# Languages of Play

Towards semantic foundations for game interfaces


## ABSTRACT

Formal models of games help us account for and predict behavior, leading to more robust and innovative designs. While the games research community has proposed many formalisms for both the "game half" (game models, game description languages) and the "human half" (player modeling) of a game experience, little attention has been paid to the *interface* between the two, particularly where it concerns the player expressing her intent toward the game. We describe an analytical and computational toolbox based on programming language theory to examine the phenomenon sitting between control schemes and game rules, which we identify as a distinct *player intent language* for each game.



## KEYWORDS

game interfaces, programming languages, formal methods

**ACM Reference format:**
. 2016. Languages of Play. In *Proceedings of ACM Conference, Cape Cod, MA, USA, August 2017 (FDG'17),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn


## 1 INTRODUCTION

To study how players interact with games, we examine both the rules of the underlying system and the choices made by the player. The field of player modeling has identified the value in constructing models of player cognition: while a game as a self-contained entity can allow us to learn about its mechanics and properties as a formal system, we cannot understand the *dynamics* of that system unless we also account for the human half of the equation. Meanwhile, Crawford [6] identifies the necessity of looking at the complete information loop created between a player and a digital game, defining *interactivity* in games as their ability to carry out a conversation with a player, including listening, processing, and responding, identifying the importance of all three to the overall experience.

Given this understanding of games-as-conversation, we should expect to discover something like a *language* through which games and players converse. In Figure 1, we illustrate the game-player loop as a process which includes an *interface* constituting such a language. The Game Ontology Project [22] describes game interfaces as follows:

> *The interface is where the player and game meet, the mapping between the embodied reactions of the player and the manipulation of game entities. It refers to both how the player interacts with the game and how the game communicates to the player.*

The first part, *how the player interacts with the game*, is called the *input*, which is further subdivided into *input device* and *input method*. Input devices are hardware controllers (mice, keyboards, joysticks, etc.) and input *methods* start to brush the surface of something more semantic: they include choices about *locus of manipulation* (which game entities can the player control?) and direct versus indirect action, such as selecting an action from a menu of options (indirect) versus pressing an arrow key to move an avatar (direct).

However, any close look at interactive fiction, recent mobile games, or rhythm games (just to name a few examples) will reveal that design choices for input methods have much more variety and possibility than these two dimensions. In this paper, we propose a framework to support analyzing and exploring that design space.

Our first step is to refine input *methods* to input *languages*: we are ultimately asking, how can a player communicate their intent, and how does a digital game recognize this intent? So, in linguistic terms, the "phonemes" of such a language are hardware controls such as button presses and joystick movement. Then, the syntax and semantics are defined by each game individually, depending on what meaning they give to each control input. This language defines the verbs of the game, which may include moving, selecting inventory items, examining world items, applying or using items, entering rooms, and combat actions. (Note that such a language is also distinct from a game's *mechanics*: mechanics include system behavior which is out of the player's control, such as falling with gravity, non-player character actions, and other autonomous behavior.) This language is both *afforded* by the game designer—she must communicate to players which verbs are available—and *constrained* by her—she may declare certain expressions invalid.

Since the constraints on such a language are wholly determined by a piece of software (the game interface), we argue that it has more in common with a *programming language*[1] (PL) than a natural language. Accordingly, each game in some sense *defines its own programming language.* In a slogan, we could term this project *games as programming languages.* Specifically, we propose *player intent languages* as a PL-inspired framework for designing player-game interfaces.

This analogy opens up a whole field of methodology to try applying to games. The PL research community has a long and deep history of assigning mathematically formal semantics to languages and analyzing those semantics. As games researchers become more interested in the emergent consequences of the systems they assemble, the tools of PL theory have a lot to offer. For example, PL


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.


*FDG'17, Cape Cod, MA, USA*


© 2016 ACM. 978-x-xxxx-xxxx-x/YY/MM…$15.00


DOI: 10.1145/nnnnnnn.nnnnnnn


[1] We define programming languages broadly as formal languages whose meaning is fully grounded in a computational system.

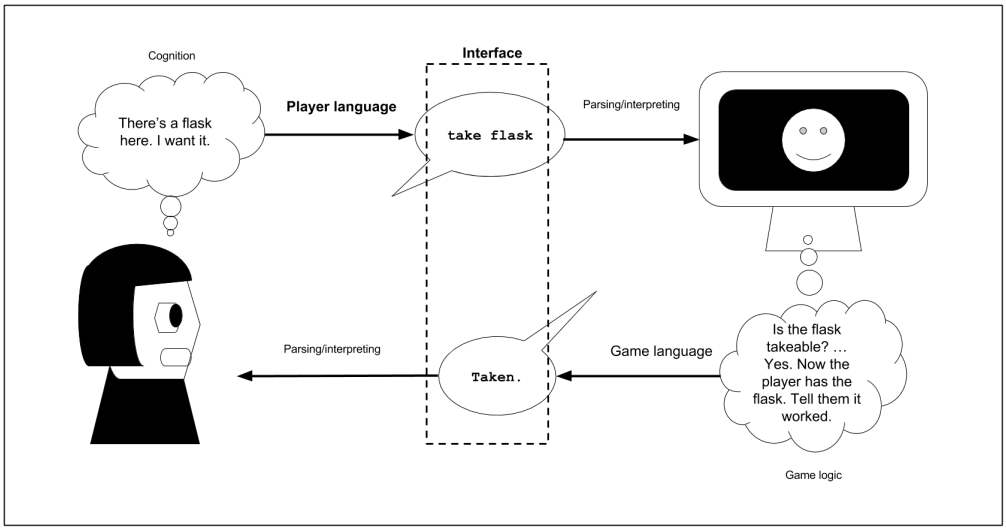


**Figure 1: A process diagram of a game loop: player and game conversation as it relates to language, interface, and cognition.**

theory provides an account of *compositionality*, i.e. how fragments of expression fit together to form higher-level meanings. In games, this translates into being able to understand player *skills* or strategies as compositions of player actions, which we demonstrate in this paper by using a formalized input language as a kind of "player AI scripting language."

