
Demand transformation analysis for concurrent constraint programs

Moreno Falaschi, Patrick Hicks, William Winsborough

- ▷ This paper presents a demand transformation analysis that maps a predicate's output demands to its input demands. This backward dataflow analysis for concurrent constraint programs is constructed in the framework of abstract interpretation. In the context of stream parallelism, this analysis identifies an amount of input data for which predicate execution can safely wait without danger of introducing deadlock. We assume that programs are well-moded and prove that our analysis is safe. We have constructed an implementation of this analysis and tested it on some small, illustrative programs and have determined that it gives useful results in practice. We identify several applications of the analysis results to distributed implementations of concurrent constraint languages, including thread construction and communication granularity control. This analysis will enable existing computational cost estimation analyses to be applied to stream-parallel logic languages. ◁
-

1. Introduction

The cheapest source of large numbers of instruction cycles is found in networks of low-end, off-the-shelf processors (workstation farms). However, loosely coupled distributed platforms remain notoriously difficult to program. It is possible that, by using declarative languages with implicit, fine-grain parallelism, the programmer will specify communication and synchronization at a sufficiently high level to make the task of writing distributed programs economical for general-purpose applications.

Address correspondence to Moreno Falaschi, Dipartimento di Matematica e Informatica, Via delle Scienze, 206, Udine, Italy. email: falaschi@dimi.uniud.it;
Patrick Hicks, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, USA. phicks@cse.psu.edu;
William Winsborough, Transarc Corporation, Gulf Tower, 707 Grant Street, Pittsburgh, PA 15219, USA. winsboro@transarc.com.

THE JOURNAL OF LOGIC PROGRAMMING

The present investigation considers concurrent logic programming and its generalization, concurrent constraint programming. There are several problems with implementing such languages effectively on loosely coupled, distributed memory multi-processors. In particular, processes and messages can be too fine grained. We believe that three tools would begin to address these problems.

1. *Computational cost estimation* [12, 13, 26] gives estimates of the computational cost of a procedure as a function of its input size. This is useful in establishing a threshold on input size above which it is economical to execute the computation remotely, but existing analyses do not currently support stream parallelism.
2. *Compile-time scheduling* orders execution steps at compile time, assembling threads and thereby reducing the overhead of runtime scheduling. It can also eliminate tests for suspension by reasoning at compile time about what data will be available at runtime. It is likely that compile-time scheduling can be based on the abstract transition systems developed in [4] for suspension analysis. This is a subject for future work.
3. *Demand transformation analysis* finds out how much input is necessary for a procedure to generate a certain amount of output. For example, a process that performs list reversal can begin producing output only upon receiving a nil-terminated list. However, waiting can introduce deadlock if there are cyclic dependencies or infinite data structures. Demand transformation analysis can be used to identify an amount of input that it is safe to wait for.

The present discussion focuses on demand transformation analysis. We raise the other two as motivation because this analysis can be used to enable them, as described presently.

The application of demand transformation analysis to a predicate yields a mapping from an output demand to an input demand. Input and output demands are measures of data, which in our context means how constrained the shared data structures must be. The current approach quantifies such a demand by specifying a non-ground type, rather than for instance a term size. (A non-ground type [23, 19] is a regular set of terms that possibly contain unbound variables.)

The safety requirement of the analysis is that, if the input provided to the process is less determined than required by the input demand, the output provided by the process will be less determined than required by the output demand. For instance, it is always trivially safe (albeit useless) to map any output demand to *no* input.

The languages we consider support stream parallelism. When partial input data becomes available, the system must determine whether to begin executing the consumer immediately or to wait for further input. There are often advantages to waiting. When data must be transmitted remotely, waiting allows message size to be increased, which reduces communication overhead. When creating a new process, waiting allows a more accurate prediction of whether the computation-to-communication cost ratio would justify remote execution, based on input size and computational cost estimates. Computational cost analyses have been developed for use with independent-and parallelism [12, 13, 26]. In the context of stream parallelism the technique entails the introduction of additional synchronization constraints: execution must wait until the input is sufficiently determined to enable

the runtime system to decide where to execute the process. Finally, waiting makes it less likely that a process will be started only to suspend due to insufficient input. Demand transformation analysis determines when it is safe to wait. It identifies cases where it is justified using scheduling rules that promote coarse-grained parallelism, but that in general are unfair.

Demand transformation analysis also enables a compiler to generate code that is optimized knowing that a consumer process waits until its input demand is met before executing. When the consumer executes, the demanded input will already be available locally. This eliminates the need for subsequent runtime checks and possibly eliminates suspension. Compile-time scheduling can also be performed, generating code that consumes the demanded input without the intervention of the runtime scheduler.

Applying demand transformation analysis to a process identifies inherent data dependence of output on input. If a process waits on an input demand prior to executing, we can view this as introducing a dependence of all further output on the input being waited for. The demand transformation analysis safety condition requires that, if a process waits on the input demand computed by the analysis, the dependence that is introduced was already present in the form of data dependence of output on input. Thus, no new dependence is introduced. We prove that our analysis is safe.

Demand transformation analysis is not the only possible approach to finding safe input demands. An alternative approach would analyze data dependence in the context where the consumer is invoked. If the dependence added by waiting on an input demand introduces no cyclic dependence, waiting is safe. An advantage of analyzing the contextual dependences would be that cyclic dependence is probably relatively rare. However, it has the drawback that it requires access to the source code from which the consumer is called, which interferes with modular compilation and is entirely unrealistic in many systems applications. By contrast, demand transformation analysis can be carried out on a process without access to the code in which the process is invoked. Note that these two kinds of analysis complement one another as demand transformation analysis characterizes the process itself while the other characterizes the context in which the process is used.

In functional programming, abstract interpretation has been used extensively to analyze *strictness*, beginning with the seminal work by Mycroft [31]. There is a family of analyses that generalize strictness by using the context of a function application to justify more eager evaluation of the argument than would be justified by strictness analysis alone. Methods that utilize context include *projection analysis* [9, 35], *evaluation-transformer analysis* [1, 2], *stream-strictness analysis* [16], *inverse image analysis* [14], and Lindstrom’s *backwards strictness analysis* [28]. Such methods, as well as demand transformation analysis, can be collected under the general rubric of *context analysis*. Although abstract interpretation has been applied to concurrent constraint programming (CCP) [4, 3, 5, 11, 27, 36] for many other applications, to our knowledge it has never been used in CCP or logic programming for context analysis. A preliminary presentation of the current research appeared as [15].

Organization is as follows. Section 2 reviews the set of Herbrand constraints and the powerdomain construction yielding our concrete domain of interpretation. It further reviews the syntax and semantics of CCP. Section 3 presents the abstract domain used in the construction of our demand-transformation analysis. It specifies

the well-modding requirement on programs analyzed by our method. It also presents the *demand collecting semantics* on which the analysis is based and the proof of correctness of the analysis. Section 4 discusses the implementation. Section 5 presents a detailed example of our analysis. Section 6 gives empirical results obtained by running the analyzer on various exemplary programs. Section 7 discusses other research in abstract interpretation of concurrent constraint programming as well as on context analyses for functional languages. Finally, Section 8 summarizes and suggests future work. We include an appendix containing the proof of the main technical lemma.

2. Concurrent Constraint Programming

This section begins by reviewing the domain of existentially quantified Herbrand equalities. While general CCP is based on cylindric constraint systems, we restrict our attention to these Herbrand constraints; it is a matter of future research to generalize our analysis. The section then presents a construction lifting Herbrand constraints to the powerdomain that forms the concrete domain of our abstract interpretation in the sequel. The section then turns to the syntax and operational semantics of the CCP language, culminating in the operational demand transformation semantics that forms the reference point of our analysis.

2.1. Herbrand constraints and their powerdomain

We confine our attention to constraint domains based on equational theories. Each individual computation has its constraint pool given by a possibly existentially quantified conjunction of equalities over Herbrand terms. The domain of such constraints is given by C . We write H_V for the set of possibly non-ground finite terms over the countably infinite set of program variables, V , and some fixed, countably infinite set of constructors (unevaluable function symbols). Each element of C has the form $\exists_{\bar{x}}(\bigwedge_{i \in I} s_i = t_i)$, where $s_i, t_i \in H_V$, $\bar{x} \subseteq \text{vars}(\{s_i = t_i\}_{i \in I})$, and I is a finite index set. We use the standard first-order existential here: if σ is a solution to some $c \in C$, then σ' is a solution to $\exists x.c$, where σ' is any assignment that is identical to σ on all variables except x , where it is arbitrary.

It is standard to use some form of existential quantification in semantic constructions to project constraints onto variables of interest. The following proposition is useful for managing the introduction and elimination of local, fresh variables.

Proposition 2.1. $\exists_x.(x = y \wedge c) \Leftrightarrow c$ if $x \notin \text{vars}(c)$.

PROOF. Suppose σ is a solution for $\exists_x.x = y \wedge c$. Then there must be a σ' that is identical to σ , except possibly on x , that is a solution to $x = y \wedge c$. σ' is clearly a solution to c as well. But, since x does not appear in c , that means σ is also a solution to c . For the converse, suppose there is a solution to c . Call it σ . Because x does not appear in c , σ' is also a solution to c where σ' is identical to σ , except that it maps x to the same value as y . Clearly σ' is a solution to $x = y \wedge c$. This proves that σ is a solution to $\exists_x.x = y \wedge c$. \square

We take $c_1, c_2 \in C$ to be ordered by logical entailment (implication): $c_1 \leq c_2$ if $c_1 \vdash c_2$. This makes *false* be the bottom element and conjunction yield the greatest lower bound. It is hoped that this reversal of the familiar information ordering does

not distract the reader. This constraint ordering has the advantage of inducing a powerdomain ordering that agrees with the subset relation, which conforms with abstract interpretation convention.

An important property in the sequel is that existentials are increasing: for all constraints c , $c \leq \exists_x c$. For further information on the general properties of cylindric algebras and of Herbrand constraints, see respectively [18] and [29].

We use the powerdomain as the concrete domain of our demand collecting semantics. The powerdomain construction takes $S_1, S_2 \subseteq C$ to be preordered by the *Hoare preordering* $S_1 \sqsubseteq S_2$ if $\forall c_1 \in S_1. \exists c_2 \in S_2. c_1 \leq c_2$. S_1 and S_2 are Hoare equivalent if $S_1 \sqsubseteq S_2$ and $S_1 \supseteq S_2$. Briefly reviewing this standard construction, the quotient of the power set $\wp(C)$ with respect to the Hoare equivalence is a complete lattice whose elements are equivalence classes partitioning $\wp(C)$. Each of those equivalence classes contains a canonical representative that is downwards-closed with respect to \leq . Specifically, for $S \in \wp(C)$, if δ is the downwards closure of S , δ is Hoare-equivalent to S . The quotient domain is therefore isomorphic to and viewed as the domain of downwards-closed subsets of C , which we denote by $\wp\downarrow(C)$. Note that if $S_1, S_2 \in \wp(C)$ and if $\delta_1, \delta_2 \in \wp\downarrow(C)$ are their respective downwards closures (i.e., canonical representatives), then $S_1 \sqsubseteq S_2$ if and only if $\delta_1 \subseteq \delta_2$.

Proposition 2.2. For $c_1 \in \delta_1$, $c_2 \in \delta_2$ implies $c_1 \wedge c_2 \in \delta_1 \cap \delta_2$

Our demand collecting semantics models computation over $\wp\downarrow(C)$. To support the embedding of individuals, we introduce $\downarrow: C \rightarrow \wp\downarrow(C)$ given by $\downarrow c = \{b \in C \mid b \leq c\}$. For instance, $\downarrow \text{true} = C$. Also note that $b \leq c$ implies $\downarrow b \subseteq \downarrow c$. We write $\text{vars}(\delta)$ for the set of variables appearing (non-trivially) in maximal elements of δ .

Because domain elements are downwards-closed sets, conjunction of individuals lift naturally to intersection of domain elements. We must also lift existential quantification to $\wp\downarrow(C)$. Since downwards closure is not preserved by pointwise application of the quantifier to a set's elements, we implicitly follow pointwise application by taking the minimal downwards-closed superset, which always exists. This additional step is necessary only because we find it convenient to work with the canonical representatives given by $\wp\downarrow(C)$ instead of the working directly with the Hoare-equivalence classes discussed above. In this way we have $\exists_V: \wp\downarrow(C) \rightarrow \wp\downarrow(C)$.

