Quasi-static Typing

(Preliminary Report)

Satish R. Thatte*
Department of Mathematics and Computer Science
Clarkson University, Potsdam, NY 13676.

Abstract

We present a new approach to dynamic typing in a static framework. Our main innovation is the use of structural subtyping for dynamic types based on the idea that possible dynamic typing as a property should be *inherited* by objects of all types. Two properties of our system set it apart from existing systems which combine static and dynamic typing: all tagging and checking takes place via implicit coercions, and the semantics of dynamic typing is representation independent. The latter property leads to a significant increase in expressive power—for instance it allows us to define a *general* call-by-value fixpoint operator.

The resulting system—which we call quasi-static typing—is a seamless merger of static and dynamic typing. The system divides programs into three categories: well-typed, ill-typed and ambivalent programs. Ill-typed programs contain expressions that are *guaranteed* to go wrong. Run-time checking is limited to doubtful function applications in ambivalent programs. Conceptually, quasi-static typing takes place in an unusual two-phase process—a first phase infers types and coercions and a second *plausibility checking* phase identifies ill-typed programs. The typing rules allow minimal typing judgements and plausibility checking can be characterized as simplification via a canonical set of rewrite rules. The two phase process can therefore be implemented with a one pass algorithm.

*This work was started while the author was at the University of California, San Diego.

1 Introduction

In programming language design, the choice between static and dynamic typing is often seen as a fundamental choice between two opposing ideologies. In reality, there are many situations where it is desirable to combine the two approaches. For instance, it has been shown convincingly [ABC+83, ACPP89] that efficient and type secure use of persistent data—such as database files or just data exchanged beween different programs—requires type information for such objects to be saved and checked at run-time. Languages which subscribe to this idea (PS-Algol [ABC+83], Amber [Car86]) often allow any data object to be persistent, including those which involve complex pointer structures such as sharing and circularities. If such a language is otherwise statically typed (like Amber), it needs to provide dynamic typing as an option in an orthogonal way. An orthogonal combination of static and dynamic typing has other interesting applications. For instance, during program development, it is sometimes more convenient to use LISP-like heterogeneous data structures—which require dynamic typing—instead of "homogeneous" structures of variant types because the exact shapes of some

variant types may not be clear at the early stages of design.

Several practical statically typed languages (CLU, Mesa, Amber, Modula-3) address this issue by making special provisions for dynamic typing. In a recent paper, Abadi, Cardelli, Pierce and Plotkin [ACPP89] survey the history of these ideas and describe an elegant general combination of static and dynamic typing based on existing features in Amber and Modula-3 [CDJ+89]. The system they describe disguises dynamically typed objects with a special "static" type Dynamic, and provides special operations to the user for explicit attachment and checking of run-time type-tags. The main shortcoming of this system is that the semantics of dynamic typing is closely tied to the underlying representation—an object in effect "remembers" the history of tagging steps through which it has passed and a successful run-time check must look for the exact tagging pattern reflecting this history. This can have counterintuitive semantic consequences, as we argue in Section 2. Moreover, when programmers must explicitly manage the static-dynamic interface, the need to keep track of tagging patterns represents a considerable burden if dynamic typing is used extensively.

We describe a new approach to dynamic typing in static languages that gives representation-independent semantics to dynamically typed objects while at the same time eliminating the need for explicit management of the static-dynamic interface. We use the general framework of [ACPP89] as our starting point. The key new idea in our approach is to treat dynamic typing as a property that is *inherited* by all types from the type Dynamic through a natural subtype structure. The subtype structure is given a *coercive* interpretation. Positive coercions (from static to dynamic types) are tagging operations. We also need *negative* coercions (from supertypes to subtypes, *i.e.*, from dynamic to static types) to account for run-time checks. The idea of automatic negative coercions is new, as far as we know. Much of the theory developed here is in fact independent of the specific subtype structure induced by Dynamic and could be presented much more abstractly as a theory of type inference involving positive and negative coercions for a class of *injective* subtype structures.

Using the ideas outlined above, we achieve (for the user) a seamless merger of static and dynamic typing which we call *quasi-static typing.* Its practical characteristics are qualitatively different from those of purely static and purely dynamic systems. Static systems traditionally attempt to prove that a program is (or can be) well-typed, rejecting those for which no such proof is possible within the system. In most static systems there is no way to prove that a program is *ill*-typed in the sense that it contains an expression that necessarily goes wrong at run-time. Given the undecidability of "semantically complete" typechecking for most languages, a system that permits ill-typing proofs has to make a three way division of programs into well-typed, ill-typed, and ambivalent ones. Such a system can combine some of the advantages of static and dynamic typing by using run-time checking only in ambivalent programs, or better yet, only at the ambivalent *applications* in ambivalent programs. Quasi-static typing is a system of this kind. The version in this paper is limited to simply typed dialects of the λ-calculus, but it should be possible to construct more general systems based on the same principles along the lines of other systems which combine subtyping with parametric polymorphism [CW85, BTCGS89]. We have elsewhere [Tha88] described constraint resolution algorithms that may be useful for improving the degree of type inference in such a generalized system.

Conceptually, quasi-static typing is a two phase process. The first phase makes typing (and coercion) judgements. A program that would normally be considered statically well-typed is accepted by this phase without any *negative* coercions and *all* other programs are accepted *with* some negative coercions. One way to understand the reason for this excessive lenience is to think of quasi-static typing as a system which matches the liberality of dynamic typing while restricting tagging and checking operations to those that are actually found

to be required based on compile-time analysis. The typing/coercion judgements express the results of this analysis. The information gained through the analysis makes it possible to identify many ill-typed programs at compile-time. This is accomplished by a novel *plausibility checking* phase that preevaluates the coercions introduced by typing. Strictly speaking, the programs rejected during plausibility checking are not *statically* ill-typed—they contain dynamic type errors that can be statically detected. It is easy to modify quasi-static typing to reintroduce the possibility of static type errors in the formal sense. This is actually highly desirable for methodological reasons. Some ideas for such modifications are discussed in Section 8.

Our typing rules allow the derivation of minimal typing/coercion judgements (see below) and plausibility checking can be characterized as a simplification process specified by a canonical (confluent terminating) set of rewrite rules. This permits a complete one pass implementation of the two phases described above.

An interesting theoretical point to note in connection with the notion of minimal judgements is that certain design decisions (discussed in Section 3) lead us to abandon coherence of typing judgements—the property that the meaning of an expression depends only on its typing judgement, not on the specific proof for that judgement (see [BTCGS89] for a discussion). The possibility of dependence on the proof is created by the fact that the coercions introduced during typing depend on the proof rather than on the typing judgement. Such a dependence on proof is clearly unacceptable. Among other things, it calls into question the notion of a minimal typing/coercion judgement, since the coercion part may not be "minimal" in any reasonable sense if it is nondeterministically related to the typing part. Fortunately, our type system does possess a weaker property—which we call *convergence*—which ensures that there is a "minimal" proof for each typing judgement which gives the "least error-prone" (and therefore canonical) semantics. The notion of a minimal typing/coercion judgement in our system is therefore defined in two steps. A minimal judgement has a minimal typing part, and a coercion part corresponding to a minimal proof for the typing part. These ideas are made precise in Sections 3 and 7.

In expressive power, our system is not directly comparable to that of [ACPP89]. Besides automation of tagging and checking, the semantics of dynamic typing in our system is significantly more abstract. For instance, a general call-by-value fixpoint operator can be expressed in our framework. The analogous construction (with explicit tagging and checking) in the language of [ACPP89] fails due to their representation oriented interpretation of dynamic typing. Our system is *less* general in one respect. The **typecase** construct of [ACPP89] (which can coexist with quasi-static typing) uses multiple patterns and has conditional branching on both success and failure of dynamic checks. It is hard to see how an *implicit* check can produce anything other than a run-time error on failure. Examples like the general purpose **print** function of [ACPP89] are therefore beyond our system.

