Dynamic Typing and Subtype Inference

Alexander Aiken Manuel Fähndrich

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776
{aiken,manuel}@cs.berkeley.edu

Abstract

Dynamic typing is a program analysis targeted at removing runtime tagging and untagging operations from programs written in dynamically typed languages. This paper compares dynamic typing with a subtyping system based on set constraints. The purpose is both to make precise the relationship between two superficially unrelated type systems and to illustrate how the advantages of dynamic typing and subtype inference can be combined. The central result is a theorem showing that a typing discipline at least as powerful as dynamic typing can be expressed using set constraints.

1 Introduction

This paper presents a study of Henglein's *dynamic typing* discipline [Hen92a, Hen92b]. Dynamic typing extends conventional static types with a single new type `Dynamic`. Special functions called *coercions* inject values into and project values from type `Dynamic`. Currently, the main application of dynamic typing is the optimization of programs written in dynamically typed languages (such as Lisp and Scheme) by removing runtime tests of type tags where they are provably unnecessary (so-called *soft typing* [CF91, WC94]). A remarkable, and to our knowledge unique, aspect of dynamic typing is that it not only permits the removal of dynamic type tag tests, but also allows the elimination of type tagging operations themselves.

The purpose and results of our study are two-fold. First, while dynamic typing is a very interesting system, it cannot remove as many type checks as other recently proposed algorithms based on *inclusion subtyping* [AWL94, WC94]. By inclusion subtyping, we mean systems where type t_1 is a subtype of t_2 if t_2 includes every value of t_1; we will refer to this simply as subtyping.

As noted above, dynamic typing has the singular ability to remove type tagging operations as well as dynamic type checks. Thus, the power of dynamic typing is incomparable to the subtyping approaches. One of our goals is to investigate whether the strengths of dynamic typing can be combined with the strengths of subtyping. Our results are positive: We present a generalization of dynamic typing that incorporates an expressive subtyping discipline. Type inference for the system has time complexity $\mathcal{O}(n^3)$ and appears amenable to a practical implementation.

Our second interest is with dynamic typing itself, irrespective of any applications. Many contemporary program analysis algorithms are based on constraint resolution, including the algorithms for dynamic typing. In constraint-based analysis, constraints are generated from the program text and solving the constraints yields the analysis of the program. It is our thesis that many constraint-based analyses can be expressed using a particular constraint theory known as *set constraints*. Set constraints are a simple, general, and well-studied theory that is powerful enough to express many program analyses [HJ90, AW92, Hei92, Aik94].

In testing our thesis on a variety of program analyses, it became apparent that dynamic typing is in some ways fundamentally different from other examples of constraint theories used in program analysis. The main technical challenge, and our central result, is establishing that set constraints can encode dynamic typing. This characterization facilitates direct comparison of dynamic typing with other constraint-based analyses. However, the set constraint formulation does not naturally suggest the very efficient resolution algorithms known for dynamic typing [Hen92b]; in this respect, dynamic typing appears to stand apart.

The rest of this section presents an overview of the paper. Some basic definitions are needed. Following [Hen92a], our results are presented using a small, paradigmatic language called *dynamically typed lambda calculus*. The expressions of the language are:

$$e ::= x \mid \lambda x.e \mid e\ e' \mid \texttt{if}\ e\ e'\ e'' \mid \texttt{true} \mid \texttt{false} \mid C\ e$$

The dynamically typed lambda calculus is a call-by-value language with two important features. First, a term $C\ e$ is a *coercion* C applied to the value of e. Intuitively, coercions model the runtime type checks implicit in dynamically typed programs. Formally, coercions are primitive functions that perform tagging and untagging operations. The semantic domain D contains four distinct kinds of elements: tagged functions, untagged functions, tagged booleans, and untagged booleans:

$$D = ((D \rightarrow D) + \mathit{Bool}) \times (\mathit{notag} + \mathit{tag})$$

For example, the coercion FUNC! tags its (function) argument as a function; FUNC! has signature $(D \rightarrow D) \times \mathit{notag} \rightarrow (D \rightarrow D) \times \mathit{tag}$. The coercion FUNC? checks that its argument is a function and returns the untagged function value or an exception; it has signature $((D \rightarrow D) + \mathit{Bool}) \times \mathit{tag} \rightarrow (D \rightarrow D) \times \mathit{notag}$. Thus, FUNC? (FUNC! $\lambda x.x) = \langle \lambda x.x, \mathit{notag} \rangle$, but the expression FUNC? (BOOL! `true`) is an exception. Similarly, BOOL! tags its (boolean) argument as a boolean and BOOL? performs a check-and-untag operation. The second important aspect of the language is that the semantic domain contains both functions and booleans. The pure lambda calculus would be uninteresting for dynamic typing because no type checking is required—no runtime errors can arise without a data type distinct from functions. The results we present are easily extended to a language with arbitrary data types.

original term	$(\lambda x.x)\ (\lambda y.y)$
canonical completion	(FUNC? (FUNC! $\lambda x.x$)) (FUNC! $\lambda y.y$)
minimal d.t. completion	$(\lambda x.x)$ (FUNC! $\lambda y.y$)
original term	(`if` `true` ($\lambda x.$`true`) `false`) `false`
canonical completion	(FUNC? `if` (BOOL? (BOOL! `true`))(FUNC! $\lambda x.$(BOOL! `true`)) (BOOL! `false`)) (BOOL! `false`)
minimal d.t. completion	(FUNC? `if` `true` (FUNC! $\lambda x.$(BOOL! `true`)) (BOOL! `false`)) (BOOL! `false`)
minimal s.c. completion	(FUNC? `if` `true` (FUNC! $\lambda x.$(BOOL! `true`)) (BOOL! `false`)) `false`

Figure 1: Example completions of dynamically typed lambda terms.

The use of a *notag* label to denote untagged values in the domain is non-standard, but no additional runtime overhead is implied; *notag* values would not carry any decoration at runtime. Rather, this representation of the domain is chosen to make clear the correspondence with a type system presented in Section 3.[1]

Let *erase*(e) be e with all coercions deleted. We say e is a *completion* of e' if *erase*$(e) = e'$. Implementations of dynamically typed languages complete user programs by inserting tagging operations where values are created and inserting type checking operations where values are used. Thus, the semantics of a dynamically typed lambda term can be defined to be the meaning of the completion that performs all possible type operations.

Definition 1.1 Let $e =$ *erase*(e). The *canonical* completion of e is defined by the following table. Each subexpression of e matching an entry on the left is modified according to the corresponding entry on the right:

Before	*After*
x	x
$\lambda x.e$	FUNC! $\lambda x.e$
$e\ e'$	(FUNC? e) e'
`if` $e\ e'\ e''$	`if` (BOOL? e) $e'\ e''$
`true`	BOOL! `true`
`false`	BOOL! `false`

Let e be a term with no coercions. A completion e' of e is *correct* if it is semantically equivalent to the canonical completion of e. We are free to choose among correct completions, though completions with fewer coercions are preferred for efficiency reasons. Thus, the goal of dynamic typing is to compute a correct completion with as few coercions as possible.

Dynamic typing, as formulated in [Hen92a], has computable *minimal* completions. A completion e' of e is minimal if every derivable completion of e includes all the coercions of e'. Two examples are given in Figure 1. The first example shows two completions of the term $(\lambda x.x)(\lambda y.y)$. Note that even in the minimal completion the value $\lambda y.y$ is tagged; this is necessary because $\lambda y.y$ is the result of evaluation, which is a tagged value in the canonical completion.

The second example is contrived to illustrate several points about the dynamic typing discipline. Consider the minimal completion under dynamic typing (labelled d.t.). Note that the boolean in the predicate position of the conditional is untagged. Dynamic typing infers that a boolean is used in a position where a boolean is expected, so no check is required to ensure the value is a boolean and, in fact, the value need not be tagged as a boolean at all. However, both branches of the conditional are tagged and a FUNC? test is applied to the result of the conditional. Dynamic typing cannot infer what type results from the conditional, so all values that can be produced have identifying tags to enable types to be determined at runtime by FUNC?. The value `true` returned by the constant function $\lambda x.$`true` on the true branch must be tagged because it is the result of the expression.