Furthermore, by considering a player's language of expression as an object of study in its own right, we center them as a co-designer of the experience afforded by a game. When we treat a player's interactions as not simply an arbitrary sequence of button presses that advances and reveals the designer's intent, but instead as its own distinct *voice* that a game system must listen and respond to, we enable the player to *co-create* with the system, potentially developing deeper systems understanding and emotional investment.

In this paper, we propose *player intent languages*, a programming languages-based approach to designing player-game interfaces as formal objects. In the remainder of the paper, we tour this approach through concrete examples. Specifically, we consider a simple game design space and make points in this space precise by introducing the components of a programming language: abstract syntax (Section 4), type system (Section 6), and operational semantics (Section 5). For each, we give a corresponding concept in the game world. By grounding these game concepts in analogous programming language concepts, we gain powerful PL reasoning tools and design methodologies to benefit the game design process.

We demonstrate the payoff of this line of thought by extending the metaphor with *play traces* as *straight-line programs* (Section 7), and *player skills* as *general programs* (e.g., programs with parameters, branching and looping) (Section 8). These structures give semantic logs and general strategies, respectively, for accomplishing a task in the game world. The framework of player intention languages gives rise to further research directions, which we briefly outline and discuss before concluding (Sections 9 and 10).

## 2 RELATED WORK

Cardona-Rivera and Young [4] detailed a conceptual framework following the slogan *games as conversation*, grounding the communicative strategies of games in cognitive science for human-to-human conversational understanding, such as Grice's maxims [9]. They offer a linguistic and semiotic approach to understanding how a game communicates affordances (possibilities for action) to a player. For an account of the game's half of the equation, which includes the visual, textual, and audio feedback mechanisms intended to be processed by the player, this application of linguistics, psychology, and design seems appropriate, much like the study of cinematic language for film. On the other hand, we argue that a PL approach better supports understanding of the player-to-game direction, since the language the player speaks toward a digital game is formal and unambiguous.

Researchers have previously recognized the value in formalizing interaction vocabularies, realizing certain interaction conventions as a *single* "video game description language" [7] whose implementation as VGDL [19] has been used in game AI research. We suggest instead that the design space of player languages is as varied as the design space of programming languages and herein give an account of what it would mean to treat each language individually. Our project suggests that an appropriately expressive computational framework analogous to VGDL should be one that can accommodate the encoding of many such languages, such as a *meta-logical framework* like the Twelf system for encoding and analyzing programming language designs [18].

Any investigation into formalizing actions within an interactive system shares ideas with "action languages" in AI extending as far back as McCarthy's situation calculus [13] and including planning languages and process calculi. These systems have been studied in the context of game design, e.g. the Ludocore system [20]; however, AI researchers are mainly interested in these formalisms as internal representations for intelligent systems and the extent to which

they support reasoning. Conversely, we are interested their potential to support player expression and facilitate human-computer conversation.

Some theoretical and experimental investigations have been carried out about differences between game interfaces along specific axes, such as whether the interface is "integrated" (or one might say diagetic), versus extrinsic to the game world in the form of menus and buttons [11, 12]. These investigations suggest an interest in more detailed and formal ontologies of game interfaces, which our work aims to provide.

From the PL research side, we note existing efforts to apply PL methodology to user interfaces, specifically in the case of program editors. Hazelnut is a formal model of a program editor that enforces that every edit state is meaningful (it consists of a well-defined syntax tree, with a well-defined type) [14]. Its type system and editing semantics permit *partial programs*, which contain missing pieces and well-marked type inconsistencies. Specifically, Hazelnut proposes a *editing language*, which defines how a cursor moves and edits the syntax tree; the planned benefits of this model range from better editing assistance, the potential to better automate systematic edits, and further context-aware assistance and automation based on statistical analysis of (semantically-rich) corpora of recorded past edits, which consist of *traces* from this language [15]. Likewise, in the context of game design, we expect similar benefits from the lens of language design.

## 3 A FRAMEWORK FOR PLAYER INTENT LANGUAGES

In the formal study of a programming langugage, one may define a language in three parts: syntax, type system, and operational semantics.

- The *syntax* is written in the form of a (usually) context-free grammar describing the allowable expressions. One sometimes distinguishes between *concrete syntax*, the literal program tokens that the programmer strings together in the act of programming, and *abstract syntax*, the normalized "syntax tree" structures that ultimately get interpreted.
- An *operational semantics* defines how runnable programs (e.g. a function applied to an argument) *reduce* to values. This part of the definition describes how actual computation takes place when programs in the language are run. It is important to note that the operational semantics need not reflect the actual *implementation* of the language, nor is it specific to a "compiled" versus "interpreted" understanding of the language: it is simply a mathematical specification for how any compiler or interpreter for the language should behave.
- A *type system* further refines the set of syntactically valid expressions into a set of *meaningful* expressions, and provides a mapping between an expression and an approximation of its meaning. Type systems are usually designed in conjunction with the operational semantics to have the property that every expression assigned a meaning by the type system should have a well-defined runtime behavior. In practice, however, type systems can only approximate this correspondence. Some err on the more permissive side–e.g. C's type system will permit invalid memory accesses with no language-defined behavior–and some err on the more restrictive side, e.g. Haskell's type system does not permit any untracked side-effects, at the expense of easily authoring e.g. file input/output (without first learning the details of the type system).

| PL concept | Game concept | |
|---|---|---|
| Syntax | Recognized player intents | (Section 4) |
| Operational semantics | Game mechanics | (Section 5) |
| Type system | Contextual interface | (Section 6) |
| Straight-line programs | Play traces | (Section 7) |
| General programs | Player skills | (Section 8) |

**Table 1: Player intent languages: Formal decomposition (left) and correspondances (right).**

Providing a formal language definition in programming languages research has several purposes. One is that it enables researchers to explore and prove formal properties of their language, such as *well-typed programs don't go wrong*, or in a language for concurrency, a property like deadlock freedom. However, an even more crucial advantage of a language specification is not mathematical rigor but human capacity to do science. A language definition is a *specification*, similar to an application programmer interface (API) or an IEEE standard: it describes an unambiguous interface to the language along an *abstraction boundary* that other human beings may access, understand, and implement, without knowing the internals of a language implementation. It is a necessary component of reproducibility of research, and it allows researchers to build on each other's work. We believe that an embrace of formal specification in games research can play a similarly important function.