The next section motivates and describes our interpretation of dynamic typing. Typing and plausibility checking rules are described in Sections 3 and 4. A complete type inference algorithm is given in Section 5. Operational and denotational semantics is described in Sections 6 and 7. We conclude with a discussion of some pragmatic issues and concluding remarks in Sections 8 and 9.

2 Partial Types and Cumulative Coercions

In this section we motivate and explain our interpretation of dynamic typing and the corresponding coercion scheme. General familiarity with existing work in the area is assumed (see [ACPP89] for a brief survey and formal treatment). For convenience of explanation, we use simple objects such as integers and booleans in examples, but it should be clear that we could have used complex objects like bitmaps and relations instead.

The type structure used in this section includes ba-

sic types Nat and Bool, and the usual type constructors for product ($\times$), list([ ]), and function($\rightarrow$) types. The formal part of the paper leaves out list types for brevity. The static type of dynamically typed objects is denoted by Ω instead of Dynamic for reasons to be explained shortly. The subtype order is based on the scheme $\tau \leq \Omega$ for all types τ, which expresses the basic inheritance relationship we wish to capture. This extends monotonically over the type constructors except for the usual antimonotonic behavior of the first argument of "$\rightarrow$". For instance we have $[\mathsf{Nat}] \leq [\Omega]$ and $[\Omega] \rightarrow \mathsf{Nat} \leq [\mathsf{Nat}] \rightarrow \Omega$. "More" in this order is just "more dynamic". The order relates all (and only) pairs $\tau_1 \leq \tau_2$ such that an object x of type τ_1 can be converted to one of type τ_2 by applying or composing tagging operations with either the whole or parts of x.

Programs are written in a "user language" which is just Church's typed λ-calculus. There are no tagging or checking constructs such as, for instance, Amber's dynamic and coerce. Analogous but somewhat different constructs are used by the type system as *implicit* coercions. One reason why Amber-like constructs cannot be used is that their semantics is too closely tied to the underlying representation—different tagging patterns lead to semantically different objects even when the underlying untagged objects and the types of their tagged versions are identical. Using these constructs as coercions (with their current semantics) would result in loss of coherence for subsumption judgements. Given $\tau_1 \leq \tau_2 \leq \tau_3$, an object x of type τ_1 would be converted to semantically distinct objects of type τ_3 depending on whether or not an intermediate conversion to type τ_2 occurs. It would be impossible to construct unique coercions for every subtype relationship as one usually expects. A very simple example illustrates the problems and suggests a solution. Suppose l is a list of type [Nat];

$$\mathsf{emb}_{\mathsf{Nat}} = \lambda x.\, \mathsf{dynamic}\ x : \mathsf{Nat}$$
$$l_0 = \mathsf{map}\ \mathsf{emb}_{\mathsf{Nat}}\ l$$
$$l_1 = \mathsf{dynamic}\ l_0 : [\Omega]$$
$$l_2 = \mathsf{dynamic}\ l : [\mathsf{Nat}]$$

Recall that the result of the expression "dynamic $x : \tau$" is an object of type Ω which is just the dynamically typed version of x (assuming that x has type τ). Clearly, l_0 converts l to an object of type $[\Omega]$, and l_1, l_2 both define tagged versions of l of type Ω. The only difference between l_1 and l_2 is that the conversion in l_1 takes place in two steps rather than one. Nonetheless, l_1 and l_2 are *observably* different objects in the system of [ACPP89]. This is not just a theoretical difficulty. The semantic distinction may cause spurious run-time errors. For instance, coerce l_1 to [Nat] (to use the Amber notation for run-time checks) fails. The more direct tagging in l_2 causes coerce l_2 to $[\Omega]$ to fail. In each case, substituting the "other" version leads to success. This is not a fatal problem when tagging is explicit because the programmer is then responsible for knowing the tagging pattern and checking it appropriately, although it does represent a greater burden than is generally recognized. When both tagging and checking are implicit, this kind of uncertainty would make the system unusable.

We conclude that the recasting of dynamic typing as an inherited property forces a change in its interpretation. In place of a representation oriented semantics we need a more abstract "information" oriented semantics in which types involving Dynamic are viewed as *partial* static types, in analogy with Wadsworth's partial terms [Wad76]. Tagging and checking clearly obey a "law of conservation of type information" which decrees that a loss of static type information is accompanied by an identical gain in dynamic type information and *vice versa*. The transitivity of our subtype order for dynamic types implies that gains and losses in type information are *cumulative*. The increments in which they occur should not affect the end result. Hence the expectation that l_1 and l_2 should behave in an identical way. To emphasize this shift in perspective, we use Wadsworth's Ω symbol—which represents lack of information—in place of Dynamic. To make tagging and checking free from representation bias, we take it as a principle that a run-time check should succeed whenever it is possible to convert the object of the check (by untagging) to a value

that belongs to a *subtype* of the required type. This may require the check to be propagated to components of a structure. Under this regime, coerce l_1 to [Nat] succeeds, and l_1 and l_2 are *observably equivalent* at run-time.

It is possible to achieve these semantic changes by simply changing the interpretation of coerce appropriately. Although this would eliminate the semantic problem in using dynamic and coerce as implicit coercions, *static* detection of type errors in the resulting system would be all but impossible. As we describe in Section 4, statically detected errors in our system correspond to cases where tagging and checking operations are combined in an *implausible* way, *i.e.*, in a way that ensures a dynamic type error. For instance, the expression 1+true is coerced by the typing phase to (the equivalent of) 1+coerce (dynamic true:Bool) to Nat, which is rejected as implausible at compile-time. The effectiveness of this technique depends on statically manifest tagging at the outermost level possible. For instance, coerce l_0 to [Bool] is *not* statically implausible, even though the type of the underlying value l is incompatible with [Bool]. There is no way to coerce l to the type $[\Omega]$ with direct (outermost) tagging using dynamic. The difficulty is caused by the fact that dynamic and coerce can only do limited conversions—those for subsumptions of the form $\tau \leq \Omega$. The obvious solution is to generalize these operations to arbitrary subsumptions. We use a family $\uparrow_\tau^\sigma$, $\downarrow_\tau^\sigma$ of postfix operators (one for each pair $\tau \leq \sigma$) for generalized tagging and checking; $e \uparrow_\tau^\sigma$ means e belongs to type τ and is being coerced to a more dynamic type σ by tagging, and $e \downarrow_\tau^\sigma$ means e belongs to type σ and is being coerced to a less dynamic type τ by checking. Thus $\mathsf{emb}_{\mathsf{Nat}}$ is redefined as $\lambda x : \mathsf{Nat}.\, x \uparrow_{\mathsf{Nat}}^{\Omega}$ and $l_3 = l \uparrow_{[\mathsf{Nat}]}^{[\Omega]}$ is equivalent to $l_4 = \mathsf{map}\ \mathsf{emb}_{\mathsf{Nat}}\ l$ at run-time, but *not* at compile-time. For instance, given a function f : [Bool] → Bool, the application $f\ l_3$ is coerced by the typing phase to $f(l \uparrow_{[\mathsf{Nat}]}^{[\Omega]} \downarrow_{[\mathsf{Bool}]}^{[\Omega]})$ but $l \uparrow_{[\mathsf{Nat}]}^{[\Omega]} \downarrow_{[\mathsf{Bool}]}^{[\Omega]}$ is rejected as ill-typed by the plausibility checking phase at compile-time. The application $f\ l_4$ is coerced to $f\ (l_4 \downarrow_{[\mathsf{Bool}]}^{[\Omega]})$ which is considered plausible and leads to a type error at run-time.