Proposition 2.3. The following properties follow easily from the pointwise lifting of \exists :

1. For $c \in C$ and $\delta \in \wp\downarrow(C)$, $c \in \delta$ implies $\exists_x c \in \exists_x \delta$
2. For $\delta \in \wp\downarrow(C)$, $\exists_x \exists_y \delta = \exists_y \exists_x \delta$
3. \exists is \cap -continuous on $\wp\downarrow(C)$.
4. For $\delta \in \wp\downarrow(C)$ with $x \notin \text{vars}(\delta)$, $\exists_x(\downarrow x = y \cap \delta) \Leftrightarrow \delta$.

PROOF. Proposition 2.3.3 follows easily from the pointwise definition of the existential on sets. We prove proposition 2.3.4:

$$\begin{aligned}
& c \in \exists_x(\downarrow(x = y) \cap \delta) \\
& \iff \text{there exists a maximal } d \in \delta \text{ such that } c \leq \exists_x(x = y \wedge d) \\
& \iff \text{there exists a maximal } d \in \delta \text{ such that } c \leq d, \text{ by Proposition 2.1}
\end{aligned}$$

$$\iff c \in \delta$$

□

For any finite set $V = \{v_1, \dots, v_n\}$ of variables and any $\delta \in \wp \downarrow(C)$, we write $\exists_V \delta$ as shorthand for $\exists_{v_1} \exists_{v_2} \dots \exists_{v_n} \delta$. For a syntactic object s , $\exists_s \delta$ is shorthand for $\exists_{\text{vars}(\delta) \setminus \text{vars}(s)} \delta$. Intuitively, $\exists_s \delta$ restricts the constraints in δ to the variables in syntactic object s , while preserving downwards closure.

2.2. The language

We consider an instance of concurrent constraint programming (CCP—see [32] for details). CCP is described by the following syntax

$$\begin{aligned} \text{Declarations } D &::= \epsilon \mid Cl \mid D_1, D_2 \\ \text{Clause } Cl &::= p(\bar{x}) :- A \\ \text{Agents } A &::= \mathbf{Stop} \mid \mathbf{tell}(c) \mid \sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i \mid A_1 \parallel A_2 \mid p(\bar{x}) \end{aligned}$$

The agent **Stop** represents successful termination. In $\mathbf{ask}(c)$ and $\mathbf{tell}(c)$, c is a *constraint*. These actions work on a common *store* which ranges over C . The execution of $\mathbf{tell}(c)$ in the store d sets the store to the conjunction $c \wedge d$. $\mathbf{ask}(c)$ is a *guard*: it performs a test on the current store d and it is *enabled* if d implies c ($d \leq c$). It does not modify the store. The *guarded choice* agent $\sum_{i=1}^n g_i \rightarrow A_i$ nondeterministically selects one guard g_i that is enabled, and then behaves like A_i . If no guards are enabled, then it *suspends*, waiting for other (parallel) agents to add information to the store. The symbol \parallel represents parallel composition. The agent $p(\bar{x})$ is a procedure call, where p is the name of the procedure and \bar{x} is the actual parameter. The meaning of $p(\bar{x})$ is given by a single *clause* of the form $p(\bar{y}) :- A$, where \bar{y} is the formal parameter. We use $+$ as a shorthand for $\sum_{i=1}^2$.

2.3. Operational demand transformation semantics

The operational model of CCP is described in terms of a transition system $T_D = (\text{Conf}, \longrightarrow)$ which is specified w.r.t. a given set of clauses, $D \in \text{Progs}$, called a *program*. Each configuration in Conf is a pair consisting of an agent and a constraint representing the store. A *partial derivation* is a finite sequence of configurations in which adjacent pairs are related by \longrightarrow . A *derivation* is a sequence of configurations in which adjacent pairs are related by \longrightarrow and that is infinite or whose final configuration is not related to any other configuration by \longrightarrow . $\text{defn}_D(A)$ is the set of variants of rules in D such that each variant has A as a head and, apart from the variables in A , has distinct new variables. Table 2.1 describes the rules of T_D .

The transition semantics of CCP considers infinite as well as finite executions. We provide a formal definition of the observables of an agent as background for the demand transformation semantics that our analysis approximates. In the following we assume the program is fixed.

Definition 2.1. The mapping $\mathcal{O} : \text{Agents} \rightarrow \wp(C \times C)$ which gives the observables of an agent, is defined by $\mathcal{O}(A) = \mathcal{O}_{\text{Fin}}(A) \cup \mathcal{O}_{\text{Inf}}(A)$, with

$$\begin{aligned} \mathcal{O}_{\text{Fin}}(A) &= \{ \langle c, d \rangle \mid \text{there exists } B \text{ s.t. } \langle A, c \rangle \longrightarrow^* \langle B, d \rangle \not\longrightarrow \} \\ \mathcal{O}_{\text{Inf}}(A) &= \left\{ \langle c, \text{lub}_{n \in \omega} (c_n) \rangle \mid \text{there exists an infinite derivation} \right. \\ &\quad \left. \langle A, c \rangle \longrightarrow \langle B_1, c_1 \rangle \longrightarrow \dots \langle B_n, c_n \rangle \longrightarrow \dots \right\} \end{aligned}$$

R1	$\langle \mathbf{tell}(c), d \rangle \longrightarrow \langle \mathbf{Stop}, c \wedge d \rangle$
R2	$\langle \sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i, d \rangle \longrightarrow \langle A_j, d \rangle \quad j \in [1, n] \text{ and } d \leq c_j$
R3	$\frac{\langle A, c \rangle \longrightarrow \langle A', c' \rangle}{\langle A \parallel B, c \rangle \longrightarrow \langle A' \parallel B, c' \rangle}$ $\langle B \parallel A, c \rangle \longrightarrow \langle B \parallel A', c' \rangle$
R4	$\langle p(\bar{x}), c \rangle \longrightarrow \langle A, c \rangle \quad p(\bar{x}) :- A \in \mathit{defn}_D(p(\bar{x}))$

TABLE 2.1. The transition system T_D

where $\not\rightarrow$ denotes the absence of outgoing transitions and \longrightarrow^* denotes the reflexive and transitive closure of \longrightarrow . We also define the set of observables corresponding to the partial derivations:

$$\mathcal{O}_{\text{part}}(A) = \{ \langle c, d \rangle \mid \text{there exists } B \text{ s.t. } \langle A, c \rangle \longrightarrow^* \langle B, d \rangle \}$$

The following demand transformation semantics defines the mapping from output demands to input demands that our analysis safely approximates. The definition says that the input demand for an agent A is given by the set of input stores that each enable a partial derivation of A in which the computed output is at least as strong as the output demand.

Definition 2.2. The demand transformation semantics for the program P and agent A is given by $\mathcal{D}_P : \text{Agents} \rightarrow C \rightarrow \wp \downarrow(C)$, where

$$\mathcal{D}_P[A]b = \{ c \mid \exists \langle B, d \rangle. \langle A, c \rangle \longrightarrow^* \langle B, d \rangle \ \& \ d \leq b \}$$

Note that each $c \in \mathcal{D}_P[A]b$ corresponds to some partial derivation, d , that starts with $\langle A, c \rangle$.

3. Constructing a computable analysis

The technique of abstract interpretation [6] is used in the construction of the analysis. The specification of the demand transformation analysis is obtained from our demand collecting semantics (constructed below in Section 3.3) by replacing the concrete domain, $\wp \downarrow(C)$, with an abstract domain (presented in Section 3.1) and reinterpreting the operations in the construction accordingly. The abstract operations must upper approximate the concrete operations in the standard way [6]. When this is the case, the abstract interpretation of the demand collecting semantics, which specifies the analysis that we have implemented, can be shown to safely approximate the concrete interpretation by using standard techniques [6].

The construction of the demand collecting semantics is presented in terms of the concrete domain. In Section 3.4, Theorem 3.1 relates that demand collecting semantics to the demand transformation semantics presented in Definition 2.2 above.

Thus the formal relationship between the analysis and the property of interest is completed. Section 3.2 presents well-moding requirements of the programs that are analyzed.

3.1. Abstract Domain

In this section we construct a domain of abstract constraints called $ACon$, which abstracts the domain $\wp\downarrow(C)$. In the construction of $ACon$, we use two domains called \mathcal{D} and \mathcal{D}_V , also introduced in this section, which consist of non-ground, downwards-closed types representing sets of terms in $\wp\downarrow(H_V)$ and some basic types, such as the set of integers. (H_V is ordered by $t_1 \leq t_2$ if t_1 is a substitution instance of t_2 .)

The domain of types is given by $\mathcal{D} ::= \top \mid ? \mid \tau \mid c(\mathcal{D}, \dots, \mathcal{D}) \mid num \mid \mathcal{D} \vee \mathcal{D} \mid \mu\tau.\mathcal{D}$. Program variables are not mentioned by types in \mathcal{D} . In the syntax of \mathcal{D} , c ranges over constructor symbols and μ is a fixpoint operator. Type variables are given by $\tau \in TV$, which are used only for fixpoint constructions. The base types \top , $?$ (read, “non-var”), and num represent H_V , $H_V \setminus V$, and the set of integers, respectively.

Example 3.1. $\{X = \top, Y = ?\}$ is an element of $ACon$ representing the downwards-closed set of constraints where X is constrained arbitrarily (including not at all) but Y must be constrained to be either a constant or a compound term.

Example 3.2. $\mu\tau.[\] \vee [\top \mid \tau]$ is an element of \mathcal{D} denoting the set of nil-terminated lists whose elements need not be constrained.

Example 3.3. Note that $\mu\tau.[[X] \mid \tau] \vee [\]$ is not in \mathcal{D} . However, as we see below, $\{X = \top, Y = \mu\tau.[[\top] \mid \tau] \vee [\]\}$ represents a set of concrete constraints in which $Y = [[X]]$ may or may not hold.

A type environment is given by $\rho : TENV = TV \rightarrow \wp\downarrow(H_V)$. Let us construct $\llbracket \cdot \rrbracket : \mathcal{D} \rightarrow TENV \rightarrow \wp\downarrow(H_V)$.

$$\begin{aligned} \llbracket \tau \rrbracket \rho &= \rho(\tau) \\ \llbracket \top \rrbracket \rho &= H_V \\ \llbracket ? \rrbracket \rho &= H_V \setminus V \\ \llbracket c(d_1, \dots, d_n) \rrbracket \rho &= \{c(t_1, \dots, t_n) \mid t_i \in \llbracket d_i \rrbracket \rho\} \\ \llbracket d_1 \vee d_2 \rrbracket \rho &= \llbracket d_1 \rrbracket \rho \cup \llbracket d_2 \rrbracket \rho \\ \llbracket \mu\tau.d \rrbracket \rho &= lfp(\lambda S : \wp\downarrow(C). \llbracket d \rrbracket (\rho[\tau/S])) \end{aligned}$$

Here, lfp denotes least fixpoint. The denotation of $d \in \mathcal{D}$ is given by $\llbracket d \rrbracket \varepsilon$, where ε is the type environment that maps all type variables to the empty set. We assume that union of types implicitly forms the Cartesian closure: Restricting attention briefly to a single binary function symbol, f , a set T is *Cartesian closed* if $f(a, b), f(c, d) \in T$ implies $f(a, d), f(c, b) \in T$. (This corresponds to the “principal function restriction” of [23].)

The domain of abstract constraints consists of systems of equations over PV -types, which are given by the syntactic category \mathcal{D}_V . Program variables are per-

mitted in PV-types (whence the ‘PV’), but only outside of unions and fixpoints¹. This enables our abstract domain to express definite aliasing at constant depth. We overload $\llbracket \cdot \rrbracket$ as follows.

$$\begin{aligned} \mathcal{D}_V &::= V \mid \mathcal{D} \mid c(\mathcal{D}_V, \dots, \mathcal{D}_V) \\ \llbracket \cdot \rrbracket &: \mathcal{D}_V \rightarrow \wp \downarrow (H_V) \\ \llbracket X \rrbracket &= X \\ \llbracket d \rrbracket &= \llbracket d \rrbracket \varepsilon, \text{ for } d \in \mathcal{D} \\ \llbracket c(d_1, \dots, d_n) \rrbracket &= \{c(t_1, \dots, t_n) \mid t_i \in \llbracket d_i \rrbracket\} \end{aligned}$$

Elements of \mathcal{D} and of \mathcal{D}_V represent sets of terms, not sets of constraints. Our domain of abstract constraints, $ACon$, consists of (equivalence classes of) sets of equations over PV-types in \mathcal{D}_V . (In addition, $ACon$ contains a bottom element, \perp .) The demand transformation analysis we have implemented based on the demand collecting semantics (see Section 3.3) computes the relevant values of a function in $Agents \rightarrow ACon \rightarrow ACon$, mapping an agent and an output demand to an input demand.