Having provided loose definitions of these terms, we now wish to draw out the analogy between a *language* specification and a *game* specification. To treat a game in this manner, we wish to consider player affordances and actions, as well as their behavior (mechanics) in the context of the game's running environment. We summarize the components of this correspondence in Table 1.

We will use as a running example a minimal virtual environment with two player actions: (1) movement through a discrete set of rooms in a pre-defined map (move); (2) acquiring objects placed in those rooms to store in a player inventory (take). We consider five (somewhat aribitrary) possibilities in the design space of interfaces for such a game, summarized visually in Figure 2:

- **Point-and-Click:** A first-person viewpoint interface where the meaning of each click is defined based on the region the cursor falls in. Clicking near any of the four screen edges moves in that direction; clicking on a sprite representing an item takes it.
- **Bird's-Eye:** A top-down viewpoint interface where the player can see multiple rooms at once, and can click on rooms and objects that are far away, but those clicks only do something to objects in the same room or adjacent rooms.
- **WASD+:** A keyboard or controller-based interface with directional buttons (e.g. arrow keys or WASD) move an

avatar in the correspondingi direction, and a separate key or button expresses the take action, which takes any object in the same room. (This interface may be used for either of the two views described above.)
- **Command-Line:** The player interacts by typing free-form text, which is then parsed into commands, such as `take lamp` and `move north`.
- **Hypertext**: A choice-based interface where all available options are enumerated as textual links from which the player chooses.

In the following sections, we will consider these possibilities in light of design choices relevant to the specified aspect of PL design.

## 4 PLAYER INTENT AS SYNTAX

The *syntax* of a game is its space of recognized player intentions. Note that *intention* is different from *action* in the sense that we don't necessarily expect each well-formed intention to change anything about the game state: a player can intend to move north, but if there is no room to the north of the player when she expresses this intent, no change to the game's internal state will occur. Nonetheless, depending on the design goals of the game, we may wish to recognize this as a valid intent so that the game may respond in some useful way (e.g. with feedback that the player cannot move in that direction).

In our example game, the choice of syntax answers questions such as: can the player click anywhere, or only in regions that have meaning? Can the player type arbitrary commands, or should we provide a menu or auto-complete text so as to prevent the player from entering meaningless commands? In PL, we can formalize these decisions by describing an *abstract syntax* for our language, which is typically assumed to be context-free and thus specified as a Bachus-Naur Form (BNF) grammar. Our examples below follow the interfaces shown visually in Figure 2.

**WASD+ Interface**: One way of writing the BNF for the WASD+ interface is:

$$\begin{aligned} \textit{direction} &::= \text{north} \mid \text{south} \mid \text{east} \mid \text{west} \\ \textit{intent} &::= \text{move}\langle \textit{direction} \rangle \mid \text{collect} \end{aligned}$$

The hardware interface maps onto this syntax quite directly: each arrow key maps onto a move action in the corresponding direction, and the specified other key maps onto collect.

**Bird's Eye View Mouse Interface**: On the other hand, a clicking-based interface to a top-down map could enable the player to click on any room on the map and any item within a room. This syntax would look like:

$$\begin{aligned} \textit{room} &::= \text{courtyard} \mid \text{library} \mid \text{quarters} \mid \text{lab} \\ \textit{item} &::= \text{flask} \mid \text{book} \\ \textit{entity} &::= \text{room} \mid \text{item} \\ \textit{intent} &::= \text{click}\langle \textit{entity} \rangle \end{aligned}$$

Note that this syntax, compared to that for WASD+, describes a larger set of possible utterances, even though it has the exact same set of permitted game behaviors (a player may only move into adjacent rooms and take items that share a room with them).

**Command Line Interface**: The command-line interface would have an even larger space of expressible utterances if we consider all typed strings of characters to be valid expressions, but that syntax is too low-level for linguistic considerations. Supposing we interpose a parsing layer between arbitrary typed strings and syntactically-well-formed commands, we can define the abstract syntax as follows (where *direction* and *item* are assumed to be defined as they were in the previous examples):

$$\textit{intent} ::= \text{move}\langle \textit{direction} \rangle \mid \text{take}\langle \textit{item} \rangle$$

Assuming the player "knows the language," i.e. knows that move and take are valid commands, and in fact the *only* valid commands, and assuming that she knows how to map the visual affordances (e.g. image of the flask) to the typed noun (e.g. `flask`), the experience afforded by this interface is quite similar to the WASD+ interface. The main difference is that the player must specify an *argument* to the take command, asking the player to formulate a more complete (and unambiguous) intent by actually naming the object she wishes to take.

**Hypertext interface**: Finally, we consider the intent language for the hypertext interface. This is one of the most difficult interfaces to formulate in linguistic terms, because it either requires that we formalize link selection in an acontextual way (e.g. as a numeric index into a list of options of unknown size) or that we formulate each link *from each page* as its own separate command, each of which has meaning in only one specific game context (namely, when the player is on the page containing that link). The former feels like a more general formulation of hypertext that is not relevant to any particular game, and since we are aiming to provide a correspondence between specific games and languages, we opt for the latter:

$$\begin{aligned} \textit{intent} &::= \text{select}\langle \textit{choice} \rangle \\ \textit{choice} &::= \text{take\_flask\_from\_lab} \\ &\mid \text{take\_book\_from\_library} \\ &\mid \text{go\_south\_from\_lab} \\ &\mid \text{go\_east\_from\_lab} \\ &\mid \text{go\_west\_from\_lab} \\ &\mid \text{go\_north\_from\_library} \\ &\mid \ldots \end{aligned}$$

Some hypertext authors put a lot of effort into scaffolding the choice-based experience with a richer language, e.g. by repeating the same set of commands that behave in consistent ways across different pages, or by creating menu-like interfaces where text cycles between options on an otherwise static page. In this way, hypertext as a medium might be said as providing a platform for designers to create their own interface conventions, rather than relying on a set of pre-established ones; by the same token, hypertext games created by inexperienced interface (or language) designers may feel to players like being asked to speak a foreign language for each new game.