Heterogeneous data structures can be typed using partial types. If $[x_1, \ldots, x_n]$ denotes the list containing items $x_1, \ldots, x_n$, then [1, 2, 3, true] may be coerced to the list $z = [1\uparrow_{\mathsf{Nat}}^{\Omega}, 2\uparrow_{\mathsf{Nat}}^{\Omega}, 3\uparrow_{\mathsf{Nat}}^{\Omega}, \mathsf{true}\uparrow_{\mathsf{Bool}}^{\Omega}]$ of type $[\Omega]$. Given a function *sum* : [Nat] → Nat, the application *sum z* is coerced to the plausible form *sum* $(z \downarrow_{[\mathsf{Nat}]}^{[\Omega]})$. The dynamic check implied by the coercion is carried out when *sum* is called and the error is caught at that time, rather than being delayed until true is actually encountered during the execution of *sum*. In general, dynamic type errors in this scheme are caught much earlier and at more "logical" points than in LISP-like pure dynamic typing under which only illegal applications of primitive functions are caught and it is often hard to find the *programming* error that led to the type error. A technique for improving the *efficiency* of generalized checks—such as $z \downarrow_{[\mathsf{Nat}]}^{[\Omega]}$ in the *sum* application above—is discussed in Section 8.

In the next section, we give additional examples including a general call-by-value fixpoint operator.

3 Typing Rules

Throughout the formal part of the paper, our notation largely follows [ACPP89]. The set of all type expressions (described in Section 2) will be denoted by *Typecode*. Letting the metavariable e range over expressions in the user language, x over identifiers and τ over type expressions, we have:

$e ::=$	x	*identifiers*
	$\mid \lambda x : \tau.\, e_{body}$	*typed abstractions*
	$\mid e_{fun}\ e_{arg}$	*applications*
	$\mid e_{left},\ e_{right}$	*pairs*

The internal language of the system also has the coercion operators $\uparrow_\tau^\sigma$ and $\downarrow_\tau^\sigma$ as described in Section 2. For brevity, we assume that the environment contains a family of primitives for elimination of tupling. In practice, one would need additional constructs for this, for conditional expressions, and so forth.

One rather unusual aspect of our inference rules for typing judgements is that we need to insert run-time

checks which are *negative* coercions (from a supertype to a subtype). The insertion of positive coercions is often left implicit as a side-effect of a subtyping rule. A similar implicit rule for the insertion of negative coercions would be equivalent to adding the relationship $\Omega \leq \tau$ which would collapse the subtype order and destroy both semantic consistency and minimal typing. We therefore make the insertion of coercions explicit in typing judgements. The general form of a judgement is "$TE \vdash e \Rightarrow e_{new} : \tau$" which may be read as: "given the set TE of typing assumptions for free variables, the (user language) expression e can be coerced to the (internal language) expression e_{new} which has the type τ". The first four rules are standard:

$$\frac{x \in Dom(TE)}{TE \vdash x \Rightarrow x : TE(x)}$$

$$\frac{TE \vdash e \Rightarrow e_{new} : \tau \quad \tau \leq \sigma}{TE \vdash e \Rightarrow (e_{new} \uparrow_{\tau}^{\sigma}) : \sigma}$$

$$\frac{TE[x \leftarrow \tau] \vdash e_{body} \Rightarrow e_{newbody} : \sigma}{TE \vdash \lambda x : \tau.\ e_{body} \Rightarrow \lambda x : \tau.\ e_{newbody} : \tau \rightarrow \sigma}$$

$$\frac{TE \vdash e_{left} \Rightarrow e_{newleft} : \sigma \quad TE \vdash e_{right} \Rightarrow e_{newright} : \tau}{TE \vdash (e_{left},\ e_{right}) \Rightarrow (e_{newleft},\ e_{newright}) : \sigma \times \tau}$$

The insertion of run-time checks takes place under two circumstances, both in the context of an application. The first case is the more common one where the type of the argument demanded by a function is a subtype of the type of the actual argument.

$$\frac{TE \vdash e_{fun} \Rightarrow e_{newfun} : \sigma \rightarrow \mu \quad TE \vdash e_{arg} \Rightarrow e_{newarg} : \tau \quad \tau \geq \sigma}{TE \vdash e_{fun}\ e_{arg} \Rightarrow e_{newfun}\ (e_{newarg} \downarrow_{\sigma}^{\tau}) : \mu}$$

The second case for applications occurs when the function part is of type Ω. This roughly resembles the situation in ML-style typechecking where the type of the function part is a type variable. The solution is roughly similar, except that a check instead of a unification is invoked:

$$\frac{TE \vdash e_{fun} \Rightarrow e_{newfun} : \Omega \quad TE \vdash e_{arg} \Rightarrow e_{newarg} : \tau}{TE \vdash e_{fun}\ e_{arg} \Rightarrow (e_{newfun} \downarrow_{\tau \rightarrow \Omega}^{\Omega})\ e_{newarg} : \Omega}$$

The form of the rules for application together with the semantics of run-time checks for function values is the central design issue in quasi-static typing. The engineering problem is a tradeoff between expressive power and static detection of errors, while attempting to maintain certain desirable "invariants" such as coherence. Expressive power is enhanced by defering checks to run-time. For instance, the semantics of checking for function values defined in Sections 6 and 7 is quite lenient: an implied check on the argument of a function is defered to run-time. Tightening the semantics of checking for functions by treating implied checks on arguments as errors would improve static detection of errors at the cost of causing the *apply* and *fix* operators defined later in this section to fail. The other choice is in the typing rules: when the actual type of an argument in an application is a supertype of the required type, the required check could be applied to either the function or the argument part. With a strict semantics of checking for functions, this choice would be immaterial, but given our lenient semantics it is significant. Applying the check to the function part yields the rule

$$\frac{TE \vdash e_{fun} \Rightarrow e_{newfun} : \sigma \rightarrow \mu \quad TE \vdash e_{arg} \Rightarrow e_{newarg} : \tau \quad \tau \geq \sigma}{TE \vdash e_{fun}\ e_{arg} \Rightarrow (e_{newfun} \downarrow_{\tau \rightarrow \mu}^{\sigma \rightarrow \mu})\ e_{newarg} : \mu}$$

Using this rule would make the system "semantically complete" in the sense that no program would be rejected unless its *untyped version* is guaranteed to lead to type error at run-time. However, this seriously damages the ability of the system to detect type errors statically because the lenient semantics of checking for function values makes all checks on them plausible. For this reason, the first rule above applies the check to the argument part. But this is not consistent with coherence—the property that the same typing judgement for an expression, arrived at by different proofs, should cor-

respond semantically to the same coerced value. The problem can be seen with a simple example. The function part of the application $(\lambda x : \Omega.\, x)\ 3$ can be coerced using the subtyping rule to $(\lambda x : \Omega.\, x) \uparrow^{\mathsf{Bool}\to\Omega}_{\Omega\to\Omega}$. The argument 3 must then be coerced to $3 \uparrow^{\Omega}_{\mathsf{Nat}} \downarrow^{\Omega}_{\mathsf{Bool}}$ to satisfy the rule above. The coerced expression is implausible; it is guaranteed to lead to error at run-time. However, the original application $(\lambda x : \Omega.\, x)\ 3$ can also be coerced by subtyping to $(\lambda x : \Omega.\, x)\ (3 \uparrow^{\Omega}_{\mathsf{Nat}})$ which evaluates without error.