The downwards closed set of conjunctions of Herbrand equations represented by a set of equations $E \in ACon$ is given by the following.

$$\begin{aligned} \llbracket E \rrbracket &= \left\{ \sigma \in C \mid \begin{array}{l} \text{for all } d_1 = d_2 \in E, \text{ there exist } t_1 \in \llbracket d_1 \rrbracket \text{ and } t_2 \in \llbracket d_2 \rrbracket \\ \text{such that } \sigma \leq t_1 = t_2 \end{array} \right\} \\ \llbracket \perp \rrbracket &= \downarrow \text{false} \end{aligned}$$

The order on sets of abstract constraints is induced under $\llbracket \cdot \rrbracket$ by the order on $\wp \downarrow (C)$. Equivalent abstract constraints are those that denote the same elements of $\wp \downarrow (C)$. The top element is given by the empty system of abstract equations, which denotes $\downarrow \text{true} \in \wp \downarrow (C)$. The bottom element is \perp , which denotes $\downarrow \text{false}$.

It is necessary to supply certain operations over PV-types to support their use as a domain of interpretation of our demand collecting semantics. These operations are analogues of the operations basic operations on $\wp \downarrow (C)$ used in the demand collecting semantics below. For our type domain (discussed in Section 3.1), intersection of downwards-closed sets of constraints is modeled by (abstract) unification of PV-type terms. Existential quantification is modeled by projection. Taking least upper bounds introduces \vee into type terms. These three operations are slight variants of those described in [23].

The μ operator is introduced into type terms by using a widening operation [6] proposed in [19]. Widening introduces approximation into the fixpoint calculation, allowing an upper approximation of the least fixpoint to be computed finitely, even in a domain containing infinite increasing chains.

3.2. Modes

The correctness of the demand collecting semantics relies on programs being well moded [33]. We assume a coarse form of modes that divides arguments into input and output. We assume that program atoms have the form $p(\bar{x}, \bar{y})$, where \bar{x} is a

¹Allowing program variables within unions and fixpoints adds technical problems to the construction with no significant benefit in expressivity.

vector of input variables and \bar{y} is a vector of output variables. It would be necessary to generalize this to richer modes (e.g. those in [33]) if we were going to handle two-way streams. However, we use the coarser form here, as it is adequate for one-way streams. It is occasionally necessary to refer to the output variables of an agent in the program. We let $\text{Out}(p(\bar{x}, \bar{y})) = \bar{y}$. For compound agents, we presume a consistent moding exists and refer to output variables of agent A by $\text{Out}(A)$.

We require the following properties of the mode system:

1. Suppose that $x \in \text{Out}(A)$ and that d is a partial derivation in which A is introduced but *not reduced*. Let the final constraint of d be σ . Then $\bar{\exists}_x(\sigma) = \text{true}$.
2. Suppose d is a partial derivation whose initial configuration is $\langle A, \sigma_0 \rangle$ and whose final configuration is $\langle B, \sigma_f \rangle$. Then $\bar{\exists}_{\text{In}(A)}\sigma_0 = \bar{\exists}_{\text{In}(A)}\sigma_f$. (This requirement states that agents (and their descendants) may not further instantiate their own input variables.)
3. If $p(\bar{x}, \bar{y}) :- A$ is a clause, then $\bar{x} \subseteq \text{In}(A)$ and $\bar{y} \subseteq \text{Out}(A)$.
4. Each agent A of the form $\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i$ has
 - (a) $\text{In}(A) \subseteq \cup_{i=1}^n (\text{vars}(c_i) \cup \text{In}(A_i))$
 - (b) $\text{In}(A) \supseteq \cup_{i=1}^n \text{In}(A_i)$
 - (c) $\text{Out}(A) = \cup_{i=1}^n \text{Out}(A_i)$
5. Each agent A of the form $A_1 \parallel A_2$ has $\text{In}(A) \subseteq \text{In}(A_1) \cup \text{In}(A_2)$, $\text{Out}(A) \subseteq \text{Out}(A_1) \cup \text{Out}(A_2)$, and $\text{Out}(A_1) \cap \text{Out}(A_2) = \emptyset$.

3.3. Demand collecting semantics

The demand collecting semantics is constructed in Figure 3.1. The demand collecting semantics defines a function environment $e : \text{Env}$, $\text{Env} = \text{Atom} \rightarrow \wp \downarrow(C) \rightarrow \wp \downarrow(C)$. For each atom, e maps a set of output constraints to a set of input constraints. Env is ordered pointwise.

1. \mathcal{P} constructs the least fixpoint of a function in $\text{Env} \rightarrow \text{Env}$. In practice, an analyzer evaluates this fixpoint in a lazy manner, computing only the portion of e needed for the problem at hand (and using the abstract domain and operations).
2. \mathcal{C} has two cases:
 - (a) For a clause defining the predicate of the call, $p(\bar{i}, \bar{o})$, \mathcal{A} is called to simulate the agent A with demand δ , renamed appropriately. The third argument to \mathcal{A} , $\bar{\exists}_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \delta)$, renames the output demand, δ , in terms of the local output variables, \bar{y} . The result of simulating the clause body A is again renamed in terms of the calling-context input variables, \bar{i} . The demand collecting semantics decomposes (via equations) output demands until they become $\downarrow \text{true}$.
 - (b) A clause contributes to the semantics of only the predicate it defines.
3. \mathcal{A} has two cases:

- (a) A nontrivial output demand is projected onto the output variables of the agent, and \mathcal{B} is used to simulate the agent appropriately.
- (b) The trivial output demand (one with $\exists_{\text{Out}(A)}\delta = \downarrow \text{true}$) is satisfied by the trivial input demand. Correctness depends on detecting soon enough that the output demand has become $\downarrow \text{true}$, as this is what keeps the computed input demand from including more asks than necessary.

4. \mathcal{B} has five cases:

- (a) *Parallel composition*: A local *greatest* fixpoint is constructed here to allow for the possibility of cyclic dependence among the parallel goals. The iteration implied by this greatest fixpoint is not essential for correctness, but improves precision. The analyzer may safely over estimate the true demand². Because the function that is iterated is a decreasing function, the least fixpoint is actually the least element, which would not satisfy the safety condition. (Again, the safety condition requires that, if the agent is provided a constraint that is not contained in the computed input demand, then it cannot generate output that is contained in the given output demand.) Hence we take the greatest fixpoint.
- (b) *Predicate call*: In the denotation of a predicate call we perform the projection of the constraint pool onto the relevant output clause variables to obtain the output constraints. Then the clause environment, e , is consulted and the corresponding input demands are returned. The input variables for the inferred input demand are then renamed again. The result is intersected with the original local constraint pool, δ , reintroducing constraints on variables that are local to the calling context.
- (c) *Ask constraint*: The denotation of the choice operator corresponds to computing for a given output demand δ , the corresponding input demand for each branch, and then unioning those possible input demands.
- (d) *Tell constraint*: The tell rule just requires that the output demand be consistent with the actual output; otherwise, the entire local constraint pool, and thus the input demand, is $\downarrow \text{false}$.
- (e) *Stop*: The computation has completed and so it does nothing.

3.4. Correctness

In this subsection we prove the correctness of the demand collecting semantics. We begin with a small result needed below.

Lemma 3.1. Consider any agent A and program P . $\mathcal{A}[A](\mathcal{P}[P])$ is \cap -continuous. That is, for any decreasing chain $\{\delta_i\}_i$,

$$\mathcal{A}[A](\mathcal{P}[P])\left(\bigcap_{i \in \omega} \delta_i\right) = \bigcap_{i \in \omega} (\mathcal{A}[A](\mathcal{P}[P])\delta_i)$$

Proof: Follows from the \cap -continuity of \exists on $\wp\downarrow(C)$ (Proposition 2.3.3). \square

²Note that this over estimation is in terms of the size of the set of possible constraints, not in terms of the strength of those constraints.

$\mathcal{P} : \text{Declarations} \rightarrow \text{Env}$	$\text{Env} = \text{Atom} \rightarrow \wp \downarrow(C) \rightarrow \wp \downarrow(C)$
$\mathcal{C} : \text{Clause} \rightarrow \text{Env} \rightarrow \text{Env}$	$e : \text{Env}$
$\mathcal{A} : \text{Agents} \rightarrow \text{Env} \rightarrow \wp \downarrow(C) \rightarrow \wp \downarrow(C)$	
$\mathcal{B} : \text{Agents} \rightarrow \text{Env} \rightarrow \wp \downarrow(C) \rightarrow \wp \downarrow(C)$	$\delta, \delta' : \wp \downarrow(C)$
$\mathcal{P}[\mathcal{P}] = \text{lfp } \lambda e. \bigsqcup_{h(\bar{x}, \bar{y}) : -A \in \mathcal{P}} \mathcal{C}[h(\bar{x}, \bar{y}) : -A] e$	
$\mathcal{C}[h(\bar{x}, \bar{y}) : -A] e [p(\bar{i}, \bar{o})] \delta =$	
$\begin{cases} \exists_{\bar{i}}(\downarrow(\bar{x} = \bar{i}) \cap \mathcal{A}[A] e (\exists_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \delta))) & \text{if } h = p \\ \downarrow \text{false} & \text{otherwise} \end{cases}$	
$\mathcal{A}[A] e \delta = \begin{cases} \exists_{\text{In}(A)}(\mathcal{B}[A] e (\exists_{\text{Out}(A)} \delta)) & \text{if } \exists_{\text{Out}(A)} \delta \neq \downarrow \text{true} \\ \downarrow \text{true} & \text{otherwise} \end{cases}$	
$\mathcal{B}[A_1 \parallel A_2] e \delta = \text{gfp } \lambda \delta'. (\mathcal{A}[A_1] e \delta' \cap \mathcal{A}[A_2] e \delta' \cap \delta)$	
$\mathcal{B}[p(\bar{u}, \bar{v})] e \delta = \exists_{\bar{u}}(\downarrow(\bar{u} = \bar{i}) \cap e[p(\bar{i}, \bar{o})]) \exists_{\bar{v}}(\downarrow(\bar{v} = \bar{o}) \cap \delta)$	
$\mathcal{B}[\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i] e \delta = \cup_{i=1}^n (\downarrow c_i \cap \mathcal{A}[A_i] e \delta)$	
$\mathcal{B}[\text{tell}(c)] e \delta = \downarrow c \cap \delta$	
$\mathcal{B}[\text{Stop}] e \delta = \delta$	

FIGURE 3.1. The Demand Collecting Semantics. We assume that the variables \bar{i} and \bar{o} do not occur in programs and we define our environments over atoms of the form $p(\bar{i}, \bar{o})$. For an output demand given by $\delta \in \wp \downarrow(C)$, the input demand given by our demand collecting semantics is $\mathcal{A}[A](\mathcal{P}[\mathcal{P}])\delta$. A specification of the demand transformation analysis is obtained by replacing $\wp \downarrow(C)$ with an abstract domain and reinterpreting the operations accordingly, as discussed at the end of Section 3.1.

To prove correctness of the demand collecting semantics, we need to show that all partial derivations are safely reflected by \mathcal{P} . That proof will be by structural induction on *partial and-trees*, which we now introduce.

Definition 3.1. *PAT*, the set of partial and-trees, is constructed as follows:

$$\begin{aligned} \square &: \text{PAT} \\ \circ &: \text{PAT} \\ \text{tree} &: \text{Agents} \times \text{PAT} \times \text{PAT} \longrightarrow \text{PAT} \cup \\ &\quad \text{Agents} \times \text{PAT} \longrightarrow \text{PAT} \end{aligned}$$

A partial and-tree is used to organize the nondeterministic choices made in a given partial derivation, d , into a tree whose nested structure corresponds to that of the applications of \mathcal{B} in our semantics. \circ is the constructor of empty subtrees that represent unreduced agents. \square is the constructor of empty subtrees that represent fully evaluated agents.