Finally, the argument `false` to the function result of the conditional is also tagged. This is peculiar, because the value is not even used by the constant function $\lambda x.$`true`. In fact, this example illustrates a weakness of dynamic typing. The completion arises because dynamic typing assigns a single type `Dynamic` to all tagged values. That is, the type of the conditional is just `Dynamic`—no structural information about what values can result from the conditional is expressed. When FUNC? is applied, nothing is known about the type of the function that results, so it must have type FUNC? : `Dynamic` $\rightsquigarrow$ (`Dynamic` $\rightarrow$ `Dynamic`), which forces the components of the function type to also be tagged and tested at runtime. (The use of $\rightsquigarrow$ instead of $\rightarrow$ in the type is for consistency with notation in [Hen92b, Hen92a] and emphasizes the special role of coercions.) In dynamic typing, if a value has type `Dynamic`, then all of its components must have type `Dynamic`.

The system we present, based on set constraints, allows components of a type to be untagged even if the type itself represents a tagged value. Figure 1 shows the minimal set constraint completion (labelled s.c.) for the second example. Note that the function argument is untagged. The example is admittedly contrived; it is difficult to construct realistic examples in the dynamically typed lambda calculus! However, the practical effect is easy to understand. In dynamic typing, if any component of a data structure is tagged (has type `Dynamic`), then all subcomponents must be tagged (have type `Dynamic`), and all associated type checking operations must be performed. Thus, the need to introduce type operations on a single component of a large data type may result in the introduction of type operations on many other components.

It is not obvious how to generalize dynamic typing to avoid this phenomenon, but it can be done. Set constraints provide one natural solution. Dynamic typing also can be modified directly to avoid the extra tagging; the resulting system is no longer dynamic typing and is closer than dynamic typing to the system we present.[2] Known results on set constraints also admit immediate generalizations in other, orthogonal ways, including adding polymorphic types [AW93] and analysis of conditional branches [AWL94].

The formal development proceeds as follows. Section 2 presents a type inference system for dynamic typing. This system proves facts of the form

$$A \vdash_D e : \tau$$

Section 3 presents an alternative formulation of dynamic typing using set constraints. It turns out that the "obvious" encoding of

[1]Furthermore, a rigorous presentation must include a *wrong* value in the domain to denote exceptions. We gloss over this well-known construction to save notation in later definitions.

[2]Fritz Henglein, private communication, January 1995

dynamic typing fails in a inclusion subtyping system; the explanation why highlights some interesting technical aspects of dynamic typing. We also state a soundness theorem for our system. The set constraint system proves facts of the form

$$A, S \vdash_S e : \sigma$$

where S is a system of set constraints. The meaning of the derivation is that under assumptions A, expression e has type $s(\sigma)$ for every substitution s that is a solution of the constraints S.

Section 4 is the heart of the paper. We prove a theorem showing that the set constraint system is at least as powerful as dynamic typing. More formally, we first define a mapping T from types τ to types σ. We then prove

$$A \vdash_D e : \tau \;\Rightarrow\; T(A), S \vdash_S e : \sigma$$

where $\sigma \subseteq T(\tau)$ and S is a consistent system of constraints. Because of the nature of the mapping T, a corollary of this theorem is that every completion that is $\vdash_D$ derivable is also $\vdash_S$ derivable. The example in Figure 1 shows that some completions are $\vdash_S$ derivable but not $\vdash_D$ derivable.

Section 5 presents an algorithm for computing completions in the set constraint system. Analysis of the algorithm shows that the set constraint system has unique minimal completions and that the completions can be computed in $\mathcal{O}(n^3)$ time in the size of the original expression.

Section 6 briefly outlines extensions and restrictions of the main result. We show that the set constraint system can be restricted to have exactly the same power as dynamic typing, thereby precisely characterizing its power with respect to other analyses based on set constraints. We also consider a variation of dynamic typing where coercions may appear at points other than value creations and uses. (We do not consider *induced coercions*, another variation on dynamic typing in Henglein's original work [Hen92a].) Finally, we report that the set constraint system can be incorporated into the most expressive system known for removing type tags, although in this case there are no longer minimal completions and constraint resolution becomes inherently exponential.

Section 7 presents discussion of related work and a few concluding remarks.

2 Dynamic Typing

The types of dynamic typing are generated by the following grammar:

$$\tau ::= \alpha \mid \texttt{Bool} \mid \texttt{Dynamic} \mid \tau \to \tau' \mid \mathrm{fix}\,\alpha.\tau$$

In this grammar, α is a type variable and $\mathrm{fix}\,\alpha.\tau$ denotes a regular recursive type that is the solution of the equation $\alpha = \tau$.

Figure 2 gives the inference rules for dynamic typing as well as signatures for each of the primitive coercions. Each inference rule allows for appropriate coercions at value creation and usage points. For example, the hypothesis of [TRUE1] requires a coercion with signature $\texttt{Bool} \leadsto \tau$. The coercion BOOL! : $\texttt{Bool} \leadsto \texttt{Dynamic}$ satisfies this hypothesis. However, we also wish to allow a value to remain untagged if possible. We introduce a new, *improper coercion* NOOP with signature $\tau \leadsto \tau$. Semantically, NOOP is the identity function. It is easy to verify that every use of coercions in an inference rule admits NOOP and the one proper coercion appropriate to that rule.

We briefly describe the function of each rule in Figure 3. The [ASSUME1] rule is standard. The [ABS1] rule constructs a lambda abstraction and possibly tags it. The coercions NOOP and FUNC! can satisfy the hypothesis of [ABS1].

The [APP1] rule is interesting. The coercions NOOP and FUNC? can satisfy the rule's hypothesis. These two possible coercions dictate the possible types for the function expression e. If the coercion NOOP is used, then e has a function type $\tau \to \tau'$. If the coercion FUNC? is used, then e has type `Dynamic`. In other words, the system allows the check-and-untag operation to be omitted only in the case that e is known to be an untagged function value. As discussed in Section 1, if the function has type `Dynamic` then the argument and result must also have type `Dynamic`.

The coercions NOOP and BOOL? can satisfy the hypothesis of the [COND1] rule. The check-and-untag operation on the predicate is only omitted in the case that the predicate is provably an untagged boolean value. Note that the two branches of the conditional are required to have the same type; this restriction guarantees that the values produced by the branches are either both tagged or both untagged.

There is a final minor issue. According to our definition of correctness, the final result of evaluation of an expression must yield a tagged value, just as the canonical completion does. Thus, we require that the conclusion of a complete derivation be $A \vdash_D e : \texttt{Dynamic}$. Figure 3 gives a complete derivation of one of the minimal completions in Figure 1.

$$\frac{}{A; x : \tau \vdash_D x : \tau} \qquad [ASSUME1]$$

$$\frac{\begin{array}{c} A; x : \tau \vdash_D e : \tau' \\ C : (\tau \to \tau') \leadsto \tau'' \end{array}}{A \vdash_D C\ (\lambda x.e) : \tau''} \qquad [ABS1]$$

$$\frac{\begin{array}{c} A \vdash_D e : \tau \\ A \vdash_D e' : \tau' \\ C : \tau \leadsto (\tau' \to \tau'') \end{array}}{A \vdash_D (C\ e)\ e' : \tau''} \qquad [APP1]$$

$$\frac{\begin{array}{c} A \vdash_D e : \tau \\ A \vdash_D e' : \tau' \\ A \vdash_D e'' : \tau' \\ C : \tau \leadsto \texttt{Bool} \end{array}}{A \vdash_D (\texttt{if}\ (C\ e)\ e'\ e'') : \tau'} \qquad [COND1]$$

$$\frac{C : \texttt{Bool} \leadsto \tau}{A \vdash_D C\ \texttt{true} : \tau} \qquad [TRUE1]$$

$$\frac{C : \texttt{Bool} \leadsto \tau}{A \vdash_D C\ \texttt{false} : \tau} \qquad [FALSE1]$$