To summarize the design issue, there are three alternatives with increasing expressive power and decreasing detection of static errors:

1. A strict semantics for checking of function values.
2. A lenient semantics with checks caused by applications applied to arguments.
3. A lenient semantics with checks caused by applications applied to functions.

Alternatives (1) and (3) are coherent while (2) is not. However, (2) is not ruled out because the system it implies does possess a "minimal judgement" property (Theorem 7 below) which not only guarantees minimal typing but also a "least error-prone" conversion of the original expression. In other words, the lack of coherence only permits introduction of spurious dynamic errors. We use alternative (2) in this paper, partly to demonstrate its viability and also because it provides an attractive balance of expressive power and static error detection. For many languages, applications such as *apply* and *fix* (see below) would be irrelevant, and alternative (1) might be the preferred choice.

The typing rules possess soundness and completeness properties. In the following, the notation $[\![e]\!]\rho$ denotes the denotation of expression e in environment ρ, and $[\![TE]\!]$ denotes a set of possible environments corresponding to the set TE of typing assumptions for free variables The denotational semantics of types and expressions is discussed in Section 7. In stating the semantic soundness of the typing rules, Theorem 1 below explicates the semantic role of typing judgements: the intended semantics of an expression in the user language is the just the ordinary semantics of its coerced version.

Theorem 1 (Soundness of Typing Rules) $TE \vdash e \Rightarrow e' : \tau$ *implies* $\forall \rho \in [\![TE]\!].\ [\![e']\!]\rho \in [\![\tau]\!]$.

In the theorem below, $\mathit{ClosedExp}(TE)$ denotes the set of expressions that are closed relative to TE, *i.e.*, those whose free variables have typing assumptions in TE.

Theorem 2 (Completeness of Typing Rules) $\forall TE.\, \forall e \in \mathit{ClosedExp}(TE).\, \exists e', \tau.\ TE \vdash e \Rightarrow e' : \tau$.

Theorem 2 implies that *all* expressions are statically well-typed in the formal sense. The purpose of the typing process is to insert enough tagging and checking operations to make the expression *prima facie* meaningful. A judgement "$TE \vdash e \Rightarrow e' : \tau$" is a well-typing judgement in the usual sense only if e' does not contain any run-time checks. The detection of ill-typed expressions takes place in the plausibility checking process described in Section 4.

As an example of the use of these typing rules, consider an example from [ACPP89]—a function that applies its first argument to its second argument. The coerced form is shown following a "$\Rightarrow$".

$$\begin{aligned} \mathit{apply} &= \lambda f : \Omega.\, \lambda x : \Omega.\, f\ x \\ &\Rightarrow \lambda f : \Omega.\, \lambda x : \Omega.\, (f \downarrow^{\Omega}_{\Omega\to\Omega})\ x \end{aligned}$$

Using the evaluation rules of Section 6, it is easy to show that the application $\mathit{apply}\ ((\lambda y{:}\mathsf{Nat}.\, y) \uparrow^{\Omega}_{\mathsf{Nat}\to\mathsf{Nat}})$, for instance, is equivalent to $\lambda x : \Omega.\, (x \downarrow^{\Omega}_{\mathsf{Nat}} \uparrow^{\Omega}_{\mathsf{Nat}})$. The details of a related evaluation are described in Section 6.

A nice demonstration of the expressive power of the system is given by the next example, which shows that in contrast to the system of [ACPP89]—which can only allow *specialized* fixpoint operators—we can express a general call-by-value fixpoint operator in a very straightforward way:

$$\begin{aligned} \mathit{fix} &= \lambda f : \Omega \to \Omega.\, d\ d, \quad \mathit{where} \\ &\quad d = \lambda x : \Omega.\, \lambda z : \Omega.\, (f\ (x\ x))\ z \\ &\Rightarrow \lambda f : \Omega \to \Omega.\, d\ (d \uparrow^{\Omega}_{\Omega\to(\Omega\to\Omega)}), \quad \mathit{where} \\ &\quad d = \lambda x : \Omega.\, \lambda z : \Omega.\, ((f\ ((x \downarrow^{\Omega}_{\Omega\to\Omega})\ x)) \downarrow^{\Omega}_{\Omega\to\Omega})\ z \end{aligned}$$

The type of *fix* is $(\Omega \to \Omega) \to (\Omega \to \Omega)$ which seems unusual but, with the semantics for coercions given in Sections 6 and 7, works properly. The details are left as an exercise for the reader. It is interesting to note that the coercions in this example are similar to those needed to provide a coercion-based semantics for the untyped λ-calculus [BTCGS89, pages 113-114 (with D=Ω)].

4 Plausibility Checking

Plausibility checking is a kind of simplification. The process is most naturally specified by a canonical—confluent terminating—set of conditional rewrite rules meant to be used at compile-time. In the rules below, we use the unusual arrow "$\rightsquigarrow$" for the rewrite relation to avoid confusion with the constructor for function types. The notation $\tau \sqcap \sigma$ denotes the GLB of the type expressions τ and σ. Algorithms for GLB and LUB are self-evident. When a rule is conditional, the condition is given above a line like a hypothesis in an inference rule. The symbol *wrong* represents dynamic type-error. It is more formally described in the denotational semantics given in Section 7. Since the tagging and checking operators are only introduced internally by the typing phase and are not available to users, we assume that their occurrences are always "well-formed". An expression such as $\mathsf{true}\downarrow^{\mathsf{Nat}}_{\mathsf{Bool}}$ will never be encountered since the typing rules—by the soundness property—never produce an expression of the form $e\downarrow^{\sigma}_{\tau}$ unless $\sigma \geq \tau$ and $[\![e]\!]\rho \in [\![\sigma]\!]$ for an appropriate environment ρ. Similar remarks apply to tagging.

$$e\downarrow^{\tau}_{\tau} \rightsquigarrow e \qquad\qquad e\uparrow^{\tau}_{\tau} \rightsquigarrow e$$

$$e\downarrow^{\tau}_{\sigma}\downarrow^{\sigma}_{\mu} \rightsquigarrow e\downarrow^{\tau}_{\mu} \qquad\qquad e\uparrow^{\sigma}_{\mu}\uparrow^{\tau}_{\sigma} \rightsquigarrow e\uparrow^{\tau}_{\mu}$$

$$\frac{\mu = \tau \sqcap \nu}{e\uparrow^{\sigma}_{\tau}\downarrow^{\sigma}_{\nu} \rightsquigarrow e\downarrow^{\tau}_{\mu}\uparrow^{\nu}_{\mu}} \qquad\qquad \frac{\nexists\mu.\ \mu = \tau \sqcap \nu}{e\uparrow^{\sigma}_{\tau}\downarrow^{\sigma}_{\nu} \rightsquigarrow \mathit{wrong}}$$

The first four of these rules are simple and require no explanation. The last two (conditional) rules embody the essence of plausibility checking. The idea is that a run-time check is plausible only if the known type of the expression being checked and the type required by the check have a *common subtype*. The possibility that this is not so arises when the best (least) known type is not a supertype of the required type. In this case, the typing phase creates a tagging operation followed by a checking operation—this is the pattern addressed by the last two rules. The rules state that, for instance, the check in $e\uparrow^{\Omega\times\Omega}_{\mathsf{Nat}\times\Omega}\downarrow^{\Omega\times\Omega}_{\Omega\times\mathsf{Bool}}$ can only be successful if the untagged value of e belongs to Nat × Bool. The first rule therefore changes the expression to $e\downarrow^{\mathsf{Nat}\times\Omega}_{\mathsf{Nat}\times\mathsf{Bool}}\uparrow^{\Omega\times\mathsf{Bool}}_{\mathsf{Nat}\times\mathsf{Bool}}$. Such a simplification is not possible for $e\uparrow^{\Omega\times\Omega}_{\mathsf{Bool}\times\Omega}\downarrow^{\Omega\times\Omega}_{\mathsf{Nat}\times\Omega}$—a type conflict is guaranteed in the first component of the product. The situation is very much like a failure in unifying the known and required type of an expression in ML-like type inference (with a new variable substituted for each occurrence of Ω). The only difference occurs for function types: the types Nat → Nat and Bool → Nat have the common subtype $\Omega \to$ Nat, but they are not unifiable. The semantic justification for these ideas is expressed Theorem 3.