The partial and-tree for an agent A , a program P and a given reduction sequence d is a tree given by $\Psi_d(A)$, as defined in Table 3.1.

$$\begin{aligned} \Psi_d &: \text{Agents} \longrightarrow \text{PAT} \\ A &: \text{Agents} \end{aligned}$$

\underline{A}	$\underline{\Psi_d(A)}$
A	$\text{tree}(A, \circ)$, if A is not reduced by d
$A_1 \parallel A_2$	$\text{tree}(A_1 \parallel A_2, \Psi_d(A_1), \Psi_d(A_2))$
$p(\bar{x}, \bar{y})$	$\text{tree}(p(\bar{x}, \bar{y}), \Psi_d(A))$
	$p(\bar{x}, \bar{y})$:- $A \in P$ is renamed as used for $p(\bar{x}, \bar{y})$ in d
$\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i$	$\text{tree}(\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i, \Psi_d(A_j))$
	A_j is the choice made in d
Stop	$\text{tree}(\text{Stop}, \square)$
tell (c)	$\text{tree}(\text{tell}(c), \square)$

TABLE 3.1. The partial and-tree, $\Psi_d(A)$. Given a program P , a partial derivation d , and an agent A , the partial and-tree $\Psi_d(A)$ is the structured value given by the table above. If partial derivation d does not reduce the agent A , then $\Psi_d(A) = \text{tree}(A, \circ)$. Otherwise, the value of $\Psi_d(A)$ depends on the form of A as shown above.

Proposition 3.1. For each partial derivation d and each agent A , the partial and-tree $\Psi_d(A)$ is well defined.

We assume that all agents in d except **Stop** are syntactically unique, adding extra variables if necessary to make this true. This allows the correspondence between agents in the reduction sequence and agents in the partial and-tree to be simple syntactic identity. (The correspondence of **Stop** agents is immaterial because **Stop** agents are not reduced.) We further assume that the variables of d are disjoint from the variables appearing in the program P . Similarly, in the main lemma and theorem below, the variables appearing in a partial derivation are renamed apart from those appearing in the demand collecting semantics.

The following lemma, whose proof can be found in the appendix, states that, if a partial derivation's final computed constraint, projected onto a given agent's output variables, is in the output demand for which the program is to be analyzed, then that final constraint, projected onto the input variables of the agent, will be in the input demand defined by the demand collecting semantics. Because our requirements for well moding imply that no constraint on an agent's input can be added by that agent or by its descendant agents, this implies, as a stated in the theorem below the lemma, that for the given agent to produce the demanded output, the environment must provide it with the demanded input. (See Theorem 3.1, below.)

Lemma 3.2. Consider any program D and any partial derivation d of the transition system T_D . Consider any agent A' appearing in d . Let ρ be a renaming satisfying $A' = \rho A$ for some A whose variables are disjoint from those of d . Let σ' be the final constraint pool of d . For brevity, we assume that $\text{In}(A) = \bar{u}$ and $\text{Out}(A) = \bar{v}$ and write $\rho\bar{u} = \bar{u}'$ and $\rho\bar{v} = \bar{v}'$. Then for all $\delta \in \wp\downarrow(C)$

$$\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}\delta \implies \bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \mathcal{A}[[A]](\mathcal{P}[[P]])\delta$$

The following theorem expresses the correctness of the demand collecting semantics. It says that for any output demand (projected onto the output variables of the initial agent A), the set of inputs which can produce an output as strong

as the demanded one (projected on the input variables of A) are included in the demanded input.

Theorem 3.1. For all $b \in C$,

$$\bar{\exists}_{\text{Out}(A)} b \in \delta \implies \{\bar{\exists}_{\text{In}(A)} \sigma \mid \sigma \in \mathcal{D}_P[A]b\} \subseteq \mathcal{A}[A](\mathcal{P}[P])\delta$$

Proof: Consider any initial agent A and any $\sigma \in \mathcal{D}_P[A]b$. By definition of \mathcal{D}_P , there exists a partial derivation d showing that $\langle A, \sigma \rangle \longrightarrow^* \langle B, c \rangle$, for some $c \leq b$. Let ρ be a renaming satisfying $A' = \rho A$ for some A' whose variables are disjoint from those of d and are not constrained by c nor (consequently) by any other constraint of d . (For brevity, we assume that $\text{In}(A) = \bar{u}$, $\text{Out}(A) = \bar{v}$, and $\text{vars}(A) = \bar{u}\bar{v} = \bar{u} \cup \bar{v}$. We write $\rho\bar{u} = \bar{u}'$, $\rho\bar{v} = \bar{v}'$, and $\rho(\bar{u}\bar{v}) = \bar{u}'\bar{v}'$. Also, without loss of generality, we assume that ρ is the identity substitution on $\text{vars}(d) - \text{vars}(A)$.)

Let $d' = \rho d$ be a partial derivation derived from d by applying ρ to each syntactic object and applying to each constraint the function $f(\phi) = \exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge \phi)$. (Note that f uses \exists , not $\bar{\exists}$.) It is easy to see that d' proves that $\langle A', f(\sigma) \rangle \longrightarrow^* \langle B', f(c) \rangle$, for some B' . Now, by monotonicity of \wedge and \exists , since we have $c \leq b$, it follows that $f(c) \leq f(b)$, that is $\exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge c) \leq \exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge b)$. Adding the same constraint and quantifying both sides gives us $\bar{\exists}_{\bar{v}'}(\bar{v} = \bar{v}' \wedge \exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge c)) \leq \bar{\exists}_{\bar{v}'}(\bar{v} = \bar{v}' \wedge \exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge b))$. But notice that $\bar{\exists}_{\bar{v}'}(\bar{v} = \bar{v}' \wedge \exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge b)) = \bar{\exists}_{\bar{v}'} b$, since b does not constrain $\bar{u}'\bar{v}'$, by assumption on ρ . Again writing $f(c)$ for $\exists_{\bar{u}'\bar{v}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge c)$, we now have $\bar{\exists}_{\bar{v}'}(\bar{v} = \bar{v}' \wedge f(c)) \leq \bar{\exists}_{\bar{v}'} b$.

By hypothesis on δ , we have $\bar{\exists}_{\bar{v}'} b \in \delta$. Together with the previous inequality and the downwards closure of δ , this gives us $\bar{\exists}_{\bar{v}'}(\bar{v} = \bar{v}' \wedge f(c)) \in \delta$ and hence $\bar{\exists}_{\bar{v}'}(\bar{v} = \bar{v}' \wedge f(c)) \in \bar{\exists}_{\bar{v}'} \delta$.

By using Lemma 3.2 we now obtain $\bar{\exists}_{\bar{u}'}(\bar{u} = \bar{u}' \wedge f(c)) \in \mathcal{A}[A](\mathcal{P}[P])\delta$. Expanding $f(c)$ once again and recalling that $\bar{u}'\bar{v}'$ and \bar{u}' are unconstrained by c , we have

$$\begin{aligned} & \bar{\exists}_{\bar{u}'}(\bar{u} = \bar{u}' \wedge f(c)) \\ &= \bar{\exists}_{\bar{u}'}(\bar{u} = \bar{u}' \wedge \exists_{\bar{u}'\bar{v}'}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge c)) \\ &= \bar{\exists}_{\bar{u}'} \bar{\exists}_{\bar{v}'}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge c) \\ &= \bar{\exists}_{\bar{u}'}(\bar{u}\bar{v} = \bar{u}'\bar{v}' \wedge c) \\ &= \bar{\exists}_{\bar{u}'}(c) \end{aligned}$$

Moreover, property (2) of the mode system gives us $\bar{\exists}_{\bar{u}'} c = \bar{\exists}_{\bar{u}'} \sigma$. Thus, we have that

$$\bar{\exists}_{\bar{u}'}(\sigma) = \bar{\exists}_{\bar{u}'}(c) = \bar{\exists}_{\bar{u}'}(\bar{u} = \bar{u}' \wedge f(c)) \in \mathcal{A}[A](\mathcal{P}[P])\delta$$

as required. \square

4. Implementation

A demand transformation analysis has been implemented in Prolog and is available via the World Wide Web³. The implementation is based on a specification obtained by reinterpreting our demand collecting semantics (Figure 3.1) over the domain discussed in Section 3.1. The operations of unification, join (least upper bound),

³via the URL <http://www.transarc.com/~winsboro/Strictness>

and less than are based on those described in [23]. Termination is ensured by using widening as described in [19].

Input programs are written in ask/tell syntax and must have mode declarations. Arbitrary output demands can be analyzed. Since we are working with equations, simplifying the abstract constraints is equivalent to performing abstract unification. The analyzer begins by reading in the subject program and then precomputing initial clause environments (δ) for each clause, by simulating the clause's ask and tell constraints. Since aliasing is introduced only by the asks and tells, not by call simulation, this precomputation simplifies unification; simulating the asks and tells first allows us to handle all aliasing before any union or recursive types are introduced. In addition, it is helpful to execute tells before other calls, as the tells generally relate output demands on the clause to output demands on the body calls.

After the one-time initialization of the clause environments, the analyzer evaluates (lazily) the outer fixpoint in \mathcal{P} by computing the portion of the function e that is needed to find an input demand for the initial output demand. The function e is represented as a table of triples. When e is applied to a call/output demand pair not occurring in the table, a new triple with the call and output demand is initialized with an input demand of \perp and added to the table. To simulate a clause, the analyzer unifies the output demand with the precomputed initial environment for the clause, and then iteratively simulates clause bodies right to left, based on the heuristic that many data dependencies run left to right. This computes the fixpoint involving f in \mathcal{B} in the parallel composition case.

As a modest optimization of the outer fixpoint calculation, each time the analyzer considers a given output demand in the table, clause simulation is applied to it iteratively until local convergence occurs. This improves speed and precision for predicates that call themselves. The improved precision results from the interaction between iteration and widening.

5. Example analysis

We analyze the output demand formalized by the abstract constraint $\{Y = ?\}$ on `mergesort(X,Y)` (see Figure 5.1)⁴. We simplify the notation of the output demand, as in the following initial abstract environment.

$$e^{(0)} = \left\langle \begin{array}{cc} & \text{output input} \\ \text{mergesort} & ? \quad \perp \end{array} \right\rangle$$

The analysis begins its fixpoint calculation at this bottom abstract environment. Theoretically, the bottom environment maps all output demands to the input demand \perp . However, the analysis computes the environment in a lazy manner, constructing input demands only for output demands in the table, and entering an output demand in the table only when its corresponding input demand is needed for the computation. The analysis considers each output demand recorded in $e^{(0)}$ (successive clause environments are denoted with superscripted numbers). It computes an approximation of the input needed to generate that output. To this end, the analysis simulates the agent that defines the predicate. The analysis begins by simulating the behavior of `mergesort/2`. Each 'ask' summand is simulated and the

⁴Please note that we abbreviate the names of the predicates in Figure 5.1

$$\begin{aligned}
ms(U, S) & :- (\mathbf{ask}(U = []) \rightarrow \mathbf{tell}(S = [])) + \\
& \quad \mathbf{ask}(U = [H]) \rightarrow \mathbf{tell}(S = [H]) + \\
& \quad \mathbf{ask}(U = [H1, H2|UT]) \rightarrow \\
spl(U, U1, U2) & :- (\mathbf{ask}(U = []) \rightarrow \mathbf{tell}(U1 = [], U2 = [])) + \\
& \quad \mathbf{ask}(U = [H]) \rightarrow \mathbf{tell}(U1 = [H], U2 = []) + \\
& \quad \mathbf{ask}(U = [H1, H2|Hs]) \rightarrow \\
& \quad \quad \mathbf{tell}(U1 = [H1|H1s], U2 = [H2|H2s]) \parallel spl(Hs, H1s, H2s) \\
me(S1, S2, S) & :- (\mathbf{ask}(S1 = [H1|T1], S2 = [H2|T2], H1 \leq H2) \rightarrow \\
& \quad \mathbf{tell}(S = [H1|T]) \parallel me(T1, S2, T) + \\
& \quad \mathbf{ask}(S1 = [H1|T1], S2 = [H2|T2], H1 > H2) \rightarrow \\
& \quad \mathbf{tell}(S = [H2|T]) \parallel me(S1, T2, T) + \\
& \quad \mathbf{ask}(S1 = []) \rightarrow \mathbf{tell}(S = S2) + \\
& \quad \mathbf{ask}(S2 = []) \rightarrow \mathbf{tell}(S = S1))
\end{aligned}$$

	output	input		output	input		output	input
ms/2	2	1	spl/3	2,3	1	me/3	3	1,2

FIGURE 5.1. Mergesort (and predicates' modes)

results are joined. Let us consider what happens with each of the summands, which we refer to as `mergesort/2/1`, `mergesort/2/2`, and `mergesort/2/3`, respectively. In each case, we begin by simulating the ask as well as any tells in the body, as these do not have to be iterated.