### Additive and subtractive properties of syntax

By now we are able to observe that, just like the rest of a game's rules, its syntax has both additive and subtractive properties. It provides the menu of options for which hardware interactions are *relevant*, i.e. likely to result in meaningful interaction with the game

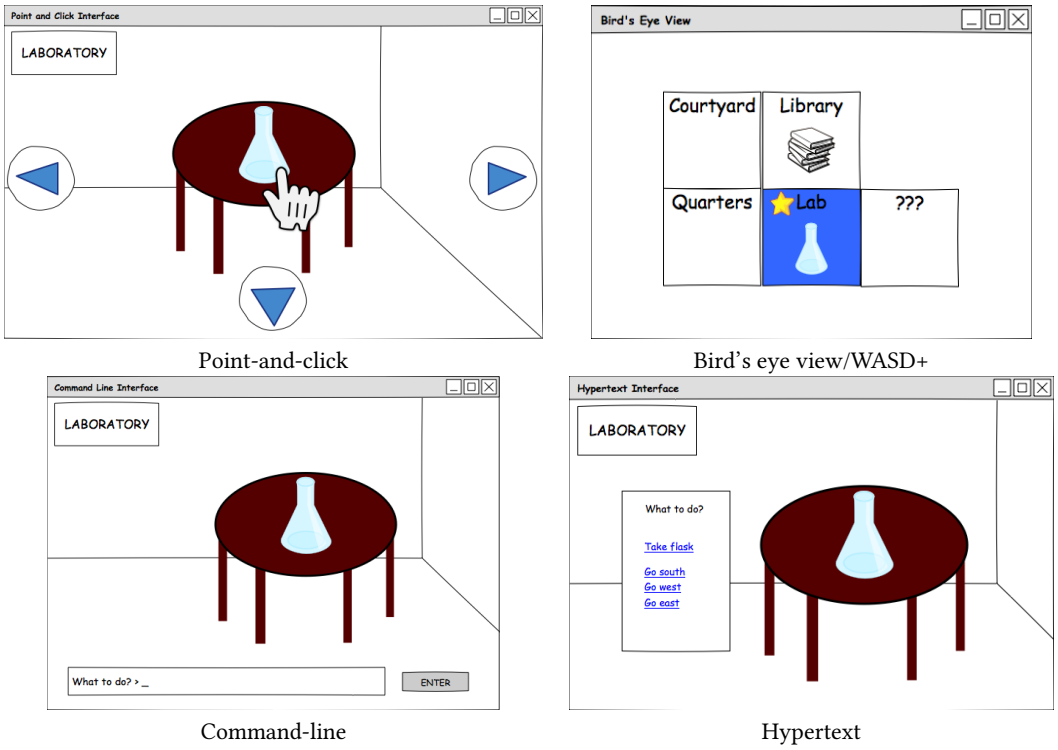


**Figure 2: Four different user interfaces for the move/take game.**

system, but it also establishes which utterances within that set are *disallowed*, or ill-formed—e.g. that it is not meaningful to say "take" without providing an object to the command, or that "take north" is ill-formed.

Correspondingly, an important decision that impacts game design is (a) how discoverable the additive affordances are (e.g. can the player determine that "examine" is a meaningful verb without already possessing literacy in the game's genre?) and (b) the extent to which the user interface makes meaningless expressions impossible to form. For example, in a hypertext interface, all links lead somewhere—so every intent the player can form, i.e. clicking a link on the page, will get a valid response from the game, whereas "take fnord" typed at a command-line interface may be recognized by the parser, but meaningless to a game where "fnord" is not a noun. Decisions about these two (related) dimensions will determine the extent to which *learning the language*, an exploratory but sometimes frustrating process, is a central challenge of the game.

## 5 GAME MECHANICS AS OPERATIONAL SEMANTICS

Consider the right-hand-side of Figure 1: The game parses and interprets an unambiguous syntax of player intent, which either advances the game state (as shown in the figure), or the game state cannot advance as the player intended, which the game somehow signals to the player (not shown).

We model a *response* to the move-take player intent with the following BNF definition, of *resp*:

$$resp \quad ::= \quad \text{success} \mid \text{failure}$$

Figure 1 shows the case where the player intent of "take flask" (formally, the syntax "take") leads to the game world performing this intent as a successful action, and responding accordingly with "Taken." Formally, we model this *resp* as "success," as defined above. Likewise, if the flask cannot be taken (e.g., the flask is not near the player, or is already in the player's possession, etc.), the game responds with failure.

As with the player's intent, which may exist as both raw input *and* as formal syntax, each formal response can be conveyed as raw output in a variety of ways (e.g., as textual words, pictures, or sounds). In real games there are often two two levels of game-to-player feedback: Feedback through the game world, and feedback *outside* the game world (e.g., a pop-up message with an error, guidance or advice). For simplicity, move-take gives feedback outside of the game state, e.g., as pop-up messages.

To capture the formal relationship between player intent as syntax, and game response as syntax, we introduce a four-place *game-step* relation:

$$\langle G_1; intent \rangle \longrightarrow \langle G_2; resp \rangle$$

This relation formalizes the dynamic behavior of the right-hand-side of Figure 1. It consists of four parts: An initial game state $G_1$, a player intent *intent*, a resulting game state $G_2$ and a game response *resp*.