Theorem 3 (Characterization of Implausibility) *If $\nexists\mu.\ \mu = \tau \sqcap \sigma$ then for any well-formed expression $e\uparrow^{\sigma}_{\tau}\downarrow^{\sigma}_{\nu}$ and any environment ρ, $[\![e\uparrow^{\sigma}_{\tau}\downarrow^{\sigma}_{\nu}]\!]\rho = \mathit{wrong}$.*

Given Theorem 3, it is easy to show that plausibility checking is *meaning preserving*. In the following, $ENV(e)$ is the set of all environments which define the free variables of e.

Theorem 4 (Soundness of Plausibility Checking) $\vdash e \rhd e'$ *implies* $\forall\rho \in ENV(e).\ [\![e]\!]\rho = [\![e']\!]\rho$.

It is easy to see how the six rewrite rules for plausibility checking can be incorporated in an algorithm. However, such an algorithm is forced to make some arbitrary choices since these rules are *nondeterministic* in the sense that more than one rule may be applicable to a given expression. It is therefore reassuring to note that the set of rules as a whole has pleasant syntactic proper-

ties that ensure the deterministic nature of plausibility checking:

Theorem 5 (Canonicity of Plausibility Checking) *If $\rightsquigarrow$ is taken to denote an abstract rewrite relation in the sense of [Hue80], then the relation defined by the six rules above is confluent and terminating,* i.e., *canonical, and therefore produces* unique *normal forms.*

Any algorithm that implements these rewrite rules must therefore compute the same abstract function for deriving the normal forms guaranteed by Theorem 5. We use the symbol **Simplify** to denote this function. The notation "$\vdash\ e\ \triangleright\ e'$" is used to express the *simplification* judgement that "e is reduced by plausibility checking to the expression e'". Simplification judgements are derived via the single obvious rule:

$$\vdash\ e\ \triangleright\ \textbf{Simplify}(e)$$

5 The Algorithm Type

The type inference algorithm **Type** is given in Figure 1. The algorithm is straightforward and efficient. The only opaque spot is the call $\textbf{Simplify}(e_a \uparrow^{\Omega}_{\tau_a} \downarrow^{\Omega}_{\tau_{fa}})$ in the application case. However, since e_a is already in normal form, the action of this call is very similar—except for the argument parts of function types—to a unification of τ_a and τ_{fa}. Note that this call on **Simplify** may *fail* (produce *wrong*), in which case the application is implausible and the algorithm **Type** returns with failure as well. **Type** is faithful to the typing rules in that when successful it returns a valid typing (and coercion) judgement.

Theorem 6 (Correctness of Type) *If the call* $\textbf{Type}(TE, e)$ *succeeds and returns* (τ, e_1), *then for some* e_2, $TE \vdash e \Rightarrow e_2 : \tau$ *and* $\textbf{Simplify}(e_2) = e_1$.

It is very desirable that a typing algorithm have a *completeness* property in the sense that any judgement in the corresponding logic can be factored uniquely into the "minimal" judgement rendered by the algorithm and a simple additional inference step. In a pure typing logic, this translates to the property that the algorithm derives the *principle* type, which in our case is the *minimal* type. When the judgement also involves coercion, an additional *coherence* property of the logic—which asserts that the semantics of the coercion part of a judgement is a function of the typing part—ensures that the factoring via the minimal type works for the coercion part as well. As we explained in Section 3, we choose to abandon coherence in order to find a balance between expressive power and static detection of type-errors. Instead of coherence, we use a weaker but practically sufficient *convergence* property which preserves the notion of a canonical judgement discovered by **Type**. The basis of this property is a partial order relation "$\sqsupseteq$" on semantic values: $v_1 \sqsupseteq v_2$ means that either v_1 is ***wrong*** or v_1 is "more error-prone" than v_2. This is defined extensionally in roughly the same way as the approximation relation in Scott-domains. A precise definition is given in Section 7. Convergence asserts that corresponding to each typing judgement, there is a *minimal* proof which derives the *least error-prone* coerced version of the given expression. For lack of space, we leave the formal statement of convergence as simply one of the corollaries of the following theorem.

Theorem 7 (Completeness of Type) *If* $TE \vdash e \Rightarrow e_1 : \tau$ *and* $\textbf{Simplify}(e_1) \neq$ *wrong, then* $\textbf{Type}(TE, e)$ *succeeds and returns* (σ, e_2) *such that* $\tau \geq \sigma$ *and* $\forall \rho \in [\![TE]\!].\ [\![e_1]\!]\rho \sqsupseteq [\![e_2 \uparrow^{\tau}_{\sigma}]\!]\rho$.

Corollary 8 *For any* TE, *any* $e \in ClosedExp(TE)$ *and any type* τ, *if there are derivations of the form* $TE \vdash e \Rightarrow e_1 : \tau$, *then there is an expression* e_0 *such that* $TE \vdash e \Rightarrow e_0 : \tau$, *and for all* $\rho \in [\![TE]\!].\ [\![e_1]\!]\rho \sqsupseteq [\![e_0]\!]\rho$ *for all such* e_1.

Proof (sketch) The only cases not covered by Theorem 7 are those in which all e_1 are semantically equivalent to ***wrong***. The corollary in that case is trivial. ⊣

$$
\begin{array}{l}
\textbf{Type}(TE, e) \;=\; \textbf{case } e \textbf{ of} \\
\qquad\qquad x : \; TE(x), x \\
\qquad\qquad e_{left}, e_{right} : \; let \;\; \tau_l, e_l = \textbf{Type}(TE, e_{left}) \;\; and \;\; \tau_r, e_r = \textbf{Type}(TE, e_{right}) \\
\qquad\qquad\qquad\qquad in \;\; \tau_l \times \tau_r, \; (e_l, e_r) \\
\qquad\qquad \lambda x : \tau. e_{body} : let \; \tau_b, e_b = \textbf{Type}(TE[x \leftarrow \tau], e_{body}) \\
\qquad\qquad\qquad\qquad in \;\; \tau \rightarrow \tau_b, \; \lambda x : \tau. e_b \\
\qquad\qquad e_{fun} \;\; e_{arg} : let \;\; \tau_f, e_f = \textbf{Type}(TE, e_{fun}) \;\; and \;\; \tau_a, e_a = \textbf{Type}(TE, e_{arg}) \\
\qquad\qquad\qquad\qquad in \; if \;\; \tau_f = \tau_{fa} \rightarrow \tau_{fr} \;\; then \\
\qquad\qquad\qquad\qquad\qquad let \; e_{na} = \textbf{Simplify}(e_a \uparrow^{\Omega}_{\tau_a} \downarrow^{\Omega}_{\tau_{fa}}) \; in \;\; \tau_{fr}, \; (e_f \; e_{na}) \\
\qquad\qquad\qquad\qquad\quad else \; if \;\; \tau_f = \Omega \; then \;\; \Omega, \; (e_f \downarrow^{\Omega}_{\tau_a \rightarrow \Omega} e_a) \\
\qquad\qquad\qquad\qquad\quad else \; fail
\end{array}
$$

Figure 1: *Algorithm* **Type**

6 Operational Semantics

The operational semantics is given in the "natural semantics" style following [ACPP89]. The idea of evaluation is to reduce an expression to a unique normal form. The evaluation judgement "$\vdash e \Rightarrow v$" is read "the expression e reduces to the normal form v". Evaluation is only defined for closed expressions. The call-by-value interpretation is used for evaluation of applications. Nonterminating computations do not have a normal form, and therefore do not have an operational meaning. It is worth noting that the normal forms derived in the operational semantics do *not* always correspond directly to the semantic values used in the denotational semantics, although the two are related by the soundness property stated in Theorem 9. This point and its implications are discussed in Section 8.