Because the local fixpoint is a *greatest* fixpoint, all variables are unconstrained when the analysis starts simulating a predicate definition. For `mergesort/2/1`, we have to simulate the unification of the asks and tells ($U = [], S = []$) in the presence of $\delta = \{U = \top, S = ?\}$, which is obtained from the output demand. This leads to the local type environment

mergesort/2/1: $\delta = \{U = [], S = []\}$

Iterating a tell never changes δ , as can be seen in the tell rule in \mathcal{B} in the semantics. So the analyzer is done with this summand, having inferred that an input of type $[]$ might be sufficient to produce the demanded output. Similarly we have:

mergesort/2/2: $\delta = \{U = [H], S = [H], H = \top\}$

On entry to the simulation of `mergesort/2/3`, we obtain the following local type environment, where the types for S and U come from the demand and the ask, respectively:

mergesort/2/3: $\delta = \{U = [H1, H2|UT], H1 = \top, H2 = \top, UT = \top, S = ?\}$

The body of `mergesort/2/3` is a parallel composition of four calls. These calls are simulated by looking up the demands in the clause environment with applications like $e^{(0)}[\mathbf{split}(U, U1, U2)]\{U1 = \top, U2 = \top\}$. The lookups of `split/3` and `mergesort/2` all return \top , illustrating the second case of \mathcal{C} . The application $e^{(0)}[\mathbf{merge}(S1, S2, S)]\{S = ?\}$ returns \perp and has the side-effect of recording this non-trivial demand in $e^{(1)}$. The input demands from each `mergesort/2` summand are joined (lub) in the ask rule in \mathcal{B} : $[] \sqcup [\top] \sqcup \perp = [] \vee [\top]$. Further local

iteration makes no further change. Our new environment becomes:

$$e^{(1)} = \left\langle \begin{array}{ccc} & \text{output} & \text{input} \\ \text{mergesort} & ? & [] \vee [\top] \\ \text{merge} & ? & \perp \end{array} \right\rangle$$

Since $e^{(0)} \neq e^{(1)}$, the analyzer iterates over the demands in $e^{(1)}$. Nothing new is learned about **mergesort**/2, but inferences are made about **merge**/3. In **merge**/3/1, after simulating the ask and tell, we have

$$\text{merge/3/1: } \delta = \left\{ \begin{array}{l} \text{H1} = \text{num}, \text{ H2} = \text{num}, \quad \text{T1} = \top, \quad \text{T2} = \top, \\ \text{T} = \top, \quad \text{S1} = [\text{H1}|\text{T1}], \text{ S2} = [\text{H2}|\text{T2}], \text{ S} = [\text{H1}|\text{T}] \end{array} \right\}$$

This leads the analyzer to look up $e^{(1)}[\text{merge}(\text{T1}, \text{S2}, \text{T})](\text{T} = \top)$ which yields $(\text{T1} = \top \wedge \text{S2} = \top)$ by the second case of \mathcal{C} . Thus, analysis of the summand **merge**/3/1 yields $\langle [\text{num}|\top], [\text{num}|\top] \rangle$ as the demanded input. The ask rule in \mathcal{B} joins input demands from the four summands for **merge**/3:

merge/3: $\langle [\text{num}|\top], [\text{num}|\top] \rangle \sqcup \langle [\text{num}|\top], [\text{num}|\top] \rangle \sqcup \langle [], ? \rangle \sqcup \langle ?, [] \rangle = \langle ?, ? \rangle$ This says that to produce some output it may be sufficient to give **merge**/3 the weakest non-trivial input on each argument. Notice that this is a slight approximation (weakening) of the truly sufficient input. We now have

$$e^{(2)} = \left\langle \begin{array}{ccc} & \text{output} & \text{input} \\ \text{mergesort} & ? & [] \vee [\top] \\ \text{merge} & ? & \langle ?, ? \rangle \end{array} \right\rangle$$

In each of the next few iterations, only the changes are shown.

$$\begin{array}{l} e^{(3)} \text{ changes: } \text{split} \quad \begin{array}{cc} \text{output} & \text{input} \\ \langle [] \vee [\top], [] \vee [\top] \rangle & \perp \end{array} \\ e^{(4)} \text{ changes: } \text{split} \quad \begin{array}{cc} \langle [] \vee [\top], [] \vee [\top] \rangle & [] \vee [\top] \\ \text{split} & \langle [], [] \rangle \\ & \perp \end{array} \\ e^{(5)} \text{ changes: } \text{split} \quad \begin{array}{cc} \langle [], [] \rangle & [] \end{array} \\ e^{(6)} \text{ changes: } \text{split} \quad \langle [] \vee [\top], [] \vee [\top] \rangle \mu\tau.[] \vee [\top|\tau] \end{array}$$

$\mu\tau.[] \vee [\top|\tau]$ has been substituted for $[] \vee [\top] \vee [\top, \top]$ by widening. When this new inference is used in **mergesort**/2/3, the input demand propagates.

$$e^{(7)} \text{ changes: } \quad \begin{array}{ccc} & \text{output} & \text{input} \\ \text{mergesort} & ? & \mu\tau.[] \vee [\top|\tau] \end{array}$$

When this inference is used in the recursive calls in **mergesort**/2/3, we get a new output demand on **split**/3.

$$e^{(8)} \text{ changes: } \quad \begin{array}{ccc} & \text{output} & \text{input} \\ \text{split} & \mu\tau.[] \vee [\top|\tau] & \perp \end{array}$$

When that output demand is considered, we obtain the final environment:

$$e^{(9)}: \quad \begin{array}{ccc} & \text{output} & \text{input} \\ \text{mergesort} & ? & \mu\tau.[] \vee [\top|\tau] \\ \text{merge} & ? & \langle ?, ? \rangle \\ \text{split} & \langle [] \vee [\top], [] \vee [\top] \rangle & \mu\tau.[] \vee [\top|\tau] \\ \text{split} & \langle [], [] \rangle & [] \\ \text{split} & \mu\tau.[] \vee [\top|\tau] & \mu\tau.[] \vee [\top|\tau] \end{array}$$

Predicate	Output Demand	Input Demand
reverse	?	$list(\top)$
mergesort	?	$list(\top)$
quicksort	?	$\langle [] \vee [\top \mid list(num)], \top \rangle$
tree count	?	$tree(\top)$
sum tree	?	$tree(num)$
sort server	$[? \mid \top]$	$[[[] \vee [\top \mid list(num)] \mid \top]$
sort server	?	\top
Gaussian elimination	?	\top
Gaussian elimination	$list(\top)$	$[\top \mid list([num \mid \top])]$
matrix transpose	?	\top
matrix transpose	$[list(\top) \mid \top]$	$list([\top \mid \top])$
cyclic dependence	?	$\langle [a \mid \top], \top \rangle$
cyclic dependence	$[b, b, b \mid \top]$	$\langle [a, a, a \mid \top], [a \mid \top] \rangle$
cyclic dependence	$list(\top)$	$\langle \perp, \perp \rangle$

TABLE 6.1. Analyzer Results: Input Demands as a Function of Output Demands

The analyzer has inferred that unless `mergesort/2` is given a nil-terminated list, it cannot produce output.

6. Results

Table 6.1 shows the input demands⁵ inferred by the analyzer⁶ for a variety of different output demands on several small programs. Note that the two occurrences of τ in the `tree` constructor need not be instantiated in the same way. Therefore, this type describes all possible, finite trees, not just balanced, finite trees. The table focuses on the output/input behavior of the processes. Demand reachability is illustrated by the analysis of `mergesort` in Section 5.

The first five benchmarks were analyzed with the weakest non-trivial output demand, $?$. Thus the table tells us that `reverse` and `mergesort` both require complete nil-terminated lists before they provide any output at all. The `quicksort` predicate uses the technique of difference lists, so it has two input arguments. The second of these (the accumulator) has no input demand. The first must be a nil-terminated list, all of whose elements, besides the first, must be numbers. The analyzer cannot infer that the first element must also be a number, because a list with one element does not need to be inspected to be sorted.

The two benchmarks involving trees help to demonstrate the expressiveness of our domain. The `tree count` benchmark counts the number of nodes in a tree. As expected, it indicates that to get any output, a finite tree (a tree that is left and right strict) is needed, but the elements in the nodes need not be instantiated. In contrast `sum tree` requires that the tree must be finite *and* the elements in the nodes must be `nums`.

⁵ $list(d)$ stands for $\mu\tau.([\] \vee [d \mid \tau])$, for arbitrary type terms $d \in \mathcal{D}$. Also, we simplify $\mu\tau.(leaf \vee tree(d, \tau, \tau))$ as $tree(d)$, for arbitrary type terms $d \in \mathcal{D}$.

⁶Available via the URL <http://www.transarc.com/~winsboro/Strictness/>

The `sort server` process takes an infinite list whose elements are finite lists of numbers, which it sorts by using `quicksort`, placing them on an infinite output list. It illustrates the potential impact on the communication granularity of finite subprocesses in the context of infinite processes. The computed input demand indicates that it is safe to wait for one completely instantiated list of numbers before starting the `quicksort` subprocess. This is relevant, for example, because it would allow the system to establish a size threshold on the input that would justify remote execution.

The last three benchmarks are programs that begin producing output with minimal input. The role of the analyzer with such processes is to propagate stronger output demands that may be imposed by their consumers or that might be of interest for compile-time scheduling and load distribution.

The input and output of `Gaussian elimination` are matrices, represented as a list of lists. The output demand asks for nil-terminated rows whose elements can be unconstrained. The computed input demand indicates that the list of rows must be nil-terminated, with each row besides the first required to have a number in its first column.

The output demand for `matrix transpose` demands the entire first row of the matrix with possibly unconstrained elements. The inferred input demand requires the entire first column of the input.

The `cyclic dependence` benchmark illustrates the need for the local fixpoint involving f in \mathcal{B} . It constructs a pair of infinite, co-routining processes that communicate back and forth. Because the processes are infinite (they have no base cases), the lists that they operate on are not nil-terminated. For the output demand that requires a partially instantiated list containing three b 's, the computed input demand indicates that at least three a 's are needed in the first argument and at least one a in the second argument. For the output demand that requires a nil-terminated list, the input demand expresses the impossibility of obtaining that output.

7. Related work

Analysis of concurrent constraint languages: Previous works on analysis of concurrent constraint languages include [4, 3, 5, 11, 27, 36]. [4, 3, 5] study a *suspension* analysis for a goal (or an agent) in a given program. If the analysis infers that a goal will not suspend, then non-suspension is guaranteed for any runtime scheduling rule. The analysis also can determine at compile time a fair scheduling of the goal. As such, it is likely to form a reasonable basis for compile-time scheduling, as discussed in the introduction.

Debray, Gudeman and Bigot [11] present an analysis that identifies concurrent logic programs that can be executed according to a depth-first, left-to-right scheduling rule without danger of suspension. When the compiler finds this common scheduling to be safe, several optimizations, which they present, are enabled. One component of their method, which they call *weak suspension analysis*, might be called a *sufficient* demand analysis. It identifies sufficient input to a goal that will allow the goal to execute to completion without suspending. Thus, the sense of this analysis's safety requirement is dual to our own.

In another paper [10] Debray presents an analysis which he calls a demand analysis for the concurrent constraint language, QD-Janus. In contrast with our,

this is a *local* demand analysis that considers only the guards of each clause to determine how much input is necessary to enable a goal to reduce. This analysis identifies input demands necessary to enable goal reduction, but says nothing about the input demands necessary to produce output. As such, it affords little assistance in enabling transformations that risk introducing deadlock, and hence changing the declarative meaning of predicates.

Zaffanella et al. [36] conduct a theoretical investigation of analysis of suspension-free concurrent constraint programs. They provide two transformations of suspension-free CCP programs that allow standard constraint-logic-program dataflow analysis techniques to be applied to the transformed programs.

King and Soper [27] present an analysis for concurrent logic programs that allows a division of processes into threads, within which execution is sequential. This work is related to load distribution and the possibilities of reciprocal benefits should be exploited.