As is standard in PL formalisms, we give the rules that define this relation as inductive *inference rules*, which can each be read as a logical inference. That is, given evidence for the premises on the top of the rule, we may conclude the bottom of the rule. For instance, here are two example rules:

$$\frac{G_1 \vdash \mathsf{playerNear\ flask} \qquad \mathsf{playerTake}(G_1, \mathsf{flask}) \equiv G_2}{\langle G_1; \mathsf{take}\rangle \longrightarrow \langle G_2; \mathsf{success}\rangle}$$

$$\frac{G \vdash \mathsf{not}\big(\mathsf{playerNear\ flask}\big)}{\langle G; \mathsf{take}\rangle \longrightarrow \langle G; \mathsf{failure}\rangle}$$

The first rule formalizes the case shown in the RHS of Figure 1. The second rule formalizes the opposite outcome, where the flask cannot be taken. Notably, the first rule has two premises: To be taken by the player, it suffices to show that in the current game state $G_1$, the flask is near the player (first premise), and that there exists an advanced game state $G_2$ that results from the player taking this flask (second premise). In the second rule, there is only one game state $G$, since the flask cannot be taken and consequently, the game state does not change.

Like the syntax of player intent and game responses, these rules are also unambiguous. Consequently, we view these rules as a mathematical definition with an associated strategy for constructing formal (and informal) proofs about the game mechanics.

For instance, we can formally state and attempt to prove that for all player intents *intent* and game worlds, $G_1$, there exists a corresponding game world $G_2$ and game response *resp*. That is, the statement of the following conjecture:

$$\forall G_1, \mathit{intent}. \quad \exists G_2, \mathit{resp}. \quad \langle G_1; \mathit{intent}\rangle \longrightarrow \langle G_2; \mathit{resp}\rangle$$

Using standard PL techniques, the proof of this conjecture gives rise to an abstract algorithm that implements the game mechanics by analyzing each possible case for the current game-state and player intent. Indeed, this is precisely the reasoning required to show that an implementation of the game is *complete* (i.e., there exists no state and input that will lead the game into an undefined situation).

Reasoning about this completeness involves reasoning about when each rule is applicable. For instance, the rule for a successful player intent of take requires two premises: $G_1 \vdash \mathsf{playerNear\ flask}$ and $\mathsf{playerTake}(G_1, \mathsf{flask}) \equiv G_2$. The first is a *logical judgment* about the game world involving the proposition playerNear flask, which may or may not be true, but which is computable. The second is a *semantic function* that transforms a game state into one where the player takes a given object; in general, this function may be undefined, e.g., if the arguments do not meet the function's *pre-conditions*. For instance, a precondition of the function $\mathsf{playerTake}(-,-)$ may be that the object is not already in the possession of the player. In this case, to show that the game mechanics do not "get stuck", we must show that playerNear flask implies that the player does not already posses the flask. (Otherwise, we should add another rule to handle the case that the player intends to take the flask but already possesses it). The design process for programming language semantics often consists of trying to write examples and prove theorems, and failing; these experiences inform systematic revisions to the language definition.

In the next section, we refine intents and semantics further by introducing the notion of a *context*.

# 6 CONTEXTUAL INTERFACES AS TYPE SYSTEMS

Decisions about interface syntax can, to some extent, limit or expand the player's ability to form intents that will be met with failure, such as moving through a wall or taking an object that does not exist. But sometimes, whether an utterance is *meaningful* or not will depend on the runtime game state, and can be considered a distinct question from whether it is well-formed. For example, whether or not we can *take flask* depends on whether the flask is present, but if the flask is an object somewhere in the game, we must treat this command as well-formed *syntax* and relegate its failure to integrate with the runtime game environment to the *mechanics* (operational semantics).

However, some user interfaces nonetheless restrict the set of recognized utterances in a way that depends on current game state. Consider a point-and-click interface that changes the shape of the cursor to a hand whenever it hovers over an interactable object, and only recognizes clicks when it is in this state. Alternatively, consider the hypertext interface, which only recognizes clicks on links made available in the current page. Providing the player with *only the option of saying* those utterances that "make sense" in this regard corresponds to a strong static type system for a programming language.

Type systems are typically formalized be defining a relation between expressions $e$ and contexts $\Gamma$. Contexts are sets of specific circumstances in which an expression is valid, or well-typed. Usually, these circumstances have to do with *variables* in the program. For example, the program expression $x + 3$ is only well-typed if $x$ is a number. "$x$ is a number" is an example of a fact that would be contained in the context. Its well-typedness could be represented as $x{:}\mathsf{num} \vdash x + 3\ \mathsf{ok}$.

In the move-take game, we can include aspects of game state in our context, such as the location of the player and the adjacency mapping between rooms in the world. An example of a typing rule we might include to codify the "only present things are takeable" rule would be:

$$\frac{\Gamma \vdash \mathsf{playerIn}(R) \quad \Gamma \vdash \mathsf{at}(O, R)}{\Gamma \vdash \mathsf{take}\langle O\rangle\ \mathsf{ok}}$$

We then need to define a relation between concrete game states $G$ and abstract conditions on those states, $\Gamma$. We might write this relation $G \ : \ \Gamma$. After such rules are codified, we can refine the "game completeness" conjecture to handle only those utterances that are well-typed:

$$\begin{array}{ll} \forall G_1, \mathit{intent}. & (G_1 \ : \ \Gamma) \wedge (\Gamma \vdash \mathit{intent}\ \mathsf{ok}) \\ \Longrightarrow \exists G_2, \mathit{resp}. & \langle G_1; \mathit{intent}\rangle \longrightarrow \langle G_2; \mathit{resp}\rangle \end{array}$$

This is nearly what we want to know about our game mechanics. However, we want to apply this reasoning iteratively as the game progresses, so that we reason next about the player intention that leads from game state $G_2$ to another possibly different game state $G_3$; but what context for player intent describes state $G_2$?

