t_1 t_2$. Furthermore, $\overline{s_{11}}$ must be either a variable or an application.
  - If $\overline{s_{11}} = y \neq x$, then $[\overline{s_2}/x]y = y$. But we already know that $[\overline{s_2}/x]y = t_1 t_2$, a contradiction.
  - If $\overline{s_{11}} = x$, then $[\overline{s_2}/x]\overline{s_{11}} = \overline{s_2} = t_1 t_2$. Furthermore, since s_2 is a value, $\overline{s_2}$ must also be a value and cannot be an application, a contradiction.
  - If $\overline{s_{11}} = s_L s_R$, then

$$\begin{aligned}[\overline{s_2}/x]\overline{s_{11}} &= [\overline{s_2}/x](s_L s_R) \\ &= ([\overline{s_2}/x]s_L)([\overline{s_2}/x]s_R) \\ &= t_1 t_2.\end{aligned}$$

    In turn, this means that $t_1 = [\overline{s_2}/x]s_L$ and $t_2 = [\overline{s_2}/x]s_R$ by matching up the application. Since $s_{11} \Rightarrow \overline{s_{11}} = s_L s_R$, the induction hypothesis yields $\hat{s}_L$ and $\hat{s}_R$ such that $s_{11} \mapsto^* \hat{s}_L\hat{s}_R$ where both $\hat{s}_L \Rightarrow s_L$ and $\hat{s}_R \Rightarrow s_R$. Thus,

$$\begin{aligned}[s_2/x]s_{11} &\mapsto^* [s_2/x]\hat{s}_L\hat{s}_R \\ &= [s_2/x]\hat{s}_L\ [s_2/x]\hat{s}_R \\ &\Rightarrow [\overline{s_2}/x]s_L\ [\overline{s_2}/x]s_R \\ &= t_1 t_2.\end{aligned}$$

    Letting $\hat{t}_1 = [s_2/x]\hat{s}_L$ and $\hat{t}_2 = [s_2/x]\hat{s}_R$ concludes this case.
- Pr-Fix2: Similar. □

We next turn our attention to the second useful lemma that relates parallel reductions to CBV reductions.

Lemma 3.7 (Flipping $\Rightarrow$ and $\mapsto$). *If $s \Rightarrow t \mapsto e$, then $s \mapsto^* \hat{t} \Rightarrow e$ for some $\hat{t}$.*

Proof. By induction on the CBV step relation.

- Ev-App1: The assumptions are $t = t_1 t_2$, with $t_1 \mapsto t_1'$ and $e = t_1' t_2$. By examination of the parallel reduction relation, only the Pr-App, Pr-Beta and Pr-Fix2 cases apply.
  - Pr-App: In this case, the assumptions are $s = s_1 s_2$, and $s_1 \Rightarrow t_1$, and $s_2 \Rightarrow t_2$. Applying the induction hypothesis to $s_1 \Rightarrow t_1 \mapsto t_1'$ yields some $\hat{t}_1$ such that $s_1 \mapsto^* \hat{t}_1 \Rightarrow t_1'$. By congruence of call-by-value reduction, $s = s_1 s_2 \mapsto^* \hat{t}_1 t_2 \Rightarrow t_1' t_2 = e$.
  - Pr-Beta: In this case, the assumptions are $s = (\lambda x.\, s_1) s_2$, and $s \Rightarrow t = t_1 t_2$. Since t is an application, by Parallel Forms (Lemma 3.6) for applications, there are $\hat{t}_1$ and $\hat{t}_2$ such that $s \mapsto^* \hat{t}_1 \hat{t}_2$ and $\hat{t}_1 \Rightarrow t_1$ and $\hat{t}_2 \Rightarrow t_2$. Applying the induction hypothesis to $\hat{t}_1 \Rightarrow t_1 \mapsto t_1'$ yields a $\hat{t}_1'$ such that $\hat{t}_1 \mapsto^* \hat{t}_1' \Rightarrow t_1'$. By congruence of call-by-value reduction again, $s \mapsto^* \hat{t}_1 \hat{t}_2 \mapsto^* \hat{t}_1' \hat{t}_2 \Rightarrow t_1' t_2 = e$.
  - Pr-Fix2: The assumptions are $s = \mu x.\, s_1$, and $s \Rightarrow t = t_1 t_2$. Since t is an application, by Parallel Forms (Lemma 3.6) for applications again, there exist $\hat{t}_1$ and $\hat{t}_2$ such that $s \mapsto^* \hat{t}_1 \hat{t}_2$. The rest is identical to the case above.
- Ev-App2: The assumptions are $t = t_1 t_2$, and $t_2 \mapsto t_2'$ and $e = t_1 t_2'$ and `value` t_1. Again we proceed by case analysis on the parallel reduction.
  - Pr-App: Similar to the Pr-App case under Ev-App1, except the induction hypothesis now applies to the reduction sequence on t_2.
  - Pr-Beta: In this case, the assumptions are $s = (\lambda x.\, s_1) s_2$ and $s \Rightarrow t = t_1 t_2$. Since t is an application, by Parallel Forms (Lemma 3.6) for applications, there are $\hat{t}_1$ and $\hat{t}_2$ such that $s \mapsto^* \hat{t}_1 \hat{t}_2$ and $\hat{t}_1 \Rightarrow t_1$ and $\hat{t}_2 \Rightarrow t_2$. Applying the induction hypothesis to $\hat{t}_2 \Rightarrow t_2 \mapsto t_2'$ yields a $\hat{t}_2'$ such that $\hat{t}_2 \mapsto^* \hat{t}_2' \Rightarrow t_2'$. The rest is identical to the Pr-Beta case under Ev-App1.
  - Pr-Fix2: This case is similar to Pr-Beta.
- Ev-Beta: The assumptions are $t = (\lambda x.\, t_1) t_2$ and `value` t_2, and $e = [t_2/x]t_1$. Since t is an application, by Parallel Forms (Lemma 3.6) for applications, there are $\hat{t}_1$ and $\hat{t}_2$ such that $s \mapsto^* \hat{t}_1 \hat{t}_2$, with $\hat{t}_1 \Rightarrow \lambda x.\, t_1$ and $\hat{t}_2 \Rightarrow t_2$.

  From the reduction $\hat{t}_1 \Rightarrow \lambda x.\, t_1$, by Parallel Forms (Lemma 3.6) for abstractions, there is $\hat{t}_{11}$ such that $\hat{t}_1 \mapsto^* \lambda x.\, \hat{t}_{11}$ and $\hat{t}_{11} \Rightarrow t_1$.