FUNC!	:	$(\texttt{Dynamic} \to \texttt{Dynamic}) \leadsto \texttt{Dynamic}$
FUNC?	:	$\texttt{Dynamic} \leadsto (\texttt{Dynamic} \to \texttt{Dynamic})$
BOOL!	:	$\texttt{Bool} \leadsto \texttt{Dynamic}$
BOOL?	:	$\texttt{Dynamic} \leadsto \texttt{Bool}$
NOOP	:	$\tau \leadsto \tau$

Figure 2: Type rules for the dynamically typed lambda calculus.

$$\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\dfrac{\text{NOOP} : \texttt{Bool} \leadsto \texttt{Bool}}{\vdash_D \text{NOOP}\;\texttt{true} : \texttt{Bool}} \quad \dfrac{\begin{array}{c}\dfrac{\text{BOOL!} : \texttt{Bool} \leadsto \texttt{Dynamic}}{x : \texttt{Dynamic} \vdash_D \text{BOOL!}\;\texttt{true} : \texttt{Dynamic}} \\ \text{FUNC!} : \texttt{Dynamic} \rightarrow \texttt{Dynamic} \leadsto \texttt{Dynamic}\end{array}}{\vdash_D \text{FUNC!}\;(\lambda x.\text{BOOL!}\;\texttt{true}) : \texttt{Dynamic}} \quad \dfrac{\text{BOOL!} : \texttt{Bool} \leadsto \texttt{Dynamic}}{\vdash_D \text{BOOL!}\;\texttt{false} : \texttt{Dynamic}} \\ \text{NOOP} : \texttt{Bool} \leadsto \texttt{Bool}\end{array}}{\vdash_D \texttt{if}\;(\text{NOOP}\;(\text{NOOP}\;\texttt{true}))\;(\text{FUNC!}\;(\lambda x.\text{BOOL!}\;\texttt{true}))\;(\text{BOOL!}\;\texttt{false}) : \texttt{Dynamic}} \\ \text{FUNC?} : \texttt{Dynamic} \leadsto \texttt{Dynamic} \rightarrow \texttt{Dynamic} \\ \dfrac{\text{BOOL!} : \texttt{Bool} \leadsto \texttt{Dynamic}}{\vdash_D \text{BOOL!}\;\texttt{false} : \texttt{Dynamic}}\end{array}}{\vdash_D\;(\text{FUNC?}\;\texttt{if}\;(\text{NOOP}\;(\text{NOOP}\;\texttt{true}))\;(\text{FUNC!}\;(\lambda x.\text{BOOL!}\;\texttt{true}))\;(\text{BOOL!}\;\texttt{false}))\;(\text{BOOL!}\;\texttt{false}) : \texttt{Dynamic}}$$

Figure 3: $\vdash_D$ derivation of an example in Figure 1

3 A Subtyping System

Our goal is to explain dynamic typing using subtyping. At first glance, there appears to be no problem. The type `Dynamic` clearly plays a rule akin to a *universal* type—a type of all values. Thus, one expects that

$$\tau \leq \texttt{Dynamic}$$

for all types τ.

However, there is a serious difficulty. Consider a conditional $\texttt{if}\; e\; e'\; e''$ and let $e' : \texttt{Bool} \rightarrow \texttt{Bool}$ and $e'' : \texttt{Dynamic} \rightarrow \texttt{Dynamic}$. Now, by subtyping $\texttt{Bool} \rightarrow \texttt{Bool} \leq \texttt{Dynamic}$ and $\texttt{Dynamic} \rightarrow \texttt{Dynamic} \leq \texttt{Dynamic}$, and so we can conclude that

$$\texttt{if}\; e\; e'\; e'' : \texttt{Dynamic}$$

assuming e has type `Bool`. Unfortunately, this conclusion is unsound, because the two expressions e' and e'' have different behavior and cannot be used in the same context (e.g., e' expects an untagged argument and e'' expects a tagged argument). Thus, $\texttt{Bool} \rightarrow \texttt{Bool} \leq \texttt{Dynamic}$ and $\texttt{Dynamic} \rightarrow \texttt{Dynamic} \leq \texttt{Dynamic}$ cannot both hold, so `Dynamic` is anything but a universal type. In dynamic typing, $\texttt{Bool} \rightarrow \texttt{Bool} \leq \texttt{Dynamic}$ does not hold; in this example, e' must be coerced to have type $\texttt{Dynamic} \rightarrow \texttt{Dynamic}$.

A different approach is needed to encode dynamic typing in a subtyping system. The intuition behind our solution follows from the definition of the semantic domain D:

$$D = ((D \rightarrow D) + \mathit{Bool}) \times (\mathit{notag} + \mathit{tag})$$

A semantic value consists of two parts: the "real" value and a tag, which is possibly absent. Thus, we represent types as pairs $[\pi, \rho]$, where π is the structural part of the type and ρ represents the tag. Formally, the types of our system are generated by the following grammar:

$$\begin{array}{rcl} \sigma & ::= & [\pi, \rho] \\ \pi & ::= & \alpha \mid \sigma \rightarrow \sigma \mid \texttt{Bool} \mid \pi \cup \pi' \mid \pi \cap \pi' \mid 0 \\ \rho & ::= & \beta \mid \texttt{tag} \mid \texttt{notag} \mid \rho \cup \rho' \end{array}$$

Types denote sets of values. For example, $\sigma \rightarrow \sigma'$ denotes the set of functions mapping arguments of type σ to results of type σ'. The expressions $\pi \cup \pi'$ and $\pi \cap \pi'$ denote set-theoretic union and intersection of types. The expression 0 represents non-termination (formally, it is the set $\{\perp\}$) and is the least type; i.e., $0 \cap \pi = 0$ and $0 \cup \pi = \pi$ for any π. For brevity, we skip the development of ideal models needed to formalize types as sets of values; the construction is well-known (e.g., see [MPS84, AW93]).

We work with systems of *set constraints* of the following forms:

$$\begin{array}{rcl} X & \subseteq & Y \\ Q & \neq & 0 \\ Q & \neq & \texttt{tag} \cup \texttt{notag} \\ T \neq 0 & \Rightarrow & Q \subseteq R \end{array}$$

Here X, Y stand for any expressions drawn from the grammar above. Q and R refer to tag expressions (grammar symbol ρ), T refers to type expressions (grammar symbol τ). The interpretation of these constraints is conventional. Given a set S of constraints a *solution* of S is a mapping of variables to types such that all of the constraints are simultaneously satisfied.

We do not include an explicit fixed point operator because recursive constraints have equivalent power. Let $X = Y$ denote the pair of constraints $X \subseteq Y$ and $Y \subseteq X$. For example, the set of fully tagged values can be defined as the unique solution of the recursive equation:

$$[\alpha, \beta] = [([\alpha, \beta] \rightarrow [\alpha, \beta]) \cup \texttt{Bool}, \texttt{tag}]$$

We use χ to denote the set of fully tagged values. Similarly, the set of all values (tagged and untagged) is the unique solution of:

$$[\alpha, \beta] = [([\alpha, \beta] \rightarrow [\alpha, \beta]) \cup \texttt{Bool}, \texttt{tag} \cup \texttt{notag}]$$

We use 1 to denote the set of all values.

Before presenting the inference rules, there are further details meriting discussion. In the grammar for types, the intent is that a variable α ranges over types of kind π and that a variable β ranges over types of kind ρ. A standard mechanism for enforcing such restrictions is to use a many-sorted algebra. However, it is possible to avoid the extra notational burden of many-sorted algebras by using constraints. Variables of kind α and β have the following associated constraints:

$$\begin{array}{rcl} \alpha & \subseteq & (1 \rightarrow 1) \cup \texttt{Bool} \\ \beta & \subseteq & \texttt{tag} \cup \texttt{notag} \\ \beta & \neq & 0 \end{array}$$

Thus, an α variable always denotes the structural part of a type and a β variable always denotes `tag`, `notag`, or both. For conciseness, these constraints are left implicit in inference rules and examples.