For this reasoning to work, we generally need to update the original context $\Gamma$, possibly changing its assumptions, and creating $\Gamma'$. We write $\Gamma \subseteq \Gamma'$ to mean that $\Gamma'$ *succeeds* $\Gamma$ in a well-defined way. Given that state $G_1$ and *intent* agree about the context of assumptions $\Gamma$, we wish to show that there exists a successor context $\Gamma'$

that agrees with the new game state $G_2$:

$$\begin{array}{lll} & \forall G_1, \mathit{intent}. & (G_1 \ : \ \Gamma) \wedge (\Gamma \vdash \mathit{intent}\ \mathrm{ok}) \\ \Longrightarrow & \exists \Gamma', G_2, \mathit{resp}. & \Big( \langle G_1; \mathit{intent} \rangle \longrightarrow \langle G_2; \mathit{resp} \rangle \Big) \\ & & \wedge (G_2 : \Gamma') \wedge (\Gamma \subseteq \Gamma') \end{array}$$

This statement closely matches the usual statement of *progress* for programming languages with sound type systems.

## 7 PLAY TRACES AS STRAIGHT-LINE PROGRAMS

If we consider the analogy of game interfaces as programming languages, the natural question arises, what is a *program* written in this programming language? We want to at least consider individual, atomic player actions to be complete programs; the preceding text provides such an account. But typical programs are more than one line long—what does it mean to sequence multiple actions in a game language?

In a typical account of an imperative programming language, we introduce a sequencing operator ; where, if $c_1$ and $c_2$ are commands in the language, then $c_1; c_2$ is also a command. The operational semantics of such a command involves the composition of transformations on states $\sigma$:

$$\frac{\langle \sigma; c_1 \rangle \longrightarrow \sigma_1 \quad \langle \sigma_1; c_2 \rangle \longrightarrow \sigma_2}{\langle \sigma; (c_1; c_2) \rangle \longrightarrow \sigma_2}$$

However, interactive software makes this account more complicated by introducing the program response as a component. Instead of issuing arbitrary commands in sequence, the player may wait for a response or process responses in parallel with their decisions. In this respect, a player's "programming" activity more closely resembles something like live-coding than traditional program authoring. Execution of code happens alongside its authorship, interleaving the two activities. If we consider a round-trip through the game loop after each individual command issued, then what we arrive at is a notion of program that resembles a *play trace*: a log of player actions and game responses during a play session, e.g.

PLAYER: go north
GAME: failure
PLAYER: take flask
GAME: success
PLAYER: go south
GAME: success

Depending on the richness of our internal mechanics model, this play trace may contain useful information about changes in internal state related to the preconditions and effects of player actions. But the main important thing to note is that, despite the informal syntax used to present them here, these traces do not consist of strings of text entered directly by the player or added as log information by the game programmer—they are structured terms with abstract syntax that may be treated to the same formal techniques of interpretation and analysis as any program. And this syntax is at a high level of game-relevant interactions, not at the level of hardware inputs and engine code.

Researchers in academia and the games industry alike have recently been increasingly interested in play trace data for the sake of analytics, such as understanding how their players are interacting with different components of the game and responding to this information with updates that support player interest [8]. For the most part, this trace data is collected through telemetry or other indirect means, like game variable monitoring, after which it must be analyzed for meaning [3]. More recently, systems of structured trace terms that may be analyzed with logical queries have been proposed [16], identifying as a benefit an ability to support automated testing at the level of design intents. Our PL analogy supports this line of inquiry and warrants further comparison and collaboration.

## 8 PLAYER SKILLS AS GENERAL PROGRAMS

While considering "straight-line" traces may have some utility in player analytics, a more exciting prospect for formalizing game interactions as program constructs is the possibility of encoding *parameterized* sequences of actions that may carry out complex tasks. After all, games with rich player action languages afford modes of exploratory and creative play: consider item crafting in Minecraft [17], puzzle solving in Zork [2], or creating sustainable autonomous systems like a supply chain in Factorio [21], a farm in Stardew Valley [1], or a transit system in Mini Metro [5]. Each of these activities asks the player to understand a complex system and construct multi-step sequences of actions to accomplish specific tasks. From the player's perspective, these plans are constructed from higher-level activities, such as *growing a crop* or *constructing a new tool*, which themselves are constructed from the lower level game intent language.

A language, as we have formalized it, gives us the atomic pieces from which we can construct these sequences, like Lego bricks can be used to construct reconfigurable components of a house or spaceship. *Compositionality* in language design is the principle that we may understand the meaning and behavior of compound structures (e.g. sequences) in terms of the meaning and behavior of each of its pieces (e.g. actions), together with the meaning of how they are combined (e.g. carried out one after the other, or in parallel). In this section, we describe how we might make sense of *player skills* in terms of programs written in a more complex version of the player language.

Such programs might be integrated into a game's mechanics so that a player explicitly writes such programs, as in the BOTS game, an interactive programming tutor that asks players to write small imperative programs that direct an avatar within a virtual world [10] (see Figure 3), or Cube Composer[2], in which players write functional programs to solve puzzles. However, for now, we primarily intend this account of player skills as a conceptual tool.