  Since t_2 is a value, it must be either unit or an abstraction. If $t_2 = ()$, then $\hat{t}_2 \Rightarrow ()$, and, by Parallel Forms (Lemma 3.6) for unit values, $\hat{t}_2$ reduces to $()$. On the other hand, if t_2 is an abstraction $\lambda x.\, t_2'$, then by Parallel Forms (Lemma 3.6) for abstractions, there exists $\hat{t}_2'$ such that $t_2 \mapsto^* \lambda x.\, \hat{t}_2'$ and $\hat{t}_2' \Rightarrow t_2'$, which means $t_2 \mapsto^* \lambda x.\, \hat{t}_2' \Rightarrow \lambda x.\, t_2'$. In either case, $\hat{t}_2 \mapsto^* v_2$ where v_2 is a value, and $v_2 \Rightarrow t_2$. By congruence of call-by-value reduction, $s \mapsto^* (\lambda x.\, \hat{t}_{11})\hat{t}_2 \mapsto^* (\lambda x.\, \hat{t}_{11}) v_2 \mapsto [v_2/x]\hat{t}_{11} \Rightarrow [t_2/x]t_1 = e$. Setting $\hat{t} = [v_2/x]\hat{t}_{11}$ concludes this case.
- Ev-Fix: The assumptions are $t = \mu x.\, t_1$ and $e = [\mu x.\, t_1/x]t_1$. By Parallel Forms (Lemma 3.6) for fixpoints, there is some $\hat{t}_1$ such that $s \mapsto^* \mu x.\, \hat{t}_1$ and $\hat{t}_1 \Rightarrow t_1$. The next reduction will be $\mu x.\, \hat{t}_1 \mapsto [\mu x.\, \hat{t}_1/x]\hat{t}_1$, and, since $\hat{t}_1 \Rightarrow t_1$, by Lemma 3.5, $s \mapsto^* \mu x.\, \hat{t}_1 \mapsto [\mu x.\, \hat{t}_1/x]\hat{t}_1 \Rightarrow [\mu x.\, t_1/x]t_1 = e$. Setting $\hat{t} = [\mu x.\, \hat{t}_1/x]\hat{t}_1$ concludes the case. □

With these facts established, we are ready to show how to embed a single parallel step into a GA typing judgment.

Lemma 3.8 (Embedding). *If $s \Rightarrow t$ and $\Gamma \vdash t \rightsquigarrow u : \tau$, then $\Gamma \vdash s \rightsquigarrow u : \tau$.*

Proof. By induction on the typing judgment.

- Ty-Unit and Ty-Var: These two cases are similar and we present the case for Ty-Unit. In this case, $t = u = ()$, and $s \Rightarrow ()$. By Parallel Forms (Lemma 3.6) for unit values, the reduction sequence $s \mapsto^* ()$ holds. Hence, simply apply multiple Ty-Eval rule to reduce s to $()$ and conclude.
- Ty-App: The assumptions are $t = t_1 t_2$ and $u = u_1 u_2$ such that $\Gamma \vdash t_1 \rightsquigarrow u_1 : \tau_2 \to \tau$ and $\Gamma \vdash t_2 \rightsquigarrow u_2 : \tau_2$. Recall that $s \Rightarrow t_1 t_2$, by Parallel Forms (Lemma 3.6) for applications, there exist $\hat{t}_1$ and $\hat{t}_2$ such that $s \mapsto^* \hat{t}_1 \hat{t}_2$ and $\hat{t}_1 \Rightarrow t_1$ and $\hat{t}_2 \Rightarrow t_2$. By the induction hypothesis, the judgments $\Gamma \vdash \hat{t}_1 \rightsquigarrow u_1 : \tau_2 \to \tau$ and $\Gamma \vdash \hat{t}_2 \rightsquigarrow u_2 : \tau_2$ hold.

  By Ty-App, the typing judgment $\Gamma \vdash \hat{t}_1 \hat{t}_2 \rightsquigarrow u_1 u_2 : \tau$ holds. Remember that $s \mapsto^* \hat{t}_1 \hat{t}_2$, hence apply multiple Ty-Eval to conclude.
- Ty-Abs and Ty-Fix: These cases are similar and we present the case for Ty-Abs only.

  The assumptions are $t = \lambda x{:}\tau_2.\, t_1$ and $u = \lambda x{:}\tau_2.\, u_1$ and $\Gamma, x : \tau_2 \vdash t_1 \rightsquigarrow u_1 : \tau_1$ where $\tau = \tau_2 \to \tau_1$.

  Recall that $s \Rightarrow \lambda x{:}\tau_2.\, t_1$, by Parallel Forms (Lemma 3.6) for abstractions, there exists $\hat{t}_1$ such that $s \mapsto^* \lambda x{:}\tau_2.\, \hat{t}_1$ and $\hat{t}_1 \Rightarrow t_1$. So by induction hypothesis, the judgment $\Gamma, x : \tau_2 \vdash \hat{t}_1 \rightsquigarrow u_1 : \tau_1$ holds, which leads to $\Gamma \vdash \lambda x{:}\tau_2.\, \hat{t}_1 \rightsquigarrow \lambda x{:}\tau_2.\, u_1 : \tau_2 \to \tau_1$ by Ty-Abs.

  Remember that $s \mapsto^* \lambda x{:}\tau_2.\, \hat{t}_1$, so applying multiple Ty-Eval rule again concludes this case.
- Ty-Eval: In this case, the assumption says $t \mapsto t'$ and $\Gamma \vdash t' \rightsquigarrow u : \tau$.

  Recall that $s \Rightarrow t \mapsto t'$, by Lemma 3.7, there is $\hat{t}$ such that $s \mapsto^* \hat{t} \Rightarrow t'$. By the induction hypothesis, the judgment $\Gamma \vdash \hat{t} \rightsquigarrow u : \tau$ holds. Since $s \mapsto^* \hat{t}$, applying multiple Ty-Eval rule concludes. □

Embedding in a single parallel step leads to embedding in 0 or more parallel steps.

Lemma 3.9 (General Embedding). *If $s \Rightarrow^k t$ and $\Gamma \vdash t \rightsquigarrow u : \tau$, then $\Gamma \vdash s \rightsquigarrow u : \tau$.*

To embed a sequence of parallel steps followed by simple typechecking into an evolution amounts to proving this lemma.

Lemma 3.10 (Simple Embedding). *If $s \Rightarrow^k t$ and $\Gamma \vdash^\bullet t : \tau$, then $\Gamma \vdash s \rightsquigarrow t : \tau$.*

Proof. General Embedding can absorb arbitrary sequences of parallel steps into a typing judgment of the form in Figure 5. Recall that $\Gamma \vdash^\bullet t : \tau$ implies $\Gamma \vdash t \rightsquigarrow t : \tau$, which, together with General Embedding proves Lemma 3.10 directly. □

Thus, the parallel reduction form of PETS and the evolution form are equivalent.