Evaluation preserves *wrong*. The role of *wrong* is a bit tricky. This constant represents run-time error, and belongs to all types rather than none. As such, it can be derived as the result of a statically well-typed and plausible expression. It in effect plays two roles in the rules below—it may be the result of evaluating an absurd expression such as (3 true) which would never be produced by the typing phase, or it may be the result of a failed run-time check. It would be more appropriate to use two different constants for these purposes. One could then show that the former case never occurs in "well-typed" expressions. This separation is absent in the rules below.

The semantics of constants, λ-expressions, applications and pairs is similar to [ACPP89]:

$$\vdash \mathit{wrong} \Rightarrow \mathit{wrong}$$

$$\vdash \lambda x{:}\tau.\, e \;\Rightarrow\; \lambda x{:}\tau.\, e$$

$$\frac{\vdash e_{fun} \Rightarrow \lambda x{:}\tau.\, e_{body} \qquad \vdash e_{arg} \Rightarrow w \quad (w \neq \mathit{wrong}) \qquad \vdash e_{body}[x \leftarrow w] \Rightarrow v}{\vdash e_{fun} \;\; e_{arg} \Rightarrow v}$$

$$\frac{\vdash e_{fun} \Rightarrow w \quad (w \neq \lambda x{:}\tau.\, e_{body})}{\vdash e_{fun} \;\; e_{arg} \Rightarrow \mathit{wrong}}$$

$$\frac{\vdash e_{arg} \Rightarrow \mathit{wrong}}{\vdash e_{fun} \;\; e_{arg} \Rightarrow \mathit{wrong}}$$

$$\frac{\vdash e_{left} \Rightarrow l \quad (l \neq \mathit{wrong}) \qquad \vdash e_{right} \Rightarrow r \quad (r \neq \mathit{wrong})}{\vdash e_{left}, e_{right} \Rightarrow l, r}$$

$$\frac{\vdash e_{left} \Rightarrow \mathit{wrong}}{\vdash e_{left}, e_{right} \Rightarrow \mathit{wrong}}$$

$$\frac{\vdash e_{right} \Rightarrow \mathit{wrong}}{\vdash e_{left}, e_{right} \Rightarrow \mathit{wrong}}$$

The simplifications performed in plausibility checking are operationally sound:

$$\frac{\vdash e \rhd e' \quad \vdash e' \Rightarrow v}{\vdash e \Rightarrow v}$$

There are three cases for evaluating tagging operations. If the value being tagged in not *wrong*, and the *result* type in the tagging operation is not a function type, then the tagged version is in normal form. Note that tagging is not propagated to the components of pairs, although checking (see below) is. Cases where a check is applied to a tagged pair are handled by the simplification judgement used in the rule above.

$$\frac{\vdash e \rhd e' \uparrow^{\sigma}_{\tau} \quad (\sigma \neq \sigma_1 \to \sigma_2) \quad \vdash e' \Rightarrow w \quad (w \neq \mathit{wrong})}{\vdash e \Rightarrow w \uparrow^{\sigma}_{\tau}}$$

Tagging of function values is resolved by applying the implied tagging operations to the argument and result.

$$\frac{\vdash e \Rightarrow \lambda x{:}\tau_1.\, e_{body}}{\vdash e \uparrow^{\sigma_1\to\sigma_2}_{\tau_1\to\tau_2} \Rightarrow \lambda y : \sigma_1.\, (e_{body}[x \leftarrow (y \uparrow^{\tau_1}_{\sigma_1})]) \uparrow^{\sigma_2}_{\tau_2}}$$

The notation $e[x \leftarrow e']$ denotes the expression obtained by replacing the variable x in e by e' avoiding capture of free variables in the usual sense. Finally, tagging preserves *wrong*.

$$\frac{\vdash e \Rightarrow \mathit{wrong}}{\vdash e \uparrow^{\sigma}_{\tau} \Rightarrow \mathit{wrong}}$$

An expression with a *check* is obviously never in normal form. When the checked value is a pair, for instance, the check must be propagated to the components.

$$\frac{\vdash e \Rightarrow l, r \quad \vdash l \downarrow^{\tau_1}_{\sigma_1}, r \downarrow^{\tau_2}_{\sigma_2} \Rightarrow v}{\vdash e \downarrow^{\tau_1\times\tau_2}_{\sigma_1\times\sigma_2} \Rightarrow v}$$

Checks on function values are resolved by applying the implied checking operations to the argument and result. The object of a (nontrivial) check on a function value often does not belong to a subtype of the required type. This is the only case in which a check nonetheless succeeds under such circumstances.

$$\frac{\vdash e \Rightarrow \lambda x{:}\sigma_1.\, e_{body}}{\vdash e \downarrow^{\sigma_1\to\sigma_2}_{\tau_1\to\tau_2} \Rightarrow \lambda y : \tau_1.\, (e_{body}[x \leftarrow (y \downarrow^{\tau_1}_{\sigma_1})]) \downarrow^{\sigma_2}_{\tau_2}}$$

All checks which remain *after simplification* but do not match the two rules above fail. There is no need to specify the behavior of checks on atomic values such as 1 or true because successful checks on such values are always resolved by simplification.

$$\frac{\vdash e \rhd e' \downarrow^{\sigma}_{\tau} \quad \vdash e' \Rightarrow w \quad (w \neq \lambda x{:}\tau.\, e,\ w \neq l, r)}{\vdash e \Rightarrow \mathit{wrong}}$$

As an application of these evaluation rules, consider the evaluation of the expression z defined as

$$z = (\mathit{apply}\ ((\lambda y{:}\mathsf{Nat}.\, y)\uparrow^{\Omega}_{\mathsf{Nat}\to\mathsf{Nat}}))\ \ (3\uparrow^{\Omega}_{\mathsf{Nat}})$$

The function part of the application in z is not in normal form. By first rule for applications, the function part reduces to the normal form

$$\lambda x{:}\Omega.\, ((\lambda y{:}\mathsf{Nat}.\, y)\uparrow^{\Omega}_{\mathsf{Nat}\to\mathsf{Nat}})\downarrow^{\Omega}_{\Omega\to\Omega}\ \ x$$

Note that the *bodies* of λ-expressions do not have to be in normal form. By the definition of *apply* and the first rule for applications, the normal form of z therefore reduces to the normal form of

$$((\lambda y{:}\mathsf{Nat}.\, y)\uparrow^{\Omega}_{\mathsf{Nat}\to\mathsf{Nat}}\downarrow^{\Omega}_{\Omega\to\Omega})\ \ (3\uparrow^{\Omega}_{\mathsf{Nat}})$$

The simplified version of the function part of this application (using the simplification judgement rule) is

$$(\lambda y{:}\mathsf{Nat}.\, y)\downarrow^{\mathsf{Nat}\to\mathsf{Nat}}_{\Omega\to\mathsf{Nat}}\uparrow^{\Omega\to\Omega}_{\Omega\to\mathsf{Nat}}$$

Applying the rules for checking and tagging for function values, this reduces to the normal form

$$\lambda y{:}\Omega.\, y\uparrow^{\Omega}_{\Omega}\downarrow^{\Omega}_{\mathsf{Nat}}\downarrow^{\mathsf{Nat}}_{\mathsf{Nat}}\uparrow^{\Omega}_{\mathsf{Nat}}$$

which is equivalent to (but does not need to be reduced to) the form $\lambda y{:}\Omega.\, y\downarrow^{\Omega}_{\mathsf{Nat}}\uparrow^{\Omega}_{\mathsf{Nat}}$ mentioned in Section 3. The normal form of z therefore reduces to the normal form of

$$(\lambda y{:}\Omega.\, y\uparrow^{\Omega}_{\Omega}\downarrow^{\Omega}_{\mathsf{Nat}}\downarrow^{\mathsf{Nat}}_{\mathsf{Nat}}\uparrow^{\Omega}_{\mathsf{Nat}})\ \ (3\uparrow^{\Omega}_{\mathsf{Nat}})$$

which by the first application rule, and simplification, reduces to just $3\uparrow^{\Omega}_{\mathsf{Nat}}$ as expected.