The inference rules and coercions for the set constraint system are given in Figure 4. The system infers facts of the form $A, S \vdash_S e : \sigma$. Informally, the meaning of this derivation is that e has the type $s(\sigma)$ for every mapping s that is a solution of the constraints S. The following lemma makes this precise.

$$\frac{}{A; x : [\pi, \rho],\ S \vdash_S x : [\pi, \rho]} \qquad [ASSUME2]$$

$$\frac{\begin{array}{l} A; x : [\pi, \rho],\ S \vdash_S e : [\pi', \rho'] \\ S' = S \cup \{\rho'' \neq \mathtt{tag} \cup \mathtt{notag}\} \\ C : [\kappa, \mathtt{notag}] \rightsquigarrow [\kappa, \rho''] \text{ where } \kappa = [\pi, \rho] \to [\pi', \rho'] \end{array}}{A, S' \vdash_S C\ (\lambda x.e) : [\kappa, \rho'']} \qquad [ABS2]$$

$$\frac{\begin{array}{l} A, S \vdash_S e : [\pi, \rho] \\ A, S \vdash_S e' : [\pi', \rho'] \\ S' = S \cup \left\{ \begin{array}{l} \pi \subseteq ([\pi', \rho'] \to [\pi'', \rho'']) \cup (\alpha \cap \mathtt{Bool}) \\ (\alpha \cap \mathtt{Bool}) \neq 0 \Rightarrow \rho = \mathtt{tag} \\ \rho \neq \mathtt{tag} \cup \mathtt{notag} \end{array} \right\} \\ C : [\kappa \cup (\alpha \cap \mathtt{Bool}), \rho] \rightsquigarrow [\kappa, \mathtt{notag}] \text{ where } \kappa = [\pi', \rho'] \to [\pi'', \rho''] \end{array}}{A, S' \vdash_S (C\ e)\ e' : [\pi'', \rho'']} \qquad [APP2]$$

$$\frac{\begin{array}{l} A, S \vdash_S e : [\pi, \rho] \\ A, S \vdash_S e' : [\pi', \rho'] \\ A, S \vdash_S e'' : [\pi'', \rho''] \\ S' = S \cup \left\{ \begin{array}{l} \pi \subseteq \mathtt{Bool} \cup (\alpha \cap (1 \to 1)) \\ \alpha \cap (1 \to 1) \neq 0 \Rightarrow \rho = \mathtt{tag} \\ \rho \neq \mathtt{tag} \cup \mathtt{notag} \end{array} \right\} \\ C : [\mathtt{Bool} \cup (\alpha \cap (1 \to 1)), \rho] \rightsquigarrow [\mathtt{Bool}, \mathtt{notag}] \end{array}}{A, S' \vdash_S (\mathtt{if}\ (C\ e)\ e'\ e'') : [\pi' \cup \pi'', \rho' \cup \rho'']} \qquad [COND2]$$

$$\frac{\begin{array}{l} C : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \rho] \\ S \supseteq \{\rho \neq \mathtt{tag} \cup \mathtt{notag}\} \end{array}}{A, S \vdash_S C\ \mathtt{true} : [\mathtt{Bool}, \rho]} \qquad [TRUE2]$$

$$\frac{\begin{array}{l} C : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \rho] \\ S \supseteq \{\rho \neq \mathtt{tag} \cup \mathtt{notag}\} \end{array}}{A, S \vdash_S C\ \mathtt{false} : [\mathtt{Bool}, \rho]} \qquad [FALSE2]$$

$$\begin{array}{rcl} \text{FUNC!} & : & [\sigma \to \sigma', \mathtt{notag}] \rightsquigarrow [\sigma \to \sigma', \mathtt{tag}] \\ \text{FUNC?} & : & [(\sigma \to \sigma') \cup \mathtt{Bool}, \mathtt{tag}] \rightsquigarrow [\sigma \to \sigma', \mathtt{notag}] \\ \text{BOOL!} & : & [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{tag}] \\ \text{BOOL?} & : & [\mathtt{Bool} \cup (1 \to 1), \mathtt{tag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{notag}] \\ \text{NOOP} & : & \sigma \rightsquigarrow \sigma \end{array}$$

Figure 4: Type rules using set constraints.

$$\frac{\begin{array}{l} \dfrac{\begin{array}{c} \dfrac{\text{BOOL!} : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{tag}]}{\vdash_S \text{BOOL! } \mathtt{false} : [\mathtt{Bool}, \mathtt{tag}]} \qquad \dfrac{\dfrac{\text{BOOL!} : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{tag}]}{\begin{array}{l} x : [\mathtt{Bool}, \mathtt{notag}] \vdash_S \text{BOOL! } \mathtt{true} : [\mathtt{Bool}, \mathtt{tag}] \\ \text{FUNC!} : [\kappa, \mathtt{notag}] \rightsquigarrow [\kappa, \mathtt{tag}] \\ \text{where } \kappa = [\mathtt{Bool}, \mathtt{notag}] \to [\mathtt{Bool}, \mathtt{tag}] \end{array}}}{\vdash_S \text{FUNC! } (\lambda x.\text{BOOL! } \mathtt{true}) : [\kappa, \mathtt{tag}]} \\ \dfrac{\text{NOOP} : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{notag}]}{\vdash_S \text{NOOP } \mathtt{true} : [\mathtt{Bool}, \mathtt{notag}]} \\ \text{NOOP} : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{notag}] \end{array}}{\vdash_S \mathtt{if}\ (\text{NOOP } (\text{NOOP } \mathtt{true}))\ (\text{FUNC! } (\lambda x.\text{BOOL! } \mathtt{true}))\ (\text{BOOL! } \mathtt{false}) : [\kappa \cup \mathtt{Bool}, \mathtt{tag}]} \\ \text{FUNC?} : [([\mathtt{Bool}, \mathtt{notag}] \to [\mathtt{Bool}, \mathtt{tag}]) \cup \mathtt{Bool}, \mathtt{tag}] \rightsquigarrow [[\mathtt{Bool}, \mathtt{notag}] \to [\mathtt{Bool}, \mathtt{tag}], \mathtt{notag}] \\ \dfrac{\text{NOOP} : [\mathtt{Bool}, \mathtt{notag}] \rightsquigarrow [\mathtt{Bool}, \mathtt{notag}]}{\vdash_S \text{NOOP } \mathtt{false} : [\mathtt{Bool}, \mathtt{notag}]} \end{array}}{\vdash_S (\text{FUNC? } \mathtt{if}\ (\text{NOOP}(\text{NOOP } \mathtt{true}))\ (\text{FUNC! } (\lambda x.\text{BOOL! } \mathtt{true}))\ (\text{BOOL! } \mathtt{false}))\ (\text{NOOP } \mathtt{false}) : [\mathtt{Bool}, \mathtt{tag}]}$$

Figure 5: $\vdash_S$ derivation of an example in Figure 1

Lemma 3.1 (Soundness) Let $A, S \vdash_S e : \sigma$, let s be any solution of the constraints S, and let v be the semantic value denoted by e in some environment E. If $E(x) \in s(A(x))$ for every free variable x of e, then $v \in s(\sigma)$.

We will not prove this lemma, but instead briefly discuss each rule. Note that coercions in this system affect the tag component of a type. For example, the tagging coercions FUNC! and BOOL! simply change a tag from `notag` to `tag`. The inverse coercions FUNC? and BOOL? both change the tag component from `tag` to `notag` (reflecting the untagging of the value) and restrict the structural component of the type (reflecting the possible values after a successful type test).

The [ASSUME2] rule is straightforward. The [ABS2] rule is the standard lambda abstraction rule, except that the tag ρ'' depends on the type of the coercion C. If C is an improper coercion NOOP : $[\kappa, \texttt{notag}] \leadsto [\kappa, \texttt{notag}]$ then $\rho'' = \texttt{notag}$. If C is the proper coercion FUNC! : $[\kappa, \texttt{notag}] \leadsto [\kappa, \texttt{tag}]$ then $\rho'' = \texttt{tag}$. For the coercion C to be well-defined, the variable ρ must stand for either `tag` or `notag` but not both. In all coercions, a constraint $\rho \neq \texttt{tag} \cup \texttt{notag}$ is associated with ρ.