4 PROPERTIES OF PETS

Having demonstrated that the declarative definition and the GA definition of PETS are equivalent, we investigate the properties of PETS that make it a proper type system in this section.

4.1 Semantic Type

The power of PETS comes from its ability to perform targeted evaluation to transform untypeable terms to typeable ones. If $\Gamma \vdash t \rightsquigarrow s : \tau$, observational equivalence tells us that even if t is not simply typeable, t and s semantically have the same type τ.

The key realization is that "closing a type under evolution-expansion" (which is what PETS essentially does) does not change its meaning: even if t is not typeable with simple types, if it evolves to a term s with type τ, then it is just as good as s behaviorally. In particular, t can safely be substituted for a variable of type τ.

Lemma 4.1 (Substitution). *If $\Gamma, x : \tau_2 \vdash e \rightsquigarrow_{k_1} e' : \tau$ and $\Gamma \vdash t \rightsquigarrow_{k_2} t' : \tau_2$, then $\Gamma \vdash [t/x]e \rightsquigarrow_{k_1+k_2} [t'/x]e' : \tau$.*

Proof. The proof proceeds by induction over the typing judgment on e (generalizing over Γ and the type of the free variable). We examine each case in turn.

- Ty-Unit: We know that $e = e' = ()$, which means substitution has no effect, and we can conclude with weakening.
- Ty-Var: We know that $e = e' = y$ is a variable. If $y \neq x$, then the substitution does nothing. Otherwise, $[t/x]x = t$ and $[t'/x]x = t'$. Because we know $\Gamma \vdash t \rightsquigarrow_{k_2} t' : \tau_2$ from the assumption, then $\Gamma \vdash [t/x]e \rightsquigarrow_{k_2} [t'/x]e' : \tau$, and we can conclude with weakening.
- Ty-App: We know that $e = e_1\ e_2$ and $e' = e_1'\ e_2'$, where $\Gamma, x : \tau_2 \vdash e_1 \rightsquigarrow_{k_1-1} e_1' : \tau_3 \to \tau$ and $\Gamma, x : \tau_2 \vdash e_2 \rightsquigarrow_{k_1-1} e_2' : \tau_3$. By the induction hypothesis, we further have that $\Gamma \vdash [t/x]e_1 \rightsquigarrow_{k_1+k_2-1} e_1' : \tau_3 \to \tau$ and $\Gamma \vdash [t/x]e_2 \rightsquigarrow_{k_1+k_2-1} [t'/x]e_2' : \tau_3$. Using these two results with Ty-App, we reach the correct conclusion.
- Ty-Abs: We know that $e = \lambda y{:}\tau_3.\, e_1$ and $e' = \lambda y{:}\tau_3.\, e_1'$, where $\Gamma, y : \tau_3, x : \tau_2 \vdash e_1 \rightsquigarrow_{k_1-1} e_1' : \tau_4$ and $\tau = \tau_3 \to \tau_4$. We can apply the induction hypothesis here with the context $\Gamma, y : \tau_3$ yielding the result

  $$\Gamma, y : \tau_3 \vdash [t/x]e_1 \rightsquigarrow_{k_1+k_2-1} [t'/x]e_1' : \tau_4.$$

  From this, using the Ty-Abs rule gives us the correct conclusion.
- Ty-Fix: This case is identical to Ty-Abs.
- Ty-Eval: We know that $e \mapsto e_1$, and $\Gamma, x : \tau_2 \vdash e_1 \rightsquigarrow_{k_1-1} e' : \tau$. By the induction hypothesis, we further know that $\Gamma \vdash [t/x]e_1 \rightsquigarrow_{k_1+k_2-1} [t'/x]e' : \tau$. Also, because $e \mapsto e_1$, clearly $[t/x]e \mapsto [t/x]e_1$, so we can use the Ty-Eval rule to get $\Gamma \vdash [t/x]e \rightsquigarrow_{k_1+k_2} [t'/x]e' : \tau$ as required. □

4.2 Unicity

The previous property described what it means for a term to be typeable under PETS; here, we show that PETS is consistent. In other words, we prove that no matter how a term is typed, the type will always be the same, or in other words, that PETS is unitary. For declarative PETS, this means that for a given term e, if e is typeable after a sequence of parallel reductions, then after any sequence of parallel reductions such that the resulting term is typeable, it will have the same type. For GA-PETS, it means that any typing derivation tree will result in the same type. We will prove this using GA-PETS.

Before we present the proof of unicity, we note that parallel reduction preserves simple types, the proof of which can be found in the extended version of this paper.

Lemma 4.2 (PR-Preservation). *If* $\Gamma \vdash^\bullet e : \tau$ *and* $e \Rightarrow e'$, *then* $\Gamma \vdash^\bullet e' : \tau$.

Theorem 4.3 (Unicity). *If* $\Gamma \vdash e \rightsquigarrow_m e_1 : \tau_1$ *and* $\Gamma \vdash e \rightsquigarrow_n e_2 : \tau_2$, *then* $\tau_1 = \tau_2$.

Proof. Using Extraction (Lemma 3.3), parallel reduction sequences can be extracted from the typing judgments as $e \Rightarrow^m e_1$ and $e \Rightarrow^n e_2$. By confluence of parallel reduction (Lemma 2.3), there exists t such that $e_1 \Rightarrow^* t$ and $e_2 \Rightarrow^* t$. Because parallel reduction preserves simple types (Lemma 4.2), and because the judgments imply that $\Gamma \vdash^\bullet e_1 : \tau_1$ and $\Gamma \vdash^\bullet e_2 : \tau_2$, the simple type judgments $\Gamma \vdash^\bullet t : \tau_1$ and $\Gamma \vdash^\bullet t : \tau_2$ hold. Finally, since simple typing obeys unicity, the proof concludes $\tau_1 = \tau_2$. □

4.3 Progress

PETS has the progress property with CBV reductions. Of course, progress makes little sense in the declarative model because parallel reduction can always step, but we can prove a progress property for GA-PETS. It relies on the following simple canonical forms lemma.

Lemma 4.4. *If* $\varnothing \vdash e \rightsquigarrow t : \tau_1 \to \tau_2$ *and* `value` e, *then* $e = \lambda x{:}\tau_1.\, e_1$.

Theorem 4.5 (Progress). *If* $\varnothing \vdash e \rightsquigarrow t : \tau$, *then either* `value` e *or there exists* e' *such that* $e \mapsto e'$.