Context analysis of functional programs: Although forward analyses have probably been more widely studied, backward flow analyses have received considerable attention in the literature on functional programming. A framework for backward analysis was defined by Cousot and Cousot in [7]. The same authors also extensively studied backward and forward frameworks for logic programs in [8].

In [22] it is shown that it is possible to ‘invert’ abstract functions and thus to pass from a forward to a backward analysis and vice versa. The loss of information in reversing the analysis depends on properties of the domain considered. Galois connections give the best results. As future work, it would be interesting to study whether it would be possible to use (and if necessary extend) the framework of [22] to model our analysis.

A pioneering paper on backward strictness analyses is the one by Hughes [20] in which he presents an analysis for a lazy functional language, based on a denotational semantics given by means of continuations. This paper introduces the notion of context, an operator ($\&$) which allows one to combine the information given by two different contexts. One important element in the domain is called *ABS* (for absent), which models the fact that a given expression is definitely not used in the given context. These concepts are simplified, by eliminating the use of continuations, and clarified in [21]. This paper presents a more general framework for backward analyses and shows several examples, including applications to strictness and to memory garbage-collection.

About the same time as Hughes introduced the notion of a context analysis in [20], Wadler and he also introduced it in [35]. By utilizing the context in which function applications occur to justify eager evaluation of function arguments, context analyses give stronger results than standard strictness analysis. The demand transformation analysis we have constructed seems similar to this notion.

Context analyses have been used to optimize both sequential and parallel functional languages. In parallel functional programming (see for instance [1]), context analysis is used to identify arguments that can be evaluated in parallel with the function call without risking wasted effort. Our application of context analysis is different. We want to increase grain size by introducing synchronization constraints delaying the execution of the call until sufficient evaluation has been performed on its producers. In this application, the principal danger is not wasted effort so much as the introduction of deadlock. Indeed, in CCP languages where inconsistent “tel-

l” constraints result in failure, there appears to be no danger of wasted work. In this context, all subgoals must be reduced to determine whether they cause the computation to fail.

Wadler and Hughes’s projection analysis. The context analysis of Wadler and Hughes [35], subsequently also studied by Davis and Wadler [9], describes a context by using the domain-theoretic concept of a projection, which in general is just a decreasing, idempotent function. Projections are particularly powerful for this application as they can capture dependencies between data demands, such as head strictness, the property that says roughly, each cons cell that is inspected will have its head inspected, too. They propose a ten point domain of projections that capture various extents of strictness on lazy lists. Our demand transformation analysis can capture several of the same context properties. Their *FAIL* appears to be equivalent to our \perp ; *STR* corresponds to $?$; *ID* corresponds to \top . However, we are not able to identify definitely ignored input, which they capture by the projection *ABS*. If in our system \top meant *definitely* unconstrained, we too could identify unneeded input. However, because our demand collecting semantics is based on downwards-closed sets of constraints, \top means *possibly* unconstrained in all candidate abstract domains. This can be construed as a significant limitation in our approach. For instance, it prevents us from capturing a demand for an open list (a list terminated by a free variable rather than a $[\]$) with non-var elements. This is precisely the notion of head strictness captured by the projection H' . However, the limitation is a fundamental consequence of the safety requirement, and hence the information ordering, that organize the design of our analysis.

It is worth mentioning that by using the abstract domain presented here, our analyzer *is* able to capture the notions of tail strictness and simultaneous head and tail strictness (projections T' and $H' \sqcap T'$). Further research is needed to investigate the potential utility of these various forms of strictness to applications of the analysis results.

Unlike the analysis of Wadler and Hughes (at least with the domain they use in [35]), our analysis can express strictness information in *any* recursive datatype. For instance, while Wadler and Hughes’s analysis can only tell that an argument is strict (*STR* for non-lists), we can say to what extent the argument is strict, for instance that a tree constructor is left strict or that it is left and right strict and also requires that the elements in the leaves be integers.

Demandedness analysis for functional logic languages. In [30], the authors present a strictness analysis for languages with lazy narrowing. Their method uses algebraic simplification of what are essentially type equations. They do not claim to have an implementation. Their analysis is quite similar to strictness analyses for functional languages, but the semantic notion of safety is slightly different. They note that whereas a failing subcomputation causes the enclosing computation to fail in functional languages, in functional logic languages, this is not the case. Their analysis uses a domain similar to ours, but theirs does not propagate demands (i.e. is not a context analysis) nor are they working in a concurrent setting in which cyclic dependence is a problem. In addition, they hope to optimize laziness in the languages they consider by identifying arguments that may be safely evaluated eagerly. However, their analysis is able to capture a notion of joint strictness which we didn’t consider. For instance, they can infer that a function which takes two lists and returns true if their length is equal and false otherwise requires two nil-terminated lists of equal length whereas we only infer that a similar predicate would

require two nil-terminated lists.

Jensen and Mogensen’s backward analysis. The compile-time garbage collection of Jensen and Mogensen [24] for a simple first order functional language is defined as a backward analysis based on the notion of contexts and having a context combinator, $\&$, and a projection, ABS , similar to those defined in [20] and [35]. Their analysis allows them to define a notion of approximate counters for references to list-like data structures (S-expressions). When their analysis derives that a data structure has been referred to at most once, such data structure can be reused via in-place updating.

Jensen and Mogensen start by presenting a domain that allows them to distinguish base-type values from structures and to count the number of references to each structure. By a construction similar to ours, they then lift this initial domain to the Hoare powerdomain consisting of sets of elements of the initial domain. As with the ABS projection of Wadler and Hughes [35] discussed above, it is not possible to identify definitely ignored input. The relationship with our domain is also similar.

The authors do not give a formal semantics for their domain. Since functional programming does not allow free variables in data values, their domain does not include variables. The domain used in [24] also does not characterize recursive types. However, the paper does discuss how their analysis can be extended to higher order functions. They do not fix a specific computation rule, so their analysis applies to languages which use either call-by-value or call-by-name parameter passing.

It is important to note that the sense of the approximations for garbage collection and for strictness are opposite. For garbage collection ABS means definitely not needed and for strictness it means possibly not needed. So, here ABS cannot correspond to top (don’t know) as it can in the case of strictness analysis.

Jones and Le Metayer’s backward analysis. Jones and Le Metayer [25] develop a compile time sharing analysis for garbage collection based on abstract interpretation. They consider a simple domain of patterns given by an infinite set of binary trees, where the internal nodes are labeled by cons operators and leaves are labeled by 0’s and 1’s, where a ‘0’ represents the fact that the associated substructure is not shared, while a ‘1’ means “possibly shared”. Their analysis characterizes whether a given substructure is possibly shared, i.e., possibly referred to more than once. They do not use reference counters and define two non-standard semantic functions which allow them to characterize, given a certain output subexpression, the input parts in which that subexpression appears, by means of a backwards, context analysis. They do not have an explicit ABS element in their domain, but rather they capture ‘absence’ by means of one of these non-standard functions. They discuss application of their analysis to strictness in a different paper. Their analysis does not use recursive types, although they believe [25] types could provide for a useful extension of their domain. Similarly to [24], they refer to a first order untyped functional language, but they also assume that the language computation rule is call by value.

Hall and Wise’s stream-strictness analysis. Hall and Wise [16, 17] use an infinite domain for their analysis and use algebraic manipulation to get their results, which are in the form of a lazy language with annotated data structures. Like our analysis, that of Hall and Wise is unable to capture the notion of absence (ABS in Wadler and Hughes’s projection analysis). Both analyses infer more detailed information about lists than other context analyses. For instance, both analyses can infer that

a list is strict in every tail and strict in every other head. However, Hall and Wise’s analysis is able to detect head strictness, which ours is unable to detect. In addition, Hall and Wise have focused their work on practical aspects of building a compiler and utilizing their analysis, but they also seem to narrow their focus on streams give no indication that they are able to analyze data structures other than lists.

Burn’s evaluation-transformer analysis [1] achieve results similar to those of Wadler and Hughes’s Projection Analysis. Like our demand transformation analysis, it is unable to capture the notions of absence and head strictness that Wadler and Hughes capture. However, it is also unable to capture the notions of absence and failure (*ABS* and *FAIL*). Furthermore, although Burn claims that the analysis may be used with more descriptive domains for recursive data structures, he only demonstrates evaluators which correspond to the points in Wadler’s four point domain [34].

Dybjer’s inverse image analysis. Peter Dybjer [14], gives a very theoretical treatment based on *open set expressions*, which denote upwards closed (Scott open) sets of partial data structures. It turns out that his analysis is also unable to detect absence (*ABS*) and head strictness (H'), but his analysis uses algebraic manipulation to solve equations and in his conclusion he writes “it is clear from the examples shown above that [his technique] would only succeed in simple cases unless clever symbol manipulation ideas are introduced.”

Lindstrom’s backward strictness analysis. Lindstrom develops a strictness analysis in [28] that seems related to our own. He also considers function reversal and a mapping of output demands to input demands. Like our analysis, his is unable to recognize internal strictness patterns like head strictness. However, his domain is limited to only a few kinds of strictness similar to those in Wadler’s four point domain [34]. Most interesting, perhaps, is that we seem to have unwittingly carried out the work that he proposes in his future work section. He notes that cyclic functions are useful and should be dealt with. Our local fixpoint calculation addresses exactly this problem. He also notes that “...more realistic data structures must be considered, including tuples of length greater than two...,” and we have incorporated this as well. He then indicates that “Application of strictness to logic programming is very appealing...” We claim that application as one of our contributions. Finally, he calls for “Extension of the domain to represent more detailed data type information...” and certainly our domain captures detailed data type information. The one aspect of his future work that we haven’t attempted to solve is the problem we cited above concerning the recognition of head strictness.

8. Contributions and future work

We have presented a demand transformation analysis that maps a predicate’s output demands to its input demands. This is the first formal construction of such an analysis for concurrent constraint programming. We have identified several optimizations of distributed implementations that are enabled by the analysis, addressing such issues as granularity control, compile-time scheduling, and load distribution. We have implemented the analysis in Prolog. The code, along with the programs in table 6.1, is available via the World Wide Web. Our evaluation of the implementation shows that useful information is inferred in practice. Our implementation is based on an infinite domain that accommodates arbitrary recursive

types. This is in contrast with the domains used in similar analyses in functional programming. The construction of the domain itself may be considered a modest contribution due to its elegant expression of depth-k sharing. As a matter of future work, it would be useful to handle open lists and head strictness. This seems to require giving up downwards closure of our domain in order to capture definite freeness.

APPENDIX

Lemma 3.2. Consider any program D and any partial derivation d of the transition system T_D . Consider any agent A' appearing in d . Let ρ be a renaming satisfying $A' = \rho A$ for some A whose variables are disjoint from those of d . Let σ' be the final constraint pool of d . For brevity, we assume that $\text{In}(A) = \bar{u}$ and $\text{Out}(A) = \bar{v}$ and write $\rho\bar{u} = \bar{u}'$ and $\rho\bar{v} = \bar{v}'$. Then for all $\delta \in \wp\downarrow(C)$

$$\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}\delta \implies \bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \mathcal{A}[[A]](\mathcal{P}[[P]])\delta$$

PROOF. The proof is by induction on the structure of $\Psi_d(A')$. In the basis, d does not reduce A' . By well moding it follows that $\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') = \text{true}$. Together with the assumption on δ , this implies that $\bar{\exists}_{\bar{v}}\delta = \downarrow \text{true}$. From the definition of \mathcal{A} it follows that $\mathcal{A}[[A]](\mathcal{P}[[P]])\delta = \downarrow \text{true}$. So the required containment holds trivially. For the induction step, we proceed with a case analysis of A .