The rule [APP2] illustrates the crux of our system. Consider an application $(C\ e)\ e'$ and let $e : [\pi, \rho]$ and $e' : [\pi', \rho']$. Now, there is no requirement that e be provably a function—that is, π need not be a function type. We want to know two things: (1) whether π is guaranteed to be a function type and (2) what function types are in π. The constraint

$$\pi \subseteq ([\pi', \rho'] \rightarrow [\pi'', \rho'']) \cup (\alpha \cap \texttt{Bool})$$

accomplishes both goals. Any solution of this constraint divides the type π into its function values $[\pi', \rho'] \rightarrow [\pi'', \rho'']$ and non-function values $\alpha \cap \texttt{Bool}$. If $\alpha \cap \texttt{Bool} = 0$ in any solution of the constraints, then the constraint simplifies to

$$\pi \subseteq [\pi', \rho'] \rightarrow [\pi'', \rho'']$$

and thus π contains only functions, implying e can only evaluate to function values by Lemma 3.1. However, if $\alpha \cap (\texttt{Bool}) \neq 0$ in all solutions of the constraints, then we cannot guarantee statically that π is a function and it is necessary to test at runtime. The constraint

$$(\alpha \cap \texttt{Bool}) \neq 0 \Rightarrow \rho = \texttt{tag}$$

forces the value to be tagged and the coercion in the application to be FUNC? whenever π may contain non-functions.

The [COND2] rule works analogously to the [APP2] rule. The constraint $\pi \subseteq \texttt{Bool} \cup (\alpha \cap (1 \rightarrow 1))$ forces any non-boolean values to be assigned to α in any solution. Thus, if $\alpha \cap (1 \rightarrow 1) = 0$, the predicate is guaranteed to be a boolean. However, if $\alpha \cap (1 \rightarrow 1) \neq 0$, then the predicate may not be a boolean and dynamic type checking is required. The constraint $\alpha \cap (1 \rightarrow 1) \neq 0 \Rightarrow \rho = \texttt{tag}$ forces the value of the predicate to be tagged in this case.

There is another aspect of the [COND2] rule worth noting. The inferred type $[\pi' \cup \pi'', \rho' \cup \rho'']$ potentially has both tagged and untagged values (e.g., if $\rho' = \texttt{tag}$ and $\rho'' = \texttt{notag}$). In contrast to the situation with dynamic typing (see the beginning of the section), this is sound. Only the [APP2] and [COND2] rules inspect tags and both rules require the tag component to be exactly `tag`. Values of type $[\pi, \texttt{tag} \cup \texttt{notag}]$ can never satisfy the constraints. Thus, a value of type $[\pi, \texttt{tag} \cup \texttt{notag}]$ can be created, but never used.

A remaining detail is guaranteeing that the result of evaluation produces a value in which all components of the type are tagged. Recall that the type of fully tagged values is χ. If the final type of a program is σ, then adding the constraint $\sigma \subseteq \chi$ forces the result to be completely tagged. We can now state that the system infers correct completions.

Lemma 3.2 Let $\emptyset, S \vdash_S e : \sigma$ where the system of constraints $S = S' \cup \{\sigma \subseteq \chi\}$ is consistent. Let $e' = \textit{erase}(e)$. Then e is a correct completion of e'.

Proof: [sketch] The previous discussion presents the proof informally. The formal argument uses soundness (Lemma 3.1) and the form of the constraints to show that the completion has the same meaning as the canonical completion. □

Figure 5 gives an example of a derivation in the set constraint system of a term from Figure 1. The constraints are elided for readability. The most interesting step in the derivation is at the function abstraction, which creates a tagged function taking an untagged argument.

4 Comparison

This section presents our main result: every completion derivable in the dynamic typing system is derivable in the set constraint system. The converse does not hold (see Figure 1), although we show in Section 6 that the set constraint system can be restricted to have exactly the same power as dynamic typing.

Because the two systems use different domains of types, we require a translation function. The function T maps types τ to types σ:

$$\begin{array}{rcl} T(\tau \rightarrow \tau') &=& [T(\tau) \rightarrow T(\tau'), \texttt{notag}] \\ T(\texttt{Bool}) &=& [\texttt{Bool}, \texttt{notag}] \\ T(\texttt{Dynamic}) &=& \chi \\ T(\text{fix}\,\alpha.\tau) &=& \text{solution of } [\pi_\alpha, \rho_\alpha] = T(\tau) \\ T(\alpha) &=& [\pi_\alpha, \rho_\alpha] \end{array}$$

A type variable α is translated to a pair $[\pi_\alpha, \rho_\alpha]$, where π_α and ρ_α are set variables uniquely associated with α. We extend T to type environments in the obvious way:

$$\begin{array}{rcl} T(A; x : \tau) &=& T(A); x : T(\tau) \\ T(\emptyset) &=& \emptyset \end{array}$$

Note that T preserves tags; that is, T maps tagged types to tagged types and untagged types to untagged types.

Theorem 4.1 Let e be an expression of the dynamically typed lambda calculus and let A be a type environment. Then

$$A \vdash_D e : \tau \ \Rightarrow\ T(A), S \vdash_S e : \sigma$$

for some $\sigma \subseteq T(\tau)$ and consistent system S of constraints.

Proof: The proof is by induction on the structure of the derivation showing $A \vdash_D e : \tau$. We present this proof in detail.

1. Assume $A; x : \tau \vdash_D x : \tau$. Using rule [ASSUME2], it follows immediately that

   $$T(A); x : T(\tau), S \vdash_S x : T(\tau)$$

   By the definition of T, we have

   $$T(A; x : \tau), S \vdash_D x : T(\tau)$$

   for any consistent system S of constraints.

2. Assume $A \vdash_D C\ (\lambda x.e) : \tau''$. Then $A; x : \tau \vdash_D e : \tau'$ and $C : (\tau \rightarrow \tau') \leadsto \tau''$. By induction, we know $T(A; x : \tau), S \vdash_S e : \sigma$ where $\sigma \subseteq T(\tau')$, from which it follows that

   $$T(A); x : T(\tau), S \vdash_S e : \sigma$$

To prove the result, we must show that

$$T(A), S \vdash_S C\ \lambda x.e : [T(\tau) \to \sigma, \rho'']$$

for some choice of ρ'' where the coercion C has an appropriate type and $[T(\tau) \to \sigma, \rho''] \subseteq T(\tau'')$. The constraints $\rho'' \neq 0 \wedge \rho'' \neq \mathtt{tag} \cup \mathtt{notag}$ imply that $\rho'' = \mathtt{tag} \vee \rho'' = \mathtt{notag}$. Thus there are two subcases.