Proof. Standard. □

4.4 Preservation

Preservation is applicable to both declarative PETS and GA-PETS; we prove it for GA-PETS.

Theorem 4.6 (Preservation). *If $\varnothing \vdash e \rightsquigarrow t : \tau$ and $e \mapsto e'$, then there exists some t' such that $\varnothing \vdash e' \rightsquigarrow t' : \tau$.*

Proof. By induction on the typing judgment.

- Ty-Unit,Ty-Var, Ty-Abs: These are impossible.
- Ty-App: We are given that $e = e_1 e_2$ and $t = t_1 t_2$ along with the typing judgments $\varnothing \vdash e_1 \rightsquigarrow t_1 : \tau_2 \to \tau$ and $\varnothing \vdash e_2 \rightsquigarrow t_2 : \tau_2$. We continue by case analysis on $e \mapsto e'$.
  - Ev-App1: We know that $e_1 \mapsto e_1'$ and $e' = e_1' e_2$. By the induction hypothesis, we have $\varnothing \vdash e_1' \rightsquigarrow t_1' : \tau_2 \to \tau$, which, with Ty-App, lets us construct $\varnothing \vdash e_1' e_2 \rightsquigarrow t_1' t_2 : \tau$ as required.
  - Ev-App2: We know that $e_2 \mapsto e_2'$ and $e' = e_1 e_2'$. By the induction hypothesis, we have $\varnothing \vdash e_2' \rightsquigarrow t_2' : \tau_2$, which, with Ty-App, lets us construct $\varnothing \vdash e_1 e_2' \rightsquigarrow t_1 t_2' : \tau$ as required.
  - Ev-Beta: We know that $e_1 = \lambda x : \tau_2 . e_{11}$ and $e \mapsto [e_2/x]e_{11}$. Furthermore, we know that the typing judgment that states $\varnothing \vdash e_1 \rightsquigarrow t_1 : \tau_2 \to \tau$ must use the Ty-Abs rule, meaning that there is some t_{11} such that $\varnothing, x : \tau_2 \vdash e_{11} \rightsquigarrow t_{11} : \tau$. Using Substitution (Lemma 4.1), we can conclude with $\varnothing \vdash [e_2/x]e_{11} \rightsquigarrow [t_2/x]t_{11} : \tau$.
- Ty-Fix: Similar to the Ev-Beta case of Ty-App.
- Ty-Eval: Trivial. □

5 A DETERMINISTIC PETS ALGORITHM

Both the nondeterminism and undecidability inherent to PETS seem to pose problems toward building a viable implementation of the system. However, GA-PETS, with its reliance only on CBV reductions instead of parallel reductions, can be practically constructed. Furthermore, the nondeterminism and undecidability can both be overcome if we limit how much evaluation is performed and choose a suitable strategy of when to evaluate.

With the guarantee of unicity, any valid typing derivation is as correct as any other; therefore, as long as an implementation can find a typing derivation, it can succeed. This reduces the problem of nondeterminism to searching through a tree that could have multiple correct paths.

Of course, because an evolution may perform an arbitrary number of CBV steps, the tree of derivations to search through is unbounded. This is why we annotated the typing derivations with a tree depth—so that we can write an algorithm such that as long as a derivation exists within a certain depth bound, the algorithm will find it.

This leads us to two simple properties that any implementation of PETS should obey. We assume the implementation is a function named *alg* of two arguments, the depth bound and the term, and it either produces *None*, indicating type failure, or $Some(t, \tau)$, indicating success with the evolved term t and type τ.

Property (Soundness). *If $alg\ k\ e = Some(t, \tau)$, then $\varnothing \vdash e \rightsquigarrow_k t : \tau$.*

Property (Depth-Complete). *If $\varnothing \vdash e \rightsquigarrow_k t : \tau$, then there exists t' such that $alg\ k\ e = Some(t', \tau)$.*

5.1 "Standard Type"-Leaning DFS

We begin by considering a bounded depth-first search through the space of possible derivations with a preference toward the standard typechecking rules (i.e., against partial evaluation). Intuitively, this design comes from the idea that a typical program will likely be "mostly" typeable and that perhaps partial evaluation is only necessary in specific, localized fragments of the code. Used in conjunction with iterative deepening, this approach can be easily expanded to find any derivation tree if it exists. We show an implementation in Figure 7; soundness is obvious by inspection, and depth-completeness is proven below.

Theorem 5.1 (*pets_dfs* is sound). *If $pets\_dfs\ k\ e = Some(t, \tau)$, then $\varnothing \vdash e \rightsquigarrow_k t : \tau$.*

```
let pets_dfs (k : nat) (e : exp) : (exp * typ) option =
  pets_dfs' empty_env k e

let rec pets_dfs' (env : env) (k : nat) (e : exp) : (exp * typ) option =
 do k' <- pred_option k;
    match e with
     | exp_var x  => do T <- lookup env x;
                        return (e, T)
     | exp_num _  => return (e, typ_num)
     | exp_bool _ => return (e, typ_bool)
     | exp_abs Tx body => do (body', Tbody) <- pets_dfs' (Tx::env) k' body;
                            return (exp_abs Tx body', typ_arr Tx Tbody)
     | exp_app e1 e2   => begin match pets_dfs' env k' e1, pets_dfs' env k' e2 with
                           | Some (e1', typ_arr Tx Tr), Some (e2', Tx') =>
                              if Tx == Tx' then return (exp_app e1' e2', Tr)
                                           else do e' <- eval e; pets_dfs' env k' e'
                           | _, _ => do e' <- eval e; pets_dfs' env k' e'
                          end
     | exp_fix Tx body => begin match pets_dfs' (Tx::env) k' body with
                           | Some (body', Tbody) =>
                              if Tx == Tbody then return (exp_fix Tx body', Tx)
                                             else do e' <- eval e; pets_dfs' env k' e'
                           | _ => do e' <- eval e; pets_dfs' env k' e'
                          end
```

Fig. 7. DFS PETS Algorithm favoring standard typing

Theorem 5.2 (*pets_dfs* is depth-complete). *If* $\varnothing \vdash e \rightsquigarrow_k t : \tau$, *then there exists* t' *such that pets_dfs* k e = *Some*(t', τ).

Proof. First, rewrite *pets_dfs*, then generalize over an environment Γ and proceed by induction on k. Let G refer to the typing judgment. The proof follows by case analysis on e.