### 8.1 Example: Stardew Valley

Our initial {move, take} example is too simple to craft really compelling examples of complex programs, so here we examine Stardew Valley and its game language for the sake of considering player skills. In Stardew Valley, the player has an inventory that permits varied interactions with the world, beginning with a number of tools for farming (axe, hoe, scythe, pickaxe) which do different things in contact with the resources in the surrounding environment; most include extracting some resource (wood, stone, fiber, and so on), which themselves enter the player's inventory and can be used

[2] http://david-peter.de/cube-composer/

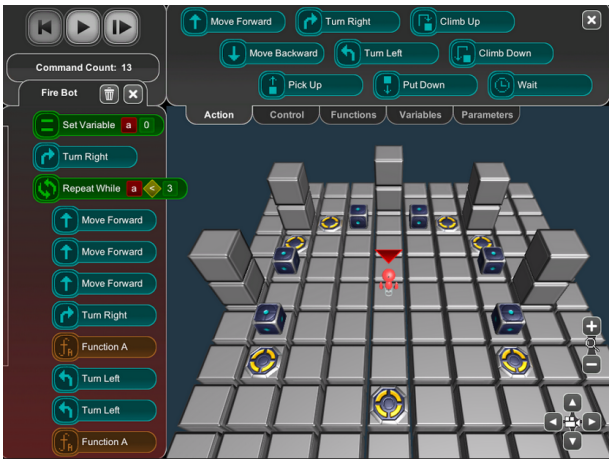


**Figure 3: A screenshot from BOTS, an educational game in which players write programs to direct a player avatar.**


**Figure 4: A screenshot from Stardew Valley showing the player's farm, inventory, and avatar.**

in further interaction with the game world. The player's avatar is shown on-screen, moved by WASD. There are also context-sensitive interactions between the player and non-player characters (NPCs), interfaces through which new items may be purchased (shops), and mini-games including fishing (fish may also be sold at high value). See Figure 4 for a typical player view of the game.

While a full account of the language that this game affords the player is beyond the scope of this paper, we include a representative sample of the actions and affordances found in this game that may be used to construct player skills. These include: directional avatar movement within and between world "rooms," point-and-click actions for selecting items in one's inventory, and interacting with in-room entities.

The player avatar must be near an entity for the player to interact with it. They can then either apply the currently-selected inventory item to the in-world entity with a left-button click, or right-button click, which does something based on the entity type, e.g. doors and chests open, characters speak, and collectible items transfer to the player's inventory. We refer to this last action as inquire. We also note that, for the sake of our example, movement towards an entity and movement offscreen (towards another room) are the only meaningful and distinct types of movement, which we refer to as move_near and move_offscreen. These actions yield the following syntax:

$$
\begin{aligned}
\textit{intent} \quad ::= \quad & \text{select}\langle \textit{item} \rangle \\
| \quad & \text{apply}\langle \textit{entity} \rangle \\
| \quad & \text{inquire}\langle \textit{entity} \rangle \\
| \quad & \text{move\_near}\langle \textit{entity} \rangle \\
| \quad & \text{move\_offscreen}\langle \textit{direction} \rangle
\end{aligned}
$$

We leave the definition of items and entities abstract, but we could imagine it to simply list all possible items and entities in the world as terminal symbols. From these atomic inputs, we can start to construct higher-level actions performed in the game most frequently—tilling land, planting seeds, conversing with NPCs, and so on. These blocks of code may be assigned names like functions to be called in many contexts:

```
action till  = select hoe; move_near hard_ground;
               apply hard_ground
action plant = select seeds; move_near tilled_ground;
               apply tilled_ground
action mine  = select pickaxe; move_near rock; apply rock
action talk  = move_near npc; inquire npc
action enter_shop = move_near shop; inquire door(shop)
```

In turn, these larger skill molecules may be combined to accomplish specific tasks or complete quests.

## 8.2 Branching programs as skills and strategies

Note that we have naively sequenced actions without consideration for the game's response. This approach to describing player skills does not take into account the possbility of a failed attempt, such as attempting to mine when there are no rocks on the current screen. One could simply assign a semantics to these sequences of action that threads failure through the program—if we fail on any action, the whole compound action fails.

However, we can go further with describing robust player skills and strategies if we consider the possibility of *handling* failure, a common feature of day-to-day programming and indeed of gameplay. Recall our simplified game response language consisting of two possibilities, success and failure. We can introduce a case construct into our language to handle each of these possibilities as a distinct branch of the program:

```
action mine =
  response = select pickaxe;
  case(response):
    success => {
      response' = move_near rock;
      case(response'):
          success => apply rock;
        | failure => fail;
    }
  | failure => fail;
```

However, to avoid handling failure at every possible action, a better approach is to explicitly indicate as parameters the world resources

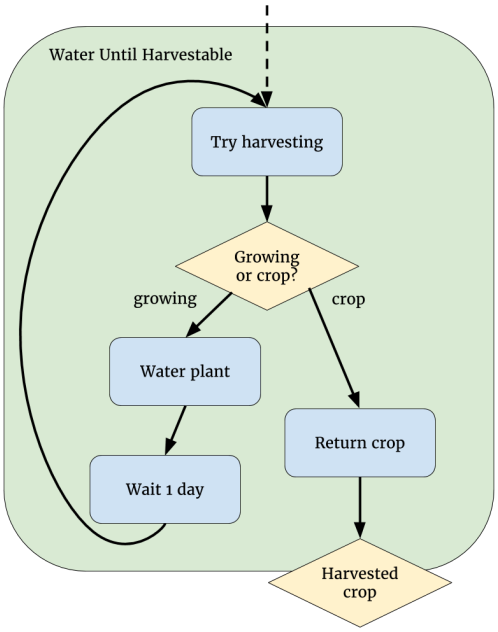


**Figure 5: Watering a crop until it may be harvested.**

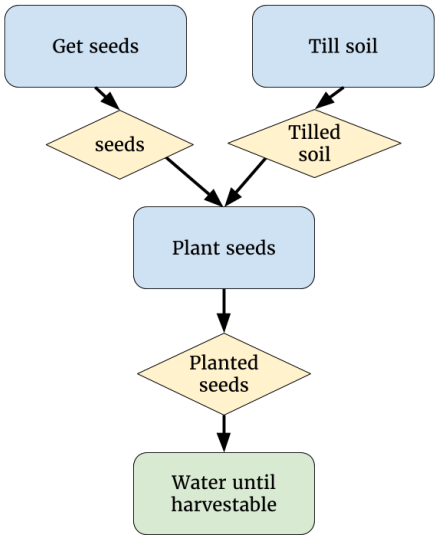


**Figure 6: Growing a crop.**

that each action needs in order to complete successfully. The overall action definition for mining, or example, would require as a precondition to the action that a pickaxe is available in the player inventory and a rock is in the same room. The actions for selecting the pickaxe and moving near the rock would depend on these resources, and the game response language could include the resources it guarantees as outputs. Then we can write the program using simpler notation that refers to resource dependencies of the appropriate type (using notation `resource : type`):

```
action mine(p:pickaxe, r:rock) =
  select p; move_near r; apply r
: mineral
```

This notation together with a branching `case` construct scales to include nondeterminism in the game world, such as the fishing minigame in Stardew Valley: the game always eventually tells the player that something is tugging at her line, but some portion of the time it is a fish while the rest of the time it is trash. These constructs can also account for incomplete player mental models, such as knowing that one must water her seeds repeatedly day after day in order to grow a crop, but not knowing how many times.