$A = A_1 \parallel A_2$ Extending our conventions, we assume $\rho A_1 = A'_1$ and $\rho A_2 = A'_2$ and that $\text{In}(A_1) = u_1$, $\text{Out}(A_1) = v_1$, $\text{In}(A_2) = u_2$, and $\text{Out}(A_2) = v_2$. Also for notational convenience, let

$$F(\delta') = \mathcal{A}[[A_1]]e\delta' \cap \mathcal{A}[[A_2]]e\delta' \cap \bar{\exists}_{\bar{v}}\delta$$

Define the Kleene sequence of F as follows: $F^0 = \downarrow \text{true}$; $F^{i+1} = F(F^i)$. By using induction on i , we will show that for all i

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in F^i$$

basis $F^0 = \downarrow \text{true}$. So the desired containment is trivial:

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in F^0$$

step Fix arbitrary $i \geq 0$. Rewriting F^{i+1} in terms of F^i , the proof obligation becomes

$$\begin{aligned} \bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \\ \in \mathcal{A}[[A_1]](\mathcal{P}[[P]])F^i \cap \mathcal{A}[[A_2]](\mathcal{P}[[P]])F^i \cap \bar{\exists}_{\bar{v}}\delta \end{aligned} \tag{A.1}$$

From the induction assumption on i , we have

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in F^i$$

Projecting both sides of that containment we obtain

$$\bar{\exists}_{\bar{v}_1}(\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma')) \in \bar{\exists}_{\bar{v}_1}F^i$$

which simplifies to

$$\bar{\exists}_{\bar{v}_1}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}_1}F^i$$

The variables in \bar{u} do not occur in either σ' or in \bar{v}' . Moreover, the variables of \bar{u} do not appear in \bar{v}_1 and hence are quantified here. So the constraint $\bar{u} = \bar{u}'$ can be dropped by 2.1. By well moding, $\bar{v} \subseteq \bar{v}_1 \cup \bar{v}_2$. So changing $\bar{v} = \bar{v}'$ to $\bar{v}_1 = \bar{v}'_1 \wedge \bar{v}_2 = \bar{v}'_2$ is justified by downwards closure of $\bar{\exists}_{\bar{v}_1} F^i$. Rewriting in this way gives us

$$\bar{\exists}_{\bar{v}_1}(\bar{v}_1 = \bar{v}'_1 \wedge \bar{v}_2 = \bar{v}'_2 \wedge \sigma') \in \bar{\exists}_{\bar{v}_1} F^i \quad (\text{A.2})$$

However, because the variables of \bar{v}_2 do not appear elsewhere in the conjunction and because well-modding gives us $\bar{v}'_1 \cap \bar{v}'_2 = \emptyset$, by Propostion 2.1 we can drop $\bar{v}_2 = \bar{v}'_2$, and obtain

$$\bar{\exists}_{\bar{v}_1}(\bar{v}_1 = \bar{v}'_1 \wedge \sigma') \in \bar{\exists}_{\bar{v}_1} F^i \quad (\text{A.3})$$

which is the antecedent of the induction assumption of the structural induction on $\Psi_d(A')$. Thus we may derive the consequent:

$$\bar{\exists}_{\bar{u}_1}(\bar{u}_1 = \bar{u}'_1 \wedge \sigma') \in \mathcal{A}[[A_1]](\mathcal{P}[[P]])F^i$$

By removing some existentials and conjoining a new equation, we strengthen the lefthand side. Since $\mathcal{A}[[A_1]](\mathcal{P}[[P]])F^i$ is downwards closed, it follows that

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \mathcal{A}[[A_1]](\mathcal{P}[[P]])F^i$$

By a similar argument we can show that

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \mathcal{A}[[A_2]](\mathcal{P}[[P]])F^i$$

Finally, by assumption on δ we have

$$\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}(\delta)$$

and because $\bar{\exists}_{\bar{v}}(\delta)$ is downwards closed this gives us

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}(\delta)$$

Together, these three containments give us (A.1), as desired.

It now follows that

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \bigcap_{i \in \omega} F^i$$

By Lemma 3.1, \mathcal{A} is \cap -continuous and thus, since intersection preserves \cap -continuity, F is \cap -continuous. Thus $\text{gfp } F = \bigcap_{i \in \omega} F^i$ and

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \text{gfp } F$$

Projecting both sides, we have

$$\bar{\exists}_{\bar{u}}(\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma')) \in \bar{\exists}_{\bar{u}}(\text{gfp } F)$$

or simply, by using Propostion 2.1,

$$\bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \bar{\exists}_{\bar{u}}(\text{gfp } F)$$

that is to say,

$$\bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \mathcal{A}[[A_1 || A_2]](\mathcal{P}[[P]])\delta$$

as desired.

$A = p(\bar{u}, \bar{v})$ We assume that $\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}\delta$ and show $\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \mathcal{A}[[p(\bar{u}, \bar{v})]](\mathcal{P}[[P]])\delta$. Assume that $p(\bar{x}, \bar{y}) :- A_0$ is the definition of p in program P and that $\rho_0(p(\bar{x}, \bar{y}) :- A_0)$ is the variant of that definition that is used in d to reduce $p(\bar{u}', \bar{v}')$. Extending our naming convention, we write $p(\bar{x}', \bar{y}') :- A'_0$ for $\rho_0(p(\bar{x}, \bar{y}) :- A_0)$. Thus, \bar{x}' and \bar{y}' are identical to \bar{u}' and \bar{v}' , respectively. We assume that $\text{In}(A_0) = \bar{u}_0$ and $\text{Out}(A_0) = \bar{v}_0$ and write \bar{u}'_0 for $\rho_0(\bar{u}_0)$ and \bar{v}'_0 for $\rho_0(\bar{v}_0)$. Thus by well moding, $\bar{x} \subseteq \bar{u}_0$, $\bar{y} \subseteq \bar{v}_0$, $\bar{x}' \subseteq \bar{u}'_0$, and $\bar{y}' \subseteq \bar{v}'_0$.

Rewriting by using the definitions in Figure 3.1, using the $c \in P$ defining $p(\bar{x}', \bar{y}')$, and by unfolding the least fixpoint by one application, we must show

$$\begin{aligned} & \bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \\ & \in \bar{\exists}_{\bar{u}}(\downarrow(\bar{u} = \bar{i}) \cap \\ & \quad \mathcal{C}[[p(\bar{x}, \bar{y}) :- A_0]](\mathcal{P}[[P]])[[p(\bar{i}, \bar{o})]](\bar{\exists}_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \bar{\exists}_{\bar{v}}\delta))) \end{aligned}$$

Without loss of generality, we assume that the variables of \bar{i} and \bar{o} do not appear in programs (as stated in the caption of Figure 3.1) or in derivations. There are two cases to consider:

1. $\bar{\exists}_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \bar{\exists}_{\bar{v}}\delta) = \downarrow \text{true}$. This case is trivial, as the right hand side of the proof obligation collapses to $\downarrow \text{true}$.
2. $\bar{\exists}_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \bar{\exists}_{\bar{v}}\delta) \neq \downarrow \text{true}$. In this case, the right hand side of the proof obligation becomes

$$\begin{aligned} & \bar{\exists}_{\bar{u}}(\downarrow(\bar{i} = \bar{u}) \cap \bar{\exists}_{\bar{i}}(\downarrow(\bar{x} = \bar{i}) \\ & \quad \cap (\mathcal{A}[A_0]](\mathcal{P}[[P]])(\bar{\exists}_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \bar{\exists}_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \bar{\exists}_{\bar{v}}\delta)))))) \end{aligned}$$

By assumption on δ , we have

$$\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}\delta$$

Adding the same constraints and quantifications to both sides of this containment, by using Propositions 2.2 and 2.3.1 we obtain

$$\begin{aligned} & \bar{\exists}_{\bar{y}}(\bar{y} = \bar{o} \wedge \bar{\exists}_{\bar{o}}(\bar{o} = \bar{v} \wedge \bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma'))) \\ & \in \bar{\exists}_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \bar{\exists}_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \bar{\exists}_{\bar{v}}\delta)) \end{aligned}$$

Focusing for a moment on the lefthand side of this containment, we eliminate the intermediate fresh variables as follows:

$$\begin{aligned} & \bar{\exists}_{\bar{y}}(\bar{y} = \bar{o} \wedge \bar{\exists}_{\bar{o}}(\bar{o} = \bar{v} \wedge \bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma'))) \\ & = \bar{\exists}_{\bar{y}}\bar{\exists}_{\bar{o}}\bar{\exists}_{\bar{v}}\bar{\exists}_{\bar{o} \cup \bar{y}}\bar{\exists}_{\bar{v} \cup \bar{o} \cup \bar{y}}(\bar{y} = \bar{o} \wedge \bar{o} = \bar{v} \wedge \bar{v} = \bar{v}' \wedge \sigma') \end{aligned}$$

Because \bar{y} , \bar{o} , \bar{v} , and \bar{v}' are mutually disjoint

$$= \bar{\exists}_{\bar{y}}(\bar{y} = \bar{o} \wedge \bar{o} = \bar{v} \wedge \bar{y} = \bar{v}' \wedge \sigma')$$

Simplifying the existentials and rearranging the equations

$$= \bar{\exists}_{\bar{y}}(\bar{y} = \bar{v}' \wedge \sigma')$$

By using Proposition 2.1 and the fact that

the variables of \bar{o} and \bar{v} are disjoint from \bar{y} , \bar{v}' , and $\text{vars}(\sigma')$

Thus, we may rewrite the containment as follows:

$$\bar{\exists}_{\bar{y}}(\bar{y} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \bar{\exists}_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \bar{\exists}_{\bar{v}}\delta))$$

Projecting both sides once more yields

$$\exists_{\bar{v}_0}(\exists_{\bar{y}}(\bar{y} = \bar{v}' \wedge \sigma')) \in \exists_{\bar{v}_0}(\exists_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \exists_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \exists_{\bar{v}}\delta)))$$

Now, $\exists_{\bar{v}_0}(\bar{y} = \bar{v}' \wedge \sigma') \leq \exists_{\bar{v}_0}(\exists_{\bar{y}}(\bar{y} = \bar{v}' \wedge \sigma'))$ because $\bar{y} \subseteq \bar{v}_0$. Furthermore, because \bar{v}'_0 is just $\rho_0(\bar{v}_0)$, $\bar{y} \subseteq \bar{v}_0$, and $\rho_0(\bar{y})$ can be written \bar{v}' , we have $\bar{v}_0 = \bar{v}'_0 \leq \bar{y} = \bar{v}'$. So, because the righthand side of the containment is downwards closed, we may simplify the containment to

$$\exists_{\bar{v}_0}(\bar{v}_0 = \bar{v}'_0 \wedge \sigma') \in \exists_{\bar{v}_0}(\exists_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \exists_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \exists_{\bar{v}}\delta)))$$

This is the antecedent of the induction hypothesis, so we may conclude the consequent:

$$\begin{aligned} & \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma') \\ & \in \mathcal{A}[[A_0]](\mathcal{P}[[P]]) (\exists_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \exists_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \exists_{\bar{v}}\delta))) \end{aligned}$$

Adding new constraints and quantifiers to both sides, by using Propositions 2.2 and 2.3.1 we obtain

$$\begin{aligned} & \exists_{\bar{i}}(\bar{i} = \bar{u} \wedge \exists_{\bar{x}}(\bar{x} = \bar{i} \wedge \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma'))) \\ & \in \exists_{\bar{i}}(\downarrow(\bar{i} = \bar{u}) \cap \exists_{\bar{x}}(\downarrow(\bar{x} = \bar{i}) \cap \\ & \quad \mathcal{A}[[A_0]](\mathcal{P}[[P]]) (\exists_{\bar{y}}(\downarrow(\bar{o} = \bar{y}) \cap \exists_{\bar{o}}(\downarrow(\bar{v} = \bar{o}) \cap \exists_{\bar{v}}\delta)))))) \end{aligned}$$

Focusing for a moment on the lefthand side of this containment, we again eliminate intermediate fresh variables as follows:

$$\begin{aligned} & \exists_{\bar{i}}(\bar{i} = \bar{u} \wedge \exists_{\bar{x}}(\bar{x} = \bar{i} \wedge \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma'))) \\ & = \exists_{\bar{u}} \exists_{\bar{i} \cup \bar{u}}(\bar{x} = \bar{u} \wedge (\bar{x} = \bar{i} \wedge \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma'))) \\ & \quad \text{because } \bar{u}, \bar{x}, \text{ and } \bar{i} \text{ are mutually disjoint} \\ & = \exists_{\bar{u}}(\bar{x} = \bar{i} \wedge (\bar{x} = \bar{u} \wedge \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma'))) \\ & \quad \text{simplifying the existential and rearranging} \\ & = \exists_{\bar{u}}(\bar{x} = \bar{u} \wedge \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma')) \\ & \quad \text{by using Proposition 2.1 and} \\ & \quad \text{the fact that } \bar{i} \text{ is disjoint from the other variables.} \end{aligned}$$