- If $e = ()$ or $e = x$, then we are done trivially.
- If $e = \lambda x{:}\tau_2.\, e_1$, and $\tau = \tau_2 \to \tau_1$, then G must be an instance of Ty-Abs, and we can use the induction hypothesis on the judgment typing the abstraction's body to conclude that *pets_dfs*′ will succeed.
- If $e = e_1\ e_2$, then G may be an instance of either Ty-App or Ty-Eval. We consider each subcase.
  - Ty-App: By the induction hypothesis, the recursive calls *pets_dfs*′ Γ $(m-1)$ e_1 and *pets_dfs*′ Γ $(m-1)$ e_2 will both be *Some* values with matching types to the typing judgment. A simple analysis of *pets_dfs*′ reveals that, under these conditions, it too will return a *Some* value with the right type.
  - Ty-Eval: In this case, we will consider the possible results to the calls *pets_dfs*′ Γ $(m-1)$ e_1 and *pets_dfs*′ Γ $(m-1)$ e_2. If either of them are *None*, then the returned value will be *pets_dfs*′ Γ $(m-1)$ (*eval* e), and by the induction hypothesis, we are done. If both terms are *Some* values, then, the result will be *Some*(t', τ'), and because *pets_dfs*′ is sound (Theorem 5.1), we know $\Gamma \vdash e \rightsquigarrow_m t' : \tau'$. Finally, by Unicity (Theorem 4.3), we know $\tau = \tau'$ as required.
- If $e = \mu x{:}\tau.\, e_1$, then G may be an instance of either Ty-Fix or Ty-Eval. We consider each subcase.

- Ty-Fix: By the induction hypothesis, the recursive call $pets\_dfs'\ (\Gamma, x : \tau)\ (m-1)\ e_1$ will be a *Some* value with the right type, as required.
- Ty-Eval: Consider the possible results of $pets\_dfs'\ (\Gamma, x : \tau)\ (m-1)\ e_1$. If it is *None*, then the returned value will be $pets\_dfs'\ \Gamma\ (m-1)\ (eval\ e)$, and by the induction hypothesis, we can conclude. If it is $Some(t', \tau')$, then because $pets\_dfs'$ is sound (Theorem 5.1), we know $\Gamma \vdash e \rightsquigarrow_m t' : \tau'$. Finally, by Unicity (Theorem 4.3), we know $\tau = \tau'$, as required. □

Remark. There seems to be a chance for optimization in the definition of $pets\_dfs'$: during the `exp_app` case, if the argument and function types don't match, perhaps the algorithm should just return *None*. However, this would be wrong. The call-by-value evaluation arrow that we use does not check the types before performing beta reduction, which means that it may indeed step even if the function and argument types don't match.

It is possible that a different evaluation arrow could be used that *does* only reduce if the types match. (The proofs of the previous section would likely still be valid up to some slight modifications.) Even so, the implementation of $pets\_dfs'$ would still be correct; when the `exp_app` case falls back to evaluation, the evaluator would simply fail to step, and *None* would be returned anyway.

By this logic, our implementation of *pets_dfs* is actually agnostic on the choice of evaluation arrow. As long as the theory can be proven sound with the chosen evaluation arrow, this implementation will remain sound and depth-complete.

Performance. As mentioned earlier, the *pets_dfs* function is optimized for programs that are either typeable or "mostly" typeable. Indeed, if a program has a typing derivation that does not use partial evaluation, *pets_dfs* will find it immediately, meaning it will perform just as well as type systems that are not enhanced by partial evaluation. Only when regular typing fails will it resort to partial evaluation.

So, what does "mostly" typeable mean? This is not a technical definition so much as a qualitative one, but it means that any necessary evaluation will only need to be performed near the leaves of the derivation tree. In practice, this corresponds to the case where most of the program typechecks normally, but there may be difficult sub-terms that need a little bit of partial evaluation to be typed. For instance, consider a long and complex program in which, at one point deep in its structure, the following term is used:[1]

$$\textbf{if}\ \mathit{true}\ \textbf{then}\ 1\ \textbf{else}\ \mathit{false}$$

Furthermore, assume that the program would be simply typeable if only this term had a numeric type. If *pets_dfs* were used to typecheck this program, it would proceed through most of the derivation without doing any partial evaluation, but when it fails to find a type for this subterm, it will immediately perform one step of evaluation on the **if** term, and then the whole program will be easily typeable.

On the other hand, *pets_dfs* will take a long time trying to find the type of a program that requires partial evaluation near the root of the tree. For example, it may exhaust just about every incorrect derivation tree before finding the correct one in the following example:

$$(\lambda x.())\ \ (\lambda z{:}\mathit{Bool}.\ (\lambda y{:}\mathit{Bool}.\ \textbf{if}\ y\ \textbf{then}\ 1\ \textbf{else}\ 2)\ (\mu x{:}\mathit{Int}.\ x))$$

As *pets_dfs* builds its derivation tree of this term, everything will progress normally at first. Recursive calls will deduce that $(\lambda y{:}\mathit{Bool}.\ \textbf{if}\ y\ \textbf{then}\ 1\ \textbf{else}\ 2)$ is a function from booleans to numbers and that $\mu x{:}\mathit{Int}.\ x$ has a numeric type. However, the application will fail to typecheck, triggering a partial evaluation step, which reduces $\mu x{:}\mathit{Int}.\ x$ before trying to typecheck again. Of course, once again, the application fails to typecheck, and more useless partial evaluation will continue. Only once the entire depth budget is exhausted will the algorithm determine that no amount of partial evaluation can fix the type of $((\lambda y{:}\mathit{Bool}.\ \textbf{if}\ y\ \textbf{then}\ 1\ \textbf{else}\ 2)\ (\mu x{:}\mathit{Int}.\ x))$. Finally, the algorithm

[1] Although our model language does not use booleans, numbers, or **if** statements, they can all be easily derived from the classic lambda calculus terms we did discuss. Therefore, we use them in our examples to be more concise.

will perform a step of evaluation on the whole term, reducing it entirely to (), which it can then typecheck trivially.

One could easily write a search algorithm like *pets_dfs* that instead chooses partial evaluation as its preferred choice for branching, and while it would speed through typechecking examples like the above, it might slow to a crawl when typechecking "mostly" typeable programs. A better solution would likely involve a good heuristic for when and where to try partial evaluation, but this may be hard to build generally as it may depend on what kinds of programs the algorithm can expect. An analytic study of this sort is outside the scope of this work, and we leave it for the future.