The first subcase is $C =$ FUNC!, in which case $\tau = \tau' = \tau'' = \mathtt{Dynamic}$. The tag ρ'' in the [ABS2] inference rule is not constrained to be either $\mathtt{tag}$ or $\mathtt{notag}$. Therefore, letting $\rho'' = \mathtt{tag}$ we have

$$\text{FUNC!} : [T(\tau) \to \sigma, \mathtt{notag}] \rightsquigarrow [T(\tau) \to \sigma, \mathtt{tag}]$$

Since all premises of the [ABS2] rule are satisfied, we conclude

$$T(A), S \vdash_S \text{FUNC! } \lambda x.e : [T(\tau) \to \sigma, \mathtt{tag}]$$

To complete this case, note that

$$\begin{array}{lll} & [T(\tau) \to \sigma, \mathtt{tag}] & \\ \subseteq & [T(\tau) \to T(\tau'), \mathtt{tag}] & \text{since } \sigma \subseteq T(\tau') \\ = & [\chi \to \chi, \mathtt{tag}] & \text{definition of } T \\ \subseteq & \chi & \text{definition of } \chi \\ = & T(\mathtt{Dynamic}) & \text{definition of } T \\ = & T(\tau'') & \end{array}$$

The second subcase is $C =$ NOOP, where $\tau'' = \tau \to \tau'$. Letting $\rho'' = \mathtt{notag}$ we have

$$\text{NOOP} : [T(\tau) \to \sigma, \mathtt{notag}] \rightsquigarrow [T(\tau) \to \sigma, \mathtt{notag}]$$

and, since the premises of [ABS2] are satisfied,

$$T(A), S \vdash_S \text{NOOP } \lambda x.e : [T(\tau) \to \sigma, \mathtt{notag}]$$

To complete this subcase, note that

$$\begin{array}{lll} & [T(\tau) \to \sigma, \mathtt{notag}] & \\ \subseteq & [T(\tau) \to T(\tau'), \mathtt{notag}] & \text{since } \sigma \subseteq T(\tau') \\ = & T(\tau \to \tau') & \text{definition of } T \\ = & T(\tau'') & \end{array}$$

3. Assume that $A \vdash_D (C\ e)\ e' : \tau''$. By the premises of the [APP1] rule, we know

$$\begin{array}{l} A \vdash_D e : \tau \\ A \vdash_D e' : \tau' \\ C : \tau \rightsquigarrow (\tau' \to \tau'') \end{array}$$

By induction, it follows that

$$\begin{array}{ll} T(A), S \vdash_S e : [\pi, \rho] & \text{where } [\pi, \rho] \subseteq T(\tau) \\ T(A), S \vdash_S e' : [\pi', \rho'] & \text{where } [\pi', \rho'] \subseteq T(\tau') \end{array}$$

To prove the theorem, we must show that

$$T(A), S' \vdash_S (C\ e)\ e' : [\pi'', \rho'']$$

where $[\pi'', \rho''] \subseteq T(\tau'')$, the coercion C has an appropriate type, and

$$S' = S \cup \left\{ \begin{array}{l} \pi \subseteq ([\pi', \rho'] \to [\pi'', \rho'']) \cup (\alpha \cap \mathtt{Bool}) \\ (\alpha \cap \mathtt{Bool}) \neq 0 \Rightarrow \rho = \mathtt{tag} \\ \rho \neq \mathtt{tag} \cup \mathtt{notag} \end{array} \right\}$$

for some ρ'', π'', and α where the constraints are satisfied. As before, there are two subcases.

The first subcase is $C =$ FUNC?. Therefore $\tau = \tau' = \tau'' = \mathtt{Dynamic}$. Let $[\pi'', \rho''] = T(\tau'') = \chi$ and let $\alpha = \mathtt{Bool}$. Furthermore, $\rho = \mathtt{tag}$ since $[\pi, \rho] \subseteq T(\tau) = \chi$. Since $\alpha \cap \mathtt{Bool} = \mathtt{Bool}$ we have

$$\begin{array}{l} \text{FUNC?} : [\kappa \cup (\alpha \cap \mathtt{Bool}), \mathtt{tag}] \rightsquigarrow [\kappa, \mathtt{notag}] \\ \quad \text{where } \kappa = [\pi', \rho'] \to [\pi'', \rho''] \end{array}$$

In addition, because $\rho = \mathtt{tag}$, the second constraint is satisfied. To finish the subcase, we show that the first constraint is satisfied. The following argument uses the fact that function types are anti-monotonic in the argument position; that is, $x \subseteq y$ implies $y \to z \subseteq x \to z$.

$$\begin{array}{lll} & \pi \subseteq ([\pi', \rho'] \to [\pi'', \rho'']) \cup (\alpha \cap \mathtt{Bool}) & \\ \Leftrightarrow & \pi \subseteq ([\pi', \rho'] \to \chi) \cup (\mathtt{Bool} \cap \mathtt{Bool}) & \text{substitution} \\ \Leftrightarrow & \pi \subseteq ([\pi', \rho'] \to \chi) \cup \mathtt{Bool} & \text{simplification} \\ \Leftarrow & (\chi \to \chi) \cup \mathtt{Bool} \subseteq ([\pi', \rho'] \to \chi) \cup \mathtt{Bool} & \\ & & \text{since } [\pi, \rho] \subseteq \chi \\ \Leftarrow & (\chi \to \chi) \cup \mathtt{Bool} \subseteq (\chi \to \chi) \cup \mathtt{Bool} & \\ & & \text{since } [\pi', \rho'] \subseteq \chi \end{array}$$

It follows that $A, S' \vdash_S (\text{FUNC? } e)\ e' : [\pi'', \rho'']$.

The second subcase is $C =$ NOOP. Therefore $\tau = \tau' \to \tau''$. Let $[\pi'', \rho''] = T(\tau'')$ and let $\alpha = 0$. Since $[\pi, \rho] \subseteq T(\tau' \to \tau'')$ it follows that $\rho = \mathtt{notag}$. Because $\alpha \cap \mathtt{Bool} = 0$, we have

$$\begin{array}{c} \text{NOOP} : [\kappa, \mathtt{notag}] \rightsquigarrow [\kappa, \mathtt{notag}] \\ \text{where } \kappa = [\pi', \rho'] \to [\pi'', \rho''] \end{array}$$

The second constraint is satisfied, also because $\alpha \cap \mathtt{Bool} = 0$. To see that the first constraint is satisfied, note that

$$\begin{array}{lll} & \pi \subseteq ([\pi', \rho'] \to [\pi'', \rho'']) \cup (\alpha \cap \mathtt{Bool}) & \\ \Leftrightarrow & \pi \subseteq ([\pi', \rho'] \to T(\tau'')) \cup (0 \cap \mathtt{Bool}) & \text{substitution} \\ \Leftrightarrow & \pi \subseteq [\pi', \rho'] \to T(\tau'') & \text{simplification} \\ \Leftarrow & \pi \subseteq T(\tau') \to T(\tau'') & [\pi', \rho'] \subseteq T(\tau') \\ \Leftrightarrow & [\pi, \rho] \subseteq [T(\tau') \to T(\tau''), \mathtt{notag}] & \rho = \mathtt{notag} \\ \Leftrightarrow & [\pi, \rho] \subseteq T(\tau' \to \tau'') & \text{definition of } T \\ \Leftrightarrow & [\pi, \rho] \subseteq T(\tau) & \text{assumption} \\ \Leftrightarrow & \text{true} & \text{by induction} \end{array}$$

It follows that $A, S' \vdash_S (\text{NOOP } e)\ e' : [\pi'', \rho'']$.

4. Assume $A \vdash_D (\mathtt{if}\ (C\ e)\ e'\ e'') : \tau'$. From the premises of the [COND1] rule, we know

$$\begin{array}{l} A \vdash_D e : \tau \\ A \vdash_D e' : \tau' \\ A \vdash_D e'' : \tau' \\ C : \tau \rightsquigarrow \mathtt{Bool} \end{array}$$

By induction, it follows that

$$\begin{array}{l} T(A), S \vdash_S e : [\pi, \rho] \subseteq T(\tau) \\ T(A), S \vdash_S e' : [\pi', \rho'] \subseteq T(\tau') \\ T(A), S \vdash_S e'' : [\pi'', \rho''] \subseteq T(\tau') \end{array}$$

Thus, to prove the result it suffices to show that

$$T(A), S' \vdash_S \mathtt{if}\ (C\ e)\ e'\ e'' : [\pi' \cup \pi'', \rho' \cup \rho'']$$

where $[\pi' \cup \pi'', \rho' \cup \rho''] \subseteq T(\tau')$, the coercion C has an appropriate type, and

$$S' = S \cup \left\{ \begin{array}{l} \pi \subseteq \mathtt{Bool} \cup (\alpha \cap 1 \to 1) \\ (\alpha \cap (1 \to 1)) \neq 0 \Rightarrow \rho = \mathtt{tag} \\ \rho \neq \mathtt{tag} \cup \mathtt{notag} \end{array} \right\}$$

for some α that satisfies the constraints.