Below, we present a notation that accounts for these aspects of player skills: a `do ... recv ...` notation indicates a command and then binds the response to a *pattern*, or structured set of variables, which can then be case-analyzed. Our first example is watering a crop until it may be harvested:

```
action water_until_harvestable[t](p: planted(t))
: crop(t) =
  do try_harvest(p)
    recv <result: crop(t) + growing(t)>.
      case result of
        c:crop(t) => c
      | g:growing(t) =>
          water(g);
          wait(day);
          try_harvest(g)
```

See Figure 5 for a control flow diagram of this code.

The next example shows a parallel construct `||`, which can be used to compose actions with distinct dependencies, as well as how an action definition may use other action definitions by threading resource dependencies through as arguments:

```
fun grow_crop[t : croptype](s:soil, w:watering_can)
: crop(t) =
  do
    get_seeds(t) || till_soil(s)
  recv <s: seeds(t), g: tilled_soil>.
    do
      plant(s, g)
    recv <p: planted(t)>.water_until_harvestable(p)
```

See Figure 6 for a control flow diagram of this code.

## 9 DISCUSSION

Having established a vocabulary of syntax and semantics for player languages, we can now revisit the potential benefits of this account mentioned in the introduction and discuss them in more detail.

### 9.1 Composing play traces, player skills

One of the major things that a programming language account provides is *compositionality*: a system for making sense of meaning of a complex artifact in terms of the meaning of its pieces. This comes up in two places for looking at games:

**Structured play traces.** With a formalized game language, the sequence of steps along the transition system described by the operational semantics forms a mathematical artifact that is subject

to deeper analysis than what can be gained simply from screen or input device recordings. For example, we can carry out causal analysis, asking "why" queries of trace data, e.g. "Why was the player able to unlock the door before defeating the boss?", as well as filtering traces for desired properties: "Show me a play trace where the player used something other than the torch to light the room." The recent PlaySpecs project [16] suggests interest in formulating traces this way to support this kind of query.

**Player skills as programs.** While a play trace may be interpreted as a straight-line program, even more interesting is the idea of latent structure in player actions, such as composing multiple low-level game actions into a higher-level skill, following the cognitive idea of "chunking." We map this idea onto that of *functions* in the programming language that take arguments, generalizing over state space possibilities (e.g.: the red key opened the red door, so for all colors $C$, a $C$ key will open a $C$ door). Further reasoning forms like case analysis to handle unpredictable game behavior and repeating an action until a condition holds are also naturally expressed as programming language constructs.

### 9.2 Abstraction boundaries between input and mechanics

Formalizing game interfaces gives us the tools to explore alternative interfaces to the same underlying mechanics, without needing to port game logic between different graphical interface frameworks. For example, the interactive fiction community has been exploring alternatives to the traditional dichotomy of "parser vs. hypertext" for presenting text-based games and interactive story-worlds. An abstraction boundary between the underlying mechanics, map, and narrative of the world, and the view and input mechanisms used to interact with it, could open the doors to research on user interfaces that support players' mental models of a world conveyed in text.

### 9.3 Enabling co-creative play

Finally, a PL formulation of player actions along with appropriate composition operators (parallel and sequential composition, branching, and passing resource dependencies) provides a "scripting language for free" to the game environment. Such a language can be used to test the game, provided as a game mechanic as in BOTS, or provided as an optional augmentation to the game's mechanics for the sake of modding or adding new content to the game world. Especially in networked game environments, like multi-user domains, massively multiplayer online games, and social spaces like Second Life, the ability for the player to program not just her avatar but parts of the game world itself introduces new opportunities for creative and collaborative play. Our framework suggests a new approach to designing these affordances for players in a way that is naturally derived from the game's existing mechanics and interface. In the spirit of *celebrating the player*, this year's conference theme, we wish to enable the player as a co-designer of her own game experience.

### 9.4 Future Work

In future work, we intend to build software for realizing game language designs and experimenting with protocol-based game AI developed as programs in these languages. In another direction, we aim to innovate in PL design outside of games, such as read-eval-print loops (REPLs) for live programming that includes rapid feedback loops motivated by gameplay, as well as distributed and concurrent systems that may benefit from the protocol-based approach proposed here.

## 10 CONCLUSION

We propose *player intent languages*, a systematic framework for applying programming language design principles to the design of player-game interfaces. We define this framework through simple examples of syntax (aka player intents), type systems (aka contextual interfaces) and operational semantics (aka game mechanics). We show how applying this framework to a player-game interface naturally gives rise to formal notions of play traces (as straight-line programs) and player skills (as general programs with branching and recursion). By defining a player intent language, game design concepts become formal objects of study, allowing existing PL methodology to inform and guide the design process.

## REFERENCES


[1] Eric Barone. 2016. Stardew Valley. Digital distribution. (2016).
[2] Marc Blank and David Lebling. 1980. Zork I: The Great Underground Empire. (1980).
[3] Alessandro Canossa. 2013. *Meaning in Gameplay: Filtering Variables, Defining Metrics, Extracting Features and Creating Models for Gameplay Analysis.* Springer London, London, 255–283. DOI: http