5.2 Other Algorithms

Partial Evaluation. Partial evaluation is most commonly used to improve a program's performance by moving computation from runtime to compile time. We can connect this idea of performance-driven partial evaluation to PETS by considering the following simple algorithm:

```
let rec simple_pe (k : nat) (e : exp) : (exp * typ) option =
  match k with
  | O => begin match simple_type_check e with
     | None => None
     | Some t => Some (e, t)
     end
  | S k' => begin match eval e with
     | None => simple_pe O e
     | Some e' => simple_pe k' e
     end
```

The `simple_pe` function performs k reduction steps (or fewer if no more CBV steps can take place) and then proceeds to call `simple_type_check`, a function that finds the simple type of a term and does not use partial evaluation. The term that `simple_pe` returns as a result will necessarily be the partial evaluation of the input term.

Clearly, `simple_pe` is sound—it corresponds to a derivation tree that starts with k instances of the Ty-Eval rule followed by regular typechecking—but just as clearly, it is not depth-complete.

Meta-Programming. Another technique that can be viewed through the PETS lens is meta-programming, which allows one to write programs that construct other programs. Typically, the programmer will designate certain fragments of code as *splices*, which are evaluated at compile time before the typechecker is invoked.[2] After the compiler evaluates these splices, the results are inserted into the code in place of the splice functions, and the program is then typechecked.

In PETS, meta-programming can be mimicked by introducing a new splice operator and creating a typechecker that performs partial evaluation only on spliced terms (and at runtime, the splice operator is replaced with a no-op). Essentially, the algorithm is exactly a standard typechecking algorithm but with the following additional case:

```
let rec meta_pe (E : env) (k : nat) (e : exp) : (exp * typ) option = ...
  | ...
  | exp_splice e =>
```

[2]In some systems, like Template Haskell (Sheard and Jones 2002), the splices are regular, typechecked programs even before they are run at compile time. Specifically, in the case of Template Haskell, a splice must produce a value of one of a handful of special types (e.g., a term, declaration) and be in a special Q monad. Regardless, when meta-programming is used in statically typed systems, the evaluation of the splice must be typeable.

```
      match (eval e >>= \e' -> meta_pe E (k-1) (exp_splice e')) with
      | Some t => return t
      | None => meta_pe E (k-1) e
```

Note that this algorithm is clearly sound but not depth-complete.

Of course, using PETS, one can leverage all the power of meta-programming without needing to manually use a splice operator. That is, the splice operator functions as an annotation telling the typechecker where partial evaluation may be necessary, but with a depth-complete algorithm, the correct path will be found regardless of annotations.

6 EXAMPLES

In this section, we demonstrate some of the capabilities of PETS with a series of further examples.

6.1 Meta-Programming

We showed in the previous section how meta-programming can be viewed as a particular algorithmic instantiation of PETS. Meta-programming creates programs that evaluate to programs of an underlying langauge. Typically, they contain fragments of code that are treated as data and, by design, must not be evaluated. The meta-program manipulates these fragments and splices them together to form the output program. Unfortunately, it is often difficult to write meta-programs as the code manipulation obfuscates the logic itself.

One use of meta-programming would be to write a function of "variable arity", such as one whose first argument indicates how many more arguments there will be. The following variable arity adder (Jones 2016) does just this:

```
p = (\t.(((t(\n.(\a.(\x.(n(\s.\z.a s (x s z)))))))) (\a.a)) zero))
```

Here, we assume a Church encoding, with the usual abbreviations (i.e., zero, one, etc.). For instance, consider the following few reductions:

```
p zero                        => \s.\z.z
(p one) three                 => \s.\z.s (s (s z))
((p two) three) four          => \s.\z.s (s (s (s (s (s (s z))))))
(((p three) three) four) five => \s.\z.s (s (s (s (s (s (s (s (s (s (s z))))))))))
```

The function p is not typeable even with PETS, but when provided with its first argument, PETS will correctly deduce, e.g., that `p two` has the type $nat \to nat \to nat$.

We can look at this same example using a standard meta-programming tool. Consider writing this variable arity adder with Template Haskell (Sheard and Jones 2002):

```
var_adder' :: Int -> Q Exp
var_adder' 0 = [| id |]
var_adder' n = do
  acc <- newName "acc"
  x <- newName "x"
  lamE [varP acc, varP x]
    (appE (var_adder' (n-1)) (appE (appE [|(+)|] (varE acc)) (varE x)))

var_adder :: Int -> Q Exp
var_adder n = appE (var_adder' n) [| 0 |]
```

Essentially, this function describes how to create a variable arity adder for a specific n by using a helper function that utilizes an accumulator. When $n = 0$, the helper function returns the program `id`, which, when applied by the main function, will return the accumulator. Otherwise, it generates names for the accumulator and the

next argument, and constructs a lambda that adds them together and provides them to the recursive call. The main function calls the helper with an initial accumulator of `0`. To use a specific variable adder (say, 4), the programmer would import the module where `var_adder` is defined and write the code `$(var_adder 4)` to splice in the generated function.

The special syntax necessary for quoting and manipulating terms is cumbersome and sometimes obfuscates the algorithm. PETS provides the benefits of meta-programming without special syntax; indeed, the variable arity adder can be written in the most natural way,

```
var_adder' b 0 = b
var_adder' b n = \x -> var_adder' (b+x) (n-1)
var_adder = var_adder' 0
```

and the partial evaluation will automatically assist the typechecker in finding appropriate types:

```
-- one_adder :: Int -> Int
one_adder = var_adder 1
-- four_adder :: Int -> Int -> Int -> Int -> Int
four_adder = var_adder 4
```

6.2 Partial Evaluation

In the previous example, we saw how PETS can be used to elegantly solve problems that would typically require some form of meta-programming or more powerful type system. However, meta-programming is also used for performance reasons: in order to move computation from runtime to compile time. Although not explicitly designed for it, this functionality is well within PETS's capabilities.

Performing computation at compile time to improve runtime performance typically falls under the general name of *partial evaluation*. A classic example is the naive power function:

```
pow e b = case e of
  | 0 -> 1
  | _ -> b * pow (e-1) b
```

The `pow` function takes two arguments, an exponent and a base, and raises the base to the exponent power. To improve performance, we would like to "unroll" the recursion for a specific instance of e to prevent the overhead of successive recursive calls.

Because `pow` has a perfectly appropriate type, a PETS algorithm (such as *pets_dfs* from the previous section), might typecheck it without ever resorting to partial evaluation. However, by being just a little bit clever, we can force evolution to do the partial application work for us. Consider the following untypeable function:

```
forcepeval c r = if c then r else bot
```

This function takes a condition c and a result r; if c is true, then r is returned, but otherwise, the untypeable term `bot` is returned. Because the two branches of the `if` statement don't match, `forcepeval` is never typeable unless partially evaluated to r. Using `forcepeval`, we can rewrite `pow`:

```
pow e b = case e of
  | 0 -> forcepeval (e==0) 1
  | _ -> b * pow (e-1) b
```

All at once, `pow` no longer typechecks. However, let's see what happens when we try to typecheck a particular partial application of `pow`, such as `(pow 2)`. Indeed, there is a typing derivation that succeeds. Specifically, after performing a few steps of partial evaluation interspersed with other typing rules, a PETS algorithm will determine that there is an evolution of `(pow 2)` that typechecks. Specifically, the evolution will be equivalent to $\texttt{pow}_2$ `b = b * b * 1` just as we wanted.