First note that $\rho' = \rho''$, because $[\pi', \rho'] \subseteq T(\tau')$ and $[\pi'', \rho''] \subseteq T(\tau')$ and $T(\tau')$ has the form $[x, \texttt{tag}]$ or $[x, \texttt{notag}]$. Therefore,

$$[\pi' \cup \pi'', \rho' \cup \rho''] = [\pi', \rho'] \cup [\pi'', \rho''] \subseteq T(\tau')$$

The rest of the argument breaks into the usual two cases. Assume $C =$ BOOL?. Then $\tau = \texttt{Dynamic}$. Let $\alpha = 1 \rightarrow 1$. Because $[\pi, \rho] \subseteq T(\tau)$, it follows that $[\pi, \rho] \subseteq \chi$, so $\rho = \texttt{tag}$. Since $\alpha \cap (1 \rightarrow 1) = 1 \rightarrow 1$, we have

$$\text{BOOL?} : [\texttt{Bool} \cup (1 \rightarrow 1), \texttt{tag}] \leadsto [\texttt{Bool}, \texttt{notag}]$$

Showing the constraints are satisfied is very similar to the corresponding subcase for application.

Now assume $C =$ NOOP. Then $\tau = \texttt{Bool}$. Let $\alpha = 0$. Because $[\pi, \rho] \subseteq T(\tau)$, it follows that $\pi \subseteq \texttt{Bool}$ and $\rho = \texttt{notag}$. Since $\alpha \cap (1 \rightarrow 1) = 0$, we have

$$\text{NOOP} : [\texttt{Bool}, \texttt{notag}] \leadsto [\texttt{Bool}, \texttt{notag}]$$

Again, showing the constraints are satisfied is very similar to the corresponding subcase for application.

5. Assume $A \vdash_D C\ \texttt{true} : \tau$. If $C =$ BOOL!, then

$$T(A), S \vdash_S \texttt{true} : [\texttt{Bool}, \texttt{tag}]$$

satisfies the theorem for any consistent system of constraints S. If $C =$ NOOP, then

$$T(A), S \vdash_S \texttt{true} : [\texttt{Bool}, \texttt{notag}]$$

satisfies the theorem.

6. Assume $A \vdash_D C\ \texttt{false} : \tau$. This case is the same as the case for $\texttt{true}$.

□

From the theorem, we immediately have the following corollary.

Corollary 4.2 Let e be any closed term without coercions. If e' is a completion of e derivable in $\vdash_D$, then e' is also derivable in $\vdash_S$.

Proof: Follows from Theorem 4.1 and the fact that T preserves tags. □

5 Computing Minimal Completions

Type inference for the system in Figure 4 can be implemented in time $\mathcal{O}(n^3)$ in the size of the expression. The bound is the worst case and, in fact, we expect the algorithm performs significantly better in practice, although it cannot be as efficient as the algorithms for dynamic typing.

The algorithm is divided into four phases:

1. Constraint generation.
2. Constraint resolution.
3. Tag variable instantiation.
4. Program completion.

The first phase is straightforward. The proof system in Figure 4 is run, but the coercions are left as unknowns. For the result of each potential coercion, fresh variables (unknowns) are inserted. The constraints are generated using fresh variables in every rule. The solutions of the resulting system S of constraints for the entire expression characterize all possible completions. This phase is linear in the size of the expression.

To discover which completions are possible, it is necessary to solve the constraints. Figure 6 gives a set of rewrite rules that, when applied until closure (until no new constraints can be generated), reduce a system of constraints to solved form. These constraint resolution rules are essentially those of [MR85, Hei92, AW93] specialized to our application. The soundness of these rules can be proven using standard techniques (e.g., see [AW92, AW93]). In Figure 6, κ stands for an arbitrary type expression and γ stands for an arbitrary variable.

Rules 10 and 11 of Figure 6 appear non-constructive, but are actually easy to implement. For Rule 10, in the process of rewriting the constraint system it may be discovered that $\pi \neq 0$—due to non-zero lower bounds on variables in π—in which case the rule can be applied. Once no constraints can be added, any remaining implication constraints can be deleted using Rule 11. A detailed justification is presented in [Hei92].

Rules 12 and 13 take advantage of the special structure of constraints involving tag variables. Rule 12 expresses the fact that if a tag variable β is known to be a subset of $\texttt{tag}$, then $\beta = \texttt{tag}$, since no tag variable can be 0. Rule 13 says that if β includes at least $\texttt{tag}$ but not both $\texttt{tag}$ and $\texttt{notag}$, then $\beta = \texttt{tag}$. Note that these special-purpose rules could be factored into a separate phase, but it is convenient to present them as part of the overall constraint resolution.

Constraint resolution is the most expensive phase. The rewrite rules work only with pairs of subexpressions of the original constraint system. Thus, the rules can produce at most $\mathcal{O}(n^2)$ constraints, where original system has size $\mathcal{O}(n)$. Each rule requires constant time to apply, with the exception of Rule 9, which may require $\mathcal{O}(n)$ time to examine all the upper and lower bounds of a variable. Thus, the entire resolution process requires at most $\mathcal{O}(n^3)$ time.

Constraint resolution does not necessarily yield a unique completion, as some tag variables may be unconstrained. However, all upper and lower bounds on variables in the resolved system are explicit, so it is easy to discern the possible solutions by inspection of the constraints. Let S be the system of resolved constraints. The third phase adds constraints to tag variables to produce a minimal completion using the following rule:

$$\text{If } \beta \subseteq \texttt{tag} \text{ is not in } S\text{, then add } \texttt{notag} \subseteq \beta \text{ to } S \qquad (14)$$

This rule adds a lower-bound of $\texttt{notag}$ to all tag variables that are not constrained to be equal to $\texttt{tag}$. It is easy to see that if $\texttt{notag} \subseteq \beta$ in any completion permitted by the constraints, then $\texttt{notag} \subseteq \beta$ according to this rule. This observation proves the existence of minimal completions for the set constraint system. The tag instantiation phase requires inspection of the upper bound of all tag variables, which takes time $\mathcal{O}(n)$.

Constructing the completion of the program is easy. Each coercion has a single tag variable ρ. Now, from the original constraints we know $\rho \neq 0$ (a constraint on all tag variables) and $\rho \neq \texttt{tag} \cup \texttt{notag}$ (a constraint on all tag variables in coercions). Thus, either $\rho = \texttt{tag}$ or $\rho = \texttt{notag}$ in every solution. If $\texttt{tag} \subseteq \rho$, then $\texttt{tag} = \rho$ (Rule 13). If $\texttt{tag} \not\subseteq \rho$, then $\texttt{notag} \subseteq \rho$ (Rule 14). Thus, each coercion is determined by the lower bound of its tag variable.