7 Denotational Semantics

The value domain used for semantics is defined by the following domain equation, in which B is the flat domain of booleans, N is the flat domain of natural numbers, and W is the type-error domain $\{w\}_\perp$.

$$\begin{aligned} V \cong\ & B + N + (V \to V) + (V \times V) \\ & + (V \times \mathit{Typecode}) + W \end{aligned}$$

The semantics of type expressions—given in Figure 2—is almost identical to that of [ACPP89]. The only difference is that *wrong* ($=w$ in V) in our semantics represents run-time type error—the only kind there is. It belongs to *every* type rather than none. This is left implicit in the equations below. The ideal *Dynamic* used as the meaning of Ω is defined as the solution of a recursive domain equation. The details including the argument for the existence of the solution are given in [ACPP89].

$$\begin{aligned} &[\![\mathsf{Bool}]\!] = B \qquad [\![\mathsf{Nat}]\!] = N \qquad [\![\Omega]\!] = \mathit{Dynamic} \\ &[\![\tau \to \sigma]\!] = \{c \mid c([\![\tau]\!]) \subseteq [\![\sigma]\!]\} \\ &[\![\tau \times \sigma]\!] = \{\langle a, b\rangle \mid a \in [\![\tau]\!] \text{ and } b \in [\![\sigma]\!]\} \end{aligned}$$

Figure 2: **Semantics of Type Expressions**

The most important constructions in the notation used for the semantics of object expressions are:

- d in V (where $d \in D$ and D is a summand of V) is the injection of d into V. Therefore we always have $(d \text{ in } V) \in V$.
- *wrong* is just w in V.
- $v \in D$ (where $v \in V$ and D is a summand of V) yields $\perp_B$ if $v = \perp_V$, true if $v = d$ in V for some $d \in D$, and false otherwise.

$$\begin{aligned} &\mathit{tag}(x, \tau, \sigma) \\ &= \mathit{if}\ \tau = \sigma\ \mathit{then}\ x \\ &\quad \mathit{else\ if}\ \sigma = \Omega\ \mathit{then}\ \langle x, \tau\rangle \text{ in } V \\ &\quad \mathit{else\ if}\ x \in V \to V\ \mathit{then} \\ &\qquad \mathit{let}\ \tau_1 \to \tau_2 = \tau\ \mathit{and}\ \sigma_1 \to \sigma_2 = \sigma \\ &\qquad \mathit{in}\ (\lambda y.\, \mathit{tag}(x|_{V\to V}(\mathit{tag}(y, \sigma_1, \tau_1))), \tau_2, \sigma_2) \text{ in } V \\ &\quad \mathit{else\ if}\ x \in V \times V\ \mathit{then} \\ &\qquad \mathit{let}\ \langle x_1, x_2\rangle = x|_{V\times V},\ \tau_1 \times \tau_2 = \tau \\ &\qquad\qquad \mathit{and}\ \sigma_1 \times \sigma_2 = \sigma \\ &\qquad \mathit{in}\ \langle \mathit{tag}(x_1, \tau_1, \sigma_1), \mathit{tag}(x_2, \tau_2, \sigma_2)\rangle \text{ in } V \\ &\quad \mathit{else\ wrong} \end{aligned}$$

Figure 3: **Semantics of Tagging**

- $v|_D$ (where $v \in V$ and D is a summand of V) yields d if $v = d$ in V for some $d \in D$, and $\perp_D$ otherwise.

Using this notation, the "more error-prone" ordering ($\sqsupseteq$) used in Theorem 7 is defined as:

- $\forall v \in V.\ \mathit{wrong} \sqsupseteq v$.
- If $f, g \in V \to V$, then $f \sqsupseteq g \iff \forall v \in V.\ f|_{V\to V}(v) \sqsupseteq g|_{V\to V}(v)$.
- If $x, y \in V \times V$, $x|_{V\times V} = \langle x_1, x_2\rangle$ and $y|_{V\times V} = \langle y_1, y_2\rangle$, then $x \sqsupseteq y \iff x_1 \sqsupseteq y_1$ and $x_2 \sqsupseteq y_2$.

The semantic equations for object expressions use two auxiliary functions which describe the semantics of tagging and checking. Of these, the tagging function is given in Figure 3 and the function for checking is given in Figure 4. The clause for function values in the latter is the only situation in which a check succeeds even if the type of the underlying value does not belong to a subtype of the required type. The semantic equations for object expressions are given in Figure 5. All cases except those for tagging and checking operations are standard. The latter assume as before that the expressions involved are well-formed.

$$
\begin{array}{l}
check(x,\tau,\sigma) \\
= \mathit{if}\ \tau = \sigma\ \mathit{then}\ x \\
\quad \mathit{else\ if}\ \tau \leq \sigma\ \mathit{then}\ tag(x,\tau,\sigma) \\
\quad \mathit{else\ if}\ x \in V \times \mathit{Typecode}\ \mathit{then} \\
\qquad \mathit{let}\ \langle y,\mu\rangle = x|_{V\times \mathit{Typecode}}\ \mathit{in}\ check(y,\mu,\sigma) \\
\quad \mathit{else\ if}\ x \in V \to V\ \mathit{then} \\
\qquad \mathit{let}\ \tau_1 \to \tau_2 = \tau\ \mathit{and}\ \sigma_1 \to \sigma_2 = \sigma \\
\qquad \mathit{in}\ (\lambda y.\, check\,(x|_{V\to V}\,(check(y,\sigma_1,\tau_1))),\tau_2,\sigma_2) \\
\quad \mathit{else\ if}\ x \in V \times V\ \mathit{then} \\
\qquad \mathit{let}\ \langle x_1,x_2\rangle = x|_{V\times V},\ \tau_1 \times \tau_2 = \tau \\
\qquad\qquad \mathit{and}\ \sigma_1 \times \sigma_2 = \sigma \\
\qquad \mathit{in}\ \langle check(x_1,\tau_1,\sigma_1), check(x_2,\tau_2,\sigma_2)\rangle \\
\quad \mathit{else\ wrong}
\end{array}
$$

Figure 4: **Semantics of Checking**

The connection between the operational and denotational semantics is expressed in the following theorem, which simply asserts that operational evaluation preserves denotational meaning.

Theorem 9 (Soundness of Evaluation) *Given an arbitrary expression e and a normal form expression v,* $\vdash e \Rightarrow v$ *implies that* $[\![e]\!]\emptyset = [\![v]\!]\emptyset$.