6.3 Piecewise Evaluation

So far, we have demonstrated how to use PETS in a static way, but it can also provide powerful capabilities in a more dynamic way. Imagine an untyped, interpreted language to which we add a new operation, $\checkmark_{\mathrm{TC}}\ e$, which typechecks its argument. If typechecking succeeds, $\checkmark_{\mathrm{TC}}\ e$ returns a *Some* result wrapping the typechecked term, and otherwise, it fails with *None*. Now consider a simple example with user inputs.

```
reversemadlibs = do
  putStrLn "Enter a format string that takes an int and a bool."
  fmt ← getLine
  let s = case ✓TC (sprintf fmt) of
    Some f → f 3 True
    None → "Bad format string"
  putStrLn s
```

In *reversemadlibs*, the user is asked to provide a format string for *sprintf* with certain type requirements. If the language were typechecked statically, the typechecker would have no way of knowing if the format string were appropriate. However, by using $\checkmark_{\mathrm{TC}}$, we can enforce that the use of *sprintf* is valid before running it.

In this simple example, it may be hard to see the difference between this form of deferred typechecking and an altogether untyped language (perhaps with dynamic type assertions). However, the term provided to $\checkmark_{\mathrm{TC}}$ does not need to be limited to something simple. This piecewise static design allows one to mark arbitrary pieces of a program as needing to be typechecked before execution is allowed.

In fact, this is the design basis on which the *Adaptive Fuzz* (Winograd-Cort et al. 2017) programming language is built. Adaptive Fuzz is an extension of Fuzz, a language for differential privacy, which uses a powerful type system to verify that the queries it produces are differentially private. Adaptive Fuzz adds to Fuzz by allowing adaptivity between queries, but because queries may depend on results from previous queries, the Fuzz typechecker cannot be used to prove any differential privacy claims about the whole program. Instead, the typechecker is extended with the power of partial evaluation and the language is extended with a $\checkmark_{\mathrm{TC}}$-like operator. The resulting piecewise structure allows the language to verify every query before it is run while still allowing adaptivity between queries.

7 RELATED WORK

Extending type systems. PETS presents a type system that employs partial evaluation at compile time to extend the set of typeable terms, such that many additional programs that don't go wrong are assigned a meaningful type. This idea of providing a more precise characterization of safe program through a type system has been explored in many other directions. In particular, intersection type systems (Barendregt et al. 1983; Coppo et al. 1981) have been shown to *exactly* characterize the set of terms that have a normal form. Van Bakel (1995) provides a comprehensive survey of different intersection type systems.

Rocca and Venneri (1983) discovered methods of finding principal typing derivations in an intersection type system by using evaluation at "compile time." Principal typing derivations are derivations from which all other typing derivations can be derived. Rocca and Venneri described an algorithm for finding these derivations by $\beta\eta$-reducing the "approximate normal forms" of the original program in the typechecking phase. This is intuitively similar to GA-PETS, where β-reductions are performed during typechecking in an effort to find evolved terms that are more typeable.

The power of intersection types come at the price of some challenging theoretical machinery. PETS provides an alternative approach towards the same goal with a more lightweight mechanism—just closing typing by subject expansion under parallel reduction.[3]

Partial evaluation. Veldhuizen (2000) described a unique approach to compiling C++ programs. First, C++ source code is first translated into an untyped intermediate language in which C++ types are reified as values. Next, a partial evaluator specializes the intermediate language, where the specialization step provides typechecking, template function instantiation, and optimization. By varying the degrees of specialization and template instantiation, the binary code produced can be much smaller than that produced using a traditional compilation approach.

Interleaving typing and evaluation has also been explored in research on staged computation. Shields et al. (1998) discussed a staged language that achieves type safety by delaying the typechecking of spliced terms until runtime. PETS, although not explicitly staged, also employs this interleaving of typing and evaluation in its GA presentation. Indeed, PETS is allowed to interleave evaluation with typechecking in any order, allowing it to capture specifics of many possible concrete (heuristic) implementations. In particular, the algorithm presented in §5 chooses to interleave evaluation with typechecking only when all other typing rules fail to apply.

Model-checking algorithms have also been shown to benefit from partial evaluation. Kobayashi (2009) presented efficient algorithms for model checking higher order functions. The traditional approach first converts programs into recursion schemes and then model checks those schemes, but when done naively, this process explodes in complexity and becomes impractical. Kobayashi suggests reducing the recursion scheme as the model checking process proceeds, which is similar to GA-PETS.

Similar Tools. There are a number of tools that, although technically different from PETS, can be used to solve similar problems. For instance, meta-programming—as in Template Haskell (Sheard and Jones 2002), MetaML (Taha and Sheard 2000), Scheme's macros (Adams et al. 1998), and others—allow a programmer to write code that, itself, generates the final program. This can sometimes be simulated with evolution in PETS, but the programmer is given more precise control in the meta-programming systems. However, in exchange for the extra freedom, the design of meta-programs is complicated by an extra layer of syntax as well as the semantics of the meta-language evaluator.

Pre-processors like that of C are often used to encode programming idioms that fall outside what the typechecker can understand. They tend to be less context sensitive than languages' built-in meta-programming tools, which can be good or bad depending on the task at hand, and they are widely used (Ernst et al. 2002).

Our example of piecewise evaluation in §6.3 has a similar feel to multi-stage programming (Taha 2004), a variety of meta-programming in which multiple phases of type-safe evaluation take place. Like other meta-programming styles, multi-stage programming requires annotating at which stage of evaluation each term should be evaluated. In our example, this is accomplished simply with the type-check operator $\checkmark_{\mathrm{TC}}$.

In many cases where PETS can find a type for a term, a dependent type system might be able to as well. In dependently typed languages (such as Cayenne (Augustsson 1999), Agda (Norell 2009), Epigram (Altenkirch et al. 2005), Idris (Brady 2013), etc.), runtime values can be lifted to the type system to provide extra static information. This information can be used to inform the type system how a term will evaluate, thus allowing it to deduce a much more precise type. One motivation for designing PETS was to achieve some of the precision of dependent type systems with a much lighter-weight mechanism. While dependent types can provide more static information to the programmer than PETS, PETS is able to accept piecewise evaluated programs, like in §6.3, while dependent type systems cannot.