Now, because \bar{u}'_0 is just $\rho_0(\bar{u}_0)$, $\bar{x} \subseteq \bar{u}_0$, and $\rho_0(\bar{x})$ can be written \bar{u}' , we have $\bar{u}_0 = \bar{u}'_0$ implies $\bar{x} = \bar{u}'$. Thus, we can continue our simplification:

$$\begin{aligned} & \exists_{\bar{u}}(\bar{x} = \bar{u} \wedge \exists_{\bar{u}_0}(\bar{u}_0 = \bar{u}'_0 \wedge \sigma')) \\ & = \exists_{\bar{u}} \exists_{\bar{u}_0 \cup \bar{u}}(\bar{x} = \bar{u} \wedge (\bar{x} = \bar{u}' \wedge (\bar{u}_0 = \bar{u}'_0 \wedge \sigma'))) \\ & \quad \text{by the observations above} \\ & = \exists_{\bar{u}}(\bar{x} = \bar{u}' \wedge \bar{u}_0 = \bar{u}'_0 \wedge (\bar{u} = \bar{u}' \wedge \sigma')) \\ & \quad \text{simplifying the existential and rearranging} \\ & = \exists_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \\ & \quad \text{by using Proposition 2.1 and} \\ & \quad \text{the fact that the program variables } \bar{x} \text{ and } \bar{u}_0 \\ & \quad \text{are disjoint from the derivation variables } \bar{u}. \end{aligned}$$

Now we can return to the containment and rewrite it as required to satisfy the proof obligation:

$$\exists_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma')$$

$$\begin{aligned} & \in \exists_{\bar{u}}(\downarrow (\bar{i} = \bar{u}) \cap \exists_{\bar{i}}(\downarrow (\bar{x} = \bar{i}) \cap \\ & \quad \mathcal{A}[[A_0]](\mathcal{P}[[P]])(\exists_{\bar{y}}(\downarrow (\bar{o} = \bar{y}) \cap \exists_{\bar{o}}(\downarrow (\bar{v} = \bar{o}) \cap \exists_{\bar{v}}\delta)))))) \end{aligned}$$

$A = \sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i$ By well moding we have $\bar{u} \subseteq \cup_{i=1}^n \text{vars}(c_i) \cup \bar{u}_i$, $\bar{u} \supseteq \cup_{i=1}^n \bar{u}_i$, and $\bar{v} = \cup_{i=1}^n \bar{v}_i$.

By assumption we have

$$\exists_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \in \exists_{\bar{v}}\delta$$

Assume that $j \in \{1 \dots n\}$ is the index of the summand selected in d to solve $\sum_{i=1}^n \mathbf{ask}(c'_i) \rightarrow A'_i$. By projecting both sides, we obtain

$$\exists_{\bar{v}_j}(\exists_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma')) \in \exists_{\bar{v}_j}(\exists_{\bar{v}}\delta)$$

which can be simplified to

$$\exists_{\bar{v}_j}(\bar{v}_j = \bar{v}'_j \wedge \sigma') \in \exists_{\bar{v}_j}(\exists_{\bar{v}}\delta)$$

This is the antecedent of the induction hypothesis, so we may conclude the consequent:

$$\exists_{\bar{u}_j}(\bar{u}_j = \bar{u}'_j \wedge \sigma') \in \mathcal{A}[[A_j]](\mathcal{P}[[P]])(\exists_{\bar{v}}\delta)$$

Since $\mathcal{A}[[A_j]](\mathcal{P}[[P]])(\exists_{\bar{v}}\delta)$ is downwards closed, we may strengthen the left hand side by removing some of the existentials:

$$\exists_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \mathcal{A}[[A_j]](\mathcal{P}[[P]])(\exists_{\bar{v}}\delta) \tag{A.4}$$

Because $\sum_{i=1}^n \mathbf{ask}(c'_i) \rightarrow A'_i$ is reduced in d by selecting choice j , it follows that

$$\exists_{\text{vars}(c_j)}(\text{vars}(c_j) = \text{vars}(c'_j) \wedge \sigma') \in \downarrow c_j$$

By using Proposition 2.2, this now combines with (A.4) to give us

$$\begin{aligned} & \exists_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \wedge \exists_{\text{vars}(c_j)}(\text{vars}(c_j) = \text{vars}(c'_j) \wedge \sigma') \\ & \in \downarrow c_j \cap \mathcal{A}[[A_j]](\mathcal{P}[[P]])(\exists_{\bar{v}}\delta) \end{aligned}$$

It follows by the properties of union that

$$\begin{aligned} & \exists_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \wedge \exists_{\text{vars}(c_j)}(\text{vars}(c_j) = \text{vars}(c'_j) \wedge \sigma') \\ & \in \cup_{i=1}^n (\downarrow c_i \cap \mathcal{A}[[A_i]](\mathcal{P}[[P]])(\exists_{\bar{v}}\delta)) \end{aligned}$$

Quantifying both sides, by using Proposition 2.3.1 we obtain

$$\begin{aligned} & \exists_{\bar{u}}(\exists_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \wedge \exists_{\text{vars}(c_j)}(\text{vars}(c_j) = \text{vars}(c'_j) \wedge \sigma')) \\ & \in \exists_{\bar{u}}(\cup_{i=1}^n (\downarrow c_i \cap \mathcal{A}[[A_i]](\mathcal{P}[[P]])(\exists_{\bar{v}}\delta))) \end{aligned}$$

which can be simplified by using Proposition 2.1 to

$$\bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \bar{\exists}_{\bar{u}}(\cup_{i=1}^n (\downarrow c_i \cap \mathcal{A}[A_i](\mathcal{P}[[P]])(\bar{\exists}_{\bar{v}}\delta)))$$

By using the definition of \mathcal{A} , we obtain

$$\bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \mathcal{A}[\sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i](\mathcal{P}[[P]])\delta$$

as required.

$A = \mathbf{tell}(c)$ Because $\mathbf{tell}(c')$ is reduced in d , it follows that

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \downarrow c \tag{A.5}$$

Because removing an existential and conjoining a new equality both strengthen any constraint, we have

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \leq \bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma')$$

Therefore, it follows from the assumption on δ and from the fact that δ is downwards closed, that

$$\bar{\exists}_{\bar{u} \cup \bar{v}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{v}}\delta$$

Combining this with (A.5) and quantifying both sides, we have

$$\bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \bar{v} = \bar{v}' \wedge \sigma') \in \bar{\exists}_{\bar{u}}(\downarrow c \cap \bar{\exists}_{\bar{v}}\delta)$$

Since the variables of \bar{v} do not occur in \bar{u} , \bar{u}' , \bar{v}' , or σ' , this simplifies by using Proposition 2.1 to

$$\bar{\exists}_{\bar{u}}(\bar{u} = \bar{u}' \wedge \sigma') \in \bar{\exists}_{\bar{u}}(\downarrow c \cap \bar{\exists}_{\bar{v}}\delta)$$

as required to complete this case.

$A = \mathbf{Stop}$ There are two cases. If $\delta \neq \downarrow \mathbf{false}$, then $\mathcal{A}[\mathbf{Stop}](\mathcal{P}[[P]])\delta = \downarrow \mathbf{true}$, so the desired containment holds trivially. If $\delta = \downarrow \mathbf{false}$, then by assumption on δ , $\bar{\exists}_{\bar{v}}(\bar{v} = \bar{v}' \wedge \sigma') \subseteq \downarrow \mathbf{false}$. From this it follows that $\sigma' = \mathbf{false}$. So the desired containment is again trivial.

□

Acknowledgement

Moreno Falaschi was partially supported by the ESPRIT Basic Research Action 6707 ('Parforce'). Patrick Hicks was partially supported by NSF CCR-9542167. William Winsborough was partially supported by NSF CCR-9210975.

REFERENCES

1. G. Burn. Evaluation transformers – a model for parallel evaluation of functional languages. In *Proc. of ACM Conf. FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 447–470. Springer-Verlag, 1987.
2. G. Burn. The evaluation transformer model of reduction and its correctness. In S. Abramsky and T.S.E. Maibaum, editors, *International Joint Conference on Theory and Practice of Software Development*, volume 494 of *Lecture Notes in Computer Science*, pages 458–482. Springer-Verlag, 1991.
3. M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 331–345. The MIT Press, 1991.
4. M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. A confluent semantic basis for the analysis of concurrent constraint logic programs. *Journal of Logic Programming*, 30(1):53–81, 1997.
5. C. Codognet, P. Codognet, and M.-M. Corsini. Abstract Interpretation for Concurrent Logic Programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 215–232. The MIT Press, 1990.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
7. P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 43–119. Cambridge Univ. Press, 1982.
8. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.
9. K. Davis and P. Wadler. Backwards strictness analysis: Proved and improved. In *Proc. of the 2nd Glasgow Workshop on Functional Programming*, Springer Workshops in Computing. Springer-Verlag, 1990.
10. S. K. Debray. QD-Janus: A sequential implementation of Janus in Prolog. *Software Practice and Experience*, 23(12):1337–1360, 1993.
11. S. K. Debray, D. Gudeman, and P. Bigot. Optimization of suspension-free logic programs. In *Proc. of International Logic Programming Symposium*, pages 487–501. The MIT Press, 1994.
12. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of ACM Sigplan PLDI'90*, pages 174–188. ACM Press, 1990.
13. S. K. Debray, P. Lopez-Garcia, M. Hermenegildo, and N.-W. Lin. Lower bound cost estimation for logic program. In Jan Maluszyński, editor, *Proc. of International Logic Programming Symposium*, pages 291–305. The MIT Press, 1997.
14. P. Dybjer. Inverse image analysis generalises strictness analysis. *Information and Computation*, 90(2):194–216, 1991.
15. M. Falaschi, P. Hicks, and W. Winsborough. Demand Transformation Analysis for Concurrent Constraint Programs. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 333–347. The MIT Press, 1996.

16. C. Hall and D. Wise. Compiling strictness into streams. In *14th ACM Symposium on Principles of Programming Languages*, pages 132–143. ACM Press, 1987.
17. C. Hall and D. Wise. Generating function versions with rational strictness patterns. *Science of Computer Programming*, 12:39–74, 1989.
18. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.
19. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog using Type Graphs. In *Proc. of PLDI*, volume 29 of *SIGPLAN Notices*, pages 337–348. ACM Press, 1994.
20. R.J.M. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 63–102. Ellis Horwood, 1987.
21. R.J.M. Hughes. Backward analysis of functional programs. In D. Bjorner, A. Ershov, and N.D. Jones, editors, *Proc. of IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, 1988.
22. R.J.M. Hughes and J. Launchbury. Reversing abstract interpretations. *Science of Computer Programming*, 22:307–326, 1994.
23. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):205–258, 1992.
24. T. P. Jensen and T. E. Mogensen. A Backwards Analysis for Compile-time Garbage Collection. In *Proc. ESOP'90*, volume 432 of *Lecture Notes in Computer Science*, pages 227–239. Springer-Verlag, 1990.
25. S. B. Jones and D. Le Metayer. Compile-time garbage collection by sharing analysis. In *Proc. of ACM Conf. FPCA*, pages 54–74. Addison-Wesley, 1989.
26. A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In Jan Maluszyński, editor, *Proc. of International Logic Programming Symposium*, pages 261–276. The MIT Press, 1997.
27. A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 478–492. The MIT Press, 1992.
28. G. Lindstrom. Static evaluation of functional programs. In R.L. Wexelblat, editor, *Proc. of the Symp. on Compiler Construction*, volume 21 of *SIGPLAN Notices*. ACM Press, 1986.
29. M.J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proc. of Third IEEE Symp. on Logic In Computer Science*, pages 348–357. IEEE Press, 1988.
30. Julio Mariño-Carballo, Angel Herranz-Nieva, and J. J. Moreno-Navarro. Demand-ness analysis with dependence information for functional logic languages. In *ILPS Workshop on Global Compilation*, pages 99–114, 1993.
31. A. Mycroft. *Abstract Interpretation and Optimising transformations for applicative programs*. PhD thesis, University of Edinburgh, 1981.
32. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of Concurrent Constraint Programming. In *18th ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.
33. K. Ueda and M. Morita. Moded Flat GHC and its Message-Oriented Implementation Technique. *New Generation Computing*, 13(1):3–43, 1994.

-
34. P. Wadler. Strictness analysis on non-flat domains. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 266–275. Ellis Horwood, 1987.
 35. P. Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Proc. of ACM Conf. FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, 1987.
 36. E. Zaffanella, G. Levi, and R. Giacobazzi. Abstracting Synchronization in Concurrent Constraint Programming. In *Proc. 5th Int'l Symposium on Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 57–72. Springer-Verlag, 1994.