It is important to note that the operational semantics is less abstract than the denotational semantics (otherwise the implication in Theorem 9 would be an equivalence) and the denotational semantics itself is not fully abstract. For instance, the values $\langle\langle 2, \mathsf{true}\rangle,\ \mathsf{Nat} \times \mathsf{Bool}\rangle$ and $\langle\langle\langle 2, \mathsf{Nat}\rangle, \mathsf{true}\rangle,\ \Omega \times \mathsf{Bool}\rangle$ are derived for the expressions $(2, \mathsf{true})\uparrow^{\Omega}_{\mathsf{Nat}\times\mathsf{Bool}}$ and $(2\uparrow^{\Omega}_{\mathsf{Nat}},\ \mathsf{true})\uparrow^{\Omega}_{\Omega\times\mathsf{Bool}}$ respectively, but these values cannot be observably distinguished. It does not seem worth while to complicate the semantics to avoid this quirk but the necessary modification is easily made if needed.

$$
\begin{array}{l}
[\![e\downarrow^{\tau}_{\sigma}]\!]\rho = \mathit{let}\ x = [\![e]\!]\rho\ \mathit{in}\ \ check(x,\tau,\sigma) \\
[\![e\uparrow^{\sigma}_{\tau}]\!]\rho = \mathit{let}\ x = [\![e]\!]\rho\ \mathit{in}\ \ tag(x,\tau,\sigma) \\
[\![e_{fun}\ e_{arg}]\!]\rho = \mathit{let}\ f = [\![e_{fun}]\!]\rho\ \mathit{and}\ x = [\![e_{arg}]\!]\rho\ \mathit{in} \\
\qquad\qquad \mathit{if}\ f \notin V \to V\ \mathit{then\ wrong\ else}\ f|_{V\to V}(x) \\
[\![e_{left},\ e_{right}]\!]\rho = \mathit{let}\ l = [\![e_{left}]\!]\rho\ \mathit{and}\ r = [\![e_{right}]\!]\rho \\
\qquad\qquad \mathit{in}\ \langle l, r\rangle \\
[\![\lambda x:\tau.\, e]\!]\rho = \lambda v.\, [\![e]\!](\rho[x \leftarrow v])
\end{array}
$$

Figure 5: **Semantics of Object Expressions**

8 Pragmatic Issues

The implementation of our system does not present any fundamentally new problems (see [ACPP89] for a discussion of relevant techniques). Type matching in our system clearly requires Amber-like structural representation of types at run-time since the subtyping scheme is based on structural matching.

From a pragmatic viewpoint, the differences between the operational and denotational semantics raise interesting questions about the details of an implementation. For instance, tagging of pair values is propagated to components in the latter but not in the former. Propagation in the denotational semantics is forced because the ideal representing tagged values makes no provision for the "result type" tag $\Omega \times \Omega$ in a normal form such as $(2, \mathsf{true})\uparrow^{\Omega\times\Omega}_{\mathsf{Nat}\times\mathsf{Bool}}$. Similar remarks apply to other data structures. For instance, the tagging in the example $l_3 = l\uparrow^{[\Omega]}_{[\mathsf{Nat}]}$ used in Section 2 must be propagated to components in the denotational semantics. There is essentially a tradeoff between time and space here. An implementation according to the operational semantics must often provide space for two tags per tagged value instead of one, but both tagging and checking would on average be faster than in an implementation according to the denotational semantics.

A related issue is raised by applications such as

sum $([1\uparrow^{\Omega}_{\mathsf{Nat}}, 2\uparrow^{\Omega}_{\mathsf{Nat}}, 3\uparrow^{\Omega}_{\mathsf{Nat}}, 4\uparrow^{\Omega}_{\mathsf{Nat}}]\downarrow^{[\Omega]}_{[\mathsf{Nat}]})$. A straightforward implementation of such checks would involve two passes over the list: one to carry out the check and another to carry out the sum (since the check is successful). A possible method to avoid this is to make checks on data structures "lazy"—propagated when the structure is eliminated. This would seem to erase our advantage—over LISP-like languages—of early detection of type errors. However, information about the *origin* of the check can be propagated along with the check, and used to pinpoint the actual source of the error if a propagated check fails. Although the detection of the error would be delayed, the reported error would be the same as in the straightforward implementation, at the cost of some overhead in carrying the information about the source of a check.

A system with automatic transitions from static to dynamic typing raises some serious methodological issues. Cardelli [Car89] points out that automatic generation of run-time *checks* could be dangerous since users might not be aware of the points where they are being used. A slight change to a statically well-typed program may unwittingly cause a whole set of automatic dynamic checks to be inserted. This could reduce the robustness of the program below the programmer's intentions. One solution would be to modify our system to require the user's "written permission" (through a dialog box, for instance) for insertion of checks. The denial of such permission is equivalent to a static type error. Such interaction may become tedious if intended checks are numerous. A better solution is to provide two different constructs for function application—a strict one which prohibits checks and a permissive one which allows them. The programmer would then retain responsibility for insertion of checks without the burden of writing the corresponding code. The language we consider in this paper does not have these safeguards, but their introduction does not seem to pose any technical problems.

9 Concluding Remarks

We have described a system that merges static and dynamic typing with very little added complexity at the user level. The interpretation of dynamic typing we use is more abstract than existing static systems which allow some dynamic typing and is closer in spirit to that in LISP-like pure dynamic systems.

Two underlying themes in the paper are worth recalling. One is the use of coercive structural subtyping as a way of specifying simple kinds of program synthesis. This technique is very promising as a way of adding expressive power to a language at relatively little cost in semantic complexity. An application of the technique to APL-like implicit scaling is described in [Tha89]. Robinson and Tennent [RT88] have suggested an application to the record update problem.

A related theme is the idea of negative coercions and plausibility checking. This is clearly applicable in many subtyping situations, both inclusive and coercive. A simple example is automatic generation (and plausibility) of bounds checks for subranges. More complex examples might involve labeled records or explicitly declared inheritance relationships among abstract types. As we mentioned in the introduction, much of the theory presented above can be generalized to account for a large class of such examples.

The coercion based interpretation of inheritance used in this paper is similar in spirit to the framework of [BTCGS89]. We hope to explore this connection to generalize the present system to polymorphic and recursive types.

10 Acknowledgements

I would like to thank Luca Cardelli for a discussion which clarified some basic ideas for me, and Jens Dill, Uwe Pleban and Fritz Ruehr for helpful comments on a previous version of this paper.

References

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[ACPP89] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. In *Proc. of Sixteenth POPL Symposium*. ACM, January 1989.

[BTCGS89] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. of Fourth LICS Symposium*. IEEE, June 1989.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*. Springer Verlag, 1986. Lecture Notes in Computer Science, Vol. 242.

[Car89] Luca Cardelli, January 1989. Personal Communication.

[CDJ+89] Luca Cardelli, James Donahue, Mick Jordan, Bill Kaslow, and Greg Nelson. The modula-3 type system. In *Proc. of Sixteenth POPL Symposium*. ACM, January 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), 1985.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting. *J. Assoc. Comp. Mach.*, 27(4):797–821, 1980.

[RT88] Edmund Robinson and Robert Tennent, October 1988. Note sent to "Types" E-mail forum.

[Tha88] Satish R. Thatte. Type inference with partial types. In Timo Lepisto and Arto Salomaa, editors, *Automata, languages and programming : 15th International Colloquium*, pages 615–629. Springer-Verlag, July 1988. Lecture Notes in Computer Science, Vol. 317.

[Tha89] Satish R. Thatte. Type inference and implicit scaling, 1989. To Appear.

[Wad76] C. Wadsworth. The relation between computational and denotational properties for Scott's D_∞ models of the λ-calculus. *SIAM J. Comput.*, 5(3):488–520, 1976.