[3]We are grateful to Jakob Rehof for this observation.

8 LIMITATIONS AND FUTURE WORK

8.1 Theory

Evaluation Schemes. In this work, we showed how one can use PETS in a language that uses call-by-value semantics. We conjecture that the work can be adapted to call-by-name semantics as well (although we have not proved this), but we have yet to explore other evaluation strategies or find a general form. Ideally, we would like to be able to list a set of laws that an evaluation strategy must follow, define a method to derive a parallel reduction scheme from such a strategy, and then have our proofs be applicable to any such strategy. This will be a topic of our attention going forward.

Effects. We defined PETS over a simple lambda calculus extended with fixed points, but can PETS work on a language with other effects? One possible solution would be to define a subset of the evaluation semantics in which no effects can be performed and use this subset as the PETS evaluation strategy. In other words, the partial evaluation would be able to perform beta reductions as it does in this work, but it would not be able to perform any effects. That said, it may be possible for certain effects to be allowable during typechecking, and we would like to explore exactly which ones and how they may work.

Type Inference. It is not clear what the interaction between PETS and type inference is. In certain situations, the type of an object must be known in order for evaluation to proceed (such as with Haskell's type classes and the implicit libraries they pass around), which means that partial evaluation may not be possible when a type cannot be found. It may be that this presents an obstacle for PETS, limiting its effectiveness in complex systems. We would like to explore this more fully.

8.2 Pragmatics

We have focused on the theoretical aspects of PETS in this paper, but there are many directions to explore in the future that would make PETS a better practical programming environment.

Search Depth. PETS algorithms are defined as always taking a depth parameter k, but what values of k should be used in practice? As k is a measure of the derivation tree depth, an easy heuristic might be to set k proportional to the size of the term being typechecked. Beyond this, a practical approach would be to iteratively increase k until either a type is found or a time limit is reached. Experimentation is likely required.

Search Heuristic. As described in §5.1, a good PETS algorithm should use a heuristic to decide whether to try partial evaluation or simple typechecking. One idea would be to start by performing simple typechecking, and when this fails, determine why. If there is an unbound variable, perform evaluation steps at its definition site until it is substituted with a value; if a function and its argument don't have the right type, try to evaluate the argument to a value and perform the beta reduction anyway. Only by trying to create good heuristics and then trying them on real examples will we be able to determine what is effective.

Fragility. One problem with PETS is that it can typecheck terms that are *fragile*. A fragile term is one that typechecks, but small changes in the code, even non-local to the term, can cause typechecking to fail. For instance, consider the following two definitions:

```
let b = true
let f x = if b then 1 else (if x then 1 else false)
```

Although `f` would not typecheck in a typical system, PETS will evaluate the outer **if** statement to 1 and declare that `f` is a constant function that returns a number. However, if we change `b` to false, then `f` is no longer typeable, even under PETS. Thus, we call `f` *fragile*. It is not clear that allowing fragile terms is a good thing: although it may make programming easier in the short term, it could lead to hard to manage code in the long term.

Error Messages. PETS introduces a new model for typechecking, and this new model necessitates a new form of interaction with the user. What should an error message for PETS look like when it cannot find a derivation? Is it useful to see what sort of partial evaluation the PETS algorithm performed when trying to typecheck the term? We do not yet have answers to these questions.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-1065060 and CNS-1513694, and a grant from the Sloan Foundation.

REFERENCES

N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. 1998. Revised^5 Report on the Algorithmic Language Scheme. *SIGPLAN Not.* 33, 9 (Sept. 1998), 26–76. DOI: http://dx.doi.org/10.1145/290229.290234

Thorsten Altenkirch, Conor McBride, and James McKinna. 2005. Why dependent types matter. *Manuscript, available online* (2005), 235.

Lennart Augustsson. 1999. *Cayenne — A Language with Dependent Types.* Springer Berlin Heidelberg, Berlin, Heidelberg, 240–267. DOI: http://dx.doi.org/10.1007/10704973_6

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *The Journal of Symbolic Logic* 48, 4 (1983), 931–940. http://www.jstor.org/stable/2273659

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (001 009 2013), 552–593. DOI: http://dx.doi.org/10.1017/S095679681300018X

Alonzo Church and J. B. Rosser. 1936. Some Properties of Conversion. *Trans. Amer. Math. Soc.* 39, 3 (1936), 472–482. http://www.jstor.org/stable/1989762

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional characters of solvable terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58.

Karl Crary. 2009. A simple proof of call-by-value standardization. *Computer Science Department* (2009), 474.

Michael D Ernst, Greg J Badros, and David Notkin. 2002. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering* 28, 12 (2002), 1146.

Neil Jones. 2016. Private communication. (2016).

Naoki Kobayashi. 2009. Model-checking Higher-order Functions. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '09)*. ACM, New York, NY, USA, 25–36. DOI: http://dx.doi.org/10.1145/1599410.1599415

Ulf Norell. 2009. *Dependently Typed Programming in Agda.* Springer Berlin Heidelberg, Berlin, Heidelberg, 230–266. DOI: http://dx.doi.org/10.1007/978-3-642-04652-0_5

Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 229–240. DOI: http://dx.doi.org/10.1145/507635.507664

S.Ronchi Della Rocca and B. Venneri. 1983. Principal type schemes for an extended type theory. *Theoretical Computer Science* 28, 1-2 (1983), 151–169. DOI: http://dx.doi.org/10.1016/0304-3975(83)90069-5

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. DOI: http://dx.doi.org/10.1145/581690.581691

Mark Shields, Tim Sheard, and Simon Peyton Jones. 1998. Dynamic Typing As Staged Type Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 289–302. DOI: http://dx.doi.org/10.1145/268946.268970

Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 30–50. DOI: http://dx.doi.org/10.1007/978-3-540-25935-0_3

Walid Taha and Tim Sheard. 2000. MetaML and Multi-stage Programming with Explicit Annotations. *Theor. Comput. Sci.* 248, 1-2 (Oct. 2000), 211–242. DOI: http://dx.doi.org/10.1016/S0304-3975(00)00053-0

Steffen Van Bakel. 1995. Intersection type assignment systems. *Theoretical Computer Science* 151, 2 (1995), 385–435.

Todd L Veldhuizen. 2000. Five compilation models for C++ templates. In *First Workshop on C++ Template Programming*. Citeseer.

Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. Submitted 2017. A Framework for Adaptive Differential Privacy. (Submitted 2017). Submitted manuscript.