$$
\begin{array}{rcll}
S \cup \{0 \subseteq \sigma\} & \Rightarrow & S & (1)\\
S \cup \{[\pi,\rho] \subseteq [\pi',\rho']\} & \Rightarrow & S \cup \{\pi \subseteq \pi', \rho \subseteq \rho'\} & (2)\\
S \cup \{\sigma_1 \to \sigma_2 \subseteq \sigma_1' \to \sigma_2'\} & \Rightarrow & S \cup \{\sigma_1' \subseteq \sigma_1, \sigma_2 \subseteq \sigma_2'\} & (3)\\
S \cup \{\kappa \cup \kappa' \subseteq \kappa''\} & \Rightarrow & S \cup \{\kappa \subseteq \kappa'', \kappa' \subseteq \kappa''\} & (4)\\
S \cup \{\kappa \subseteq \kappa' \cap \kappa''\} & \Rightarrow & S \cup \{\kappa \subseteq \kappa', \kappa \subseteq \kappa''\} & (5)\\
S \cup \{\sigma \to \sigma' \subseteq \pi \cup (\kappa \cap \texttt{Bool})\} & \Rightarrow & S \cup \{\sigma \to \sigma' \subseteq \pi\} & (6)\\
S \cup \{\texttt{Bool} \subseteq \pi \cup (\kappa \cap 1 \to 1)\} & \Rightarrow & S \cup \{\texttt{Bool} \subseteq \pi\} & (7)\\
S \cup \{\kappa \subseteq \kappa\} & \Rightarrow & S & (8)\\
S \cup \{\kappa \subseteq \gamma, \gamma \subseteq \kappa'\} & \Rightarrow & S \cup \{\kappa \subseteq \gamma, \gamma \subseteq \kappa', \kappa \subseteq \kappa'\} & (9)\\
S \cup \{\pi \neq 0 \Rightarrow \kappa \subseteq \kappa'\} \text{ and } S \Rightarrow \pi \neq 0 & \Rightarrow & S \cup \{\kappa \subseteq \kappa'\} & (10)\\
S \cup \{\pi \neq 0 \Rightarrow \kappa \subseteq \kappa'\} \text{ and } S \not\Rightarrow \pi \neq 0 & \Rightarrow & S & (11)\\
S \cup \{\beta \neq 0, \beta \subseteq \texttt{tag}\} & \Rightarrow & S \cup \{\texttt{tag} \subseteq \beta, \beta \subseteq \texttt{tag}\} & (12)\\
S \cup \{\beta \neq \texttt{tag} \cup \texttt{notag}, \texttt{tag} \subseteq \beta\} & \Rightarrow & S \cup \{\texttt{tag} \subseteq \beta, \beta \subseteq \texttt{tag}\} & (13)
\end{array}
$$

Figure 6: Rules for simplifying constraints.

Finally, it is important to note that the constraints always have at least one solution, namely $\pi_1 = \pi_2 = \ldots = \chi$ and $\rho_1 = \rho_2 = \ldots = \texttt{tag}$. This solution produces the canonical completion—all values are tagged.

6 Variations

Set constraints are a very expressive and flexible framework for specifying program analyses, making it quite easy to extend analyses in various ways. This section discusses a number of variations on the basic system we have presented. For space reasons, each modification is described only briefly.

6.1 Dynamic Typing Revisited

As discussed in Section 4, the set constraint system is strictly more powerful than dynamic typing. To achieve exactly dynamic typing, we must guarantee that whenever a tagged type arises, all components of the type are also tagged. This condition is easy to express with additional constraints. For each type $[\pi, \rho]$ used in a derivation, add a constraint:

$$(\rho \cap \texttt{tag}) \neq 0 \Rightarrow [\pi, \rho] = \chi$$

When applied to the type in the conclusion of [COND2], this constraint also guarantees that the branches of a conditional are consistently tagged. We state without proof that under these additional constraints, a completion is $\vdash_S$ derivable if and only if it is $\vdash_D$ derivable.

While this observation gives an alternative characterization of dynamic typing, it appears no more efficient to implement than the more accurate version. Thus, while set constraints are expressive enough to encode dynamic typing, one apparently cannot derive the most efficient algorithms known for dynamic typing directly from this encoding.

6.2 Coercions at Arbitrary Points

So far we have considered only coercions at value creation and use points. Allowing coercions at arbitrary program points can sometimes result in better completions. To permit coercions to appear anywhere, the inference system must be altered to allow any of the four proper coercions to be applied to any expression. That is, the possible completions of each subexpression e are expressed by

$$C_{\text{FUNC!}}\ (C_{\text{BOOL!}}\ (C_{\text{FUNC?}}\ (C_{\text{BOOL?}}\ e)))$$

where C_x is potentially either the coercion named x or NOOP. For dynamic typing, where all components of tagged values are tagged, it is possible to modify the inference rules and the constraint resolution algorithm to handle coercions at arbitrary points.

6.3 Polymorphism

The semantics of polymorphic types based on set constraints has been developed in [AWL94]. A polymorphic type has the form $\forall \gamma_1, \ldots, \gamma_n.(\sigma \text{ where } S)$. Intuitively, this type expresses bounded quantification, with the set of constraints S acting as bounds on the quantified variables. More formally, the meaning is the intersection of all types $s(\sigma)$ where s is a solution of the constraints S for some choice of $\gamma_1, \ldots, \gamma_n$.

Polymorphism in the style of [AWL94] can be added to our system without modifying any other aspect. When tag variables are quantified, the meaning of coercions is parameterized in the type. In other words, types with quantified tag variables denote functions polymorphic in their coercions.

6.4 Adding Control-Flow Information

The simple idea of modelling a type as a pair consisting of a value part and a tag part leads to a system where tag inference is largely orthogonal to the inference of the structural part of the type. Thus, the same technique should integrate easily into other systems for analyzing dynamically typed programs. The system in [AWL94] is probably the most expressive and accurate such inference system known. Besides polymorphism, the most significant difference between [AWL94] and the system we have described is that types in [AWL94] can express control-flow through runtime tests. That is, given a conditional `if` $e\ e'\ e''$, the types of e' and e'' are constrained to reflect the values for which e is true and false respectively.

We can report that it is in fact straightforward to adapt the techniques reported in this paper to the system of [AWL94], yielding a system that can both remove as many dynamic type checks as [AWL94] and as many runtime tags as dynamic typing. We omit all details for lack of space. In this extended system, however, the system no longer has minimal completions and constraint resolution requires exponential time in the worst case.

7 Conclusions and Related Work

This work is part of a longer-term effort to investigate the principles underlying constraint-based program analyses. We believe that set constraints are a particularly useful formalism for expressing program analyses, but our interest was first aroused because it appeared that dynamic typing could not be expressed using set constraints or any other discipline using inclusion-based subtyping.

We have shown, however, that set constraints can encode dynamic typing, and in fact a substantial generalization of dynamic typing is naturally expressed using set constraints. Our system also has an efficient inference procedure. The flexibility and generality of set constraints allows our system to be extended in a variety of ways outlined in Section 6.

Based on our previous experience with constraint-based program analysis, we believe the algorithm we have presented could serve as the core of a practical analysis system for dynamically typed programs. However, the prime candidates for this kind of analysis are programs written in Lisp and Scheme. Analyzing such programs requires proper handling of side effects, an issue we have not considered.

Besides previous work on program analysis using set constraints, Henglein's work on dynamic typing is the most closely related to our own. Henglein's work is based, in turn, on earlier works of Thatte and Gomard [Gom90] . Thatte originally worked with a system called *partial types* [Tha88], in which types could be coerced to a universal type, but not vice versa—a pure subtyping system. Coercions from type `Dynamic` were introduced in a subsequent paper [Tha90].

A large number of analysis algorithms for dynamically typed languages have been proposed in recent years [Gom90, AM91, CF91, Hen92b, WH92, WC94]. With the exception of the works of Henglein, Thatte, and Gomard, it is fair to characterize all of these as (inclusion-based) subtyping systems; none treat tag inference. In this paper, we have shown how to combine expressive subtyping with the ability to infer minimal completions of tagging and untagging operations.

8 Acknowledgements

Fritz Henglein's insights on constraint-based program analysis in general and dynamic typing in particular were most helpful.

References

[Aik94] A. Aiken. Set constraints: Results, applications, and future directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, pages 171–179, Orcas Island, Washingtion, May 1994. Springer-Verlag LNCS no. 874.

[AM91] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, January 1991.

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[Gom90] C. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 282–287, 1990.

[Hei92] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

[Hen92a] F. Henglein. Dynamic typing. In *Proceedings of the Eurpean Symposium on Programming*, February 1992.

[Hen92b] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, July 1992.

[HJ90] N. Heintze and J. Jaffar. A decision procedure for a class of Herbrand set constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[MPS84] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21, 1985.

[Tha88] S. Thatte. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium*, pages 615–629. Springer-Verlag Lecture Notes in Computer Science, vol. 317, July 1988.

[Tha90] S. Thatte. Quasi-static typing. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 367–381, January 1990.

[WC94] A. Wright and R. Cartwright. A practical soft typing system for Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 250–262, June 1994.

[WH92] E. Wang and P. N. Hilfinger. Analysis of recursive types in Lisp-like languages. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 216–225, June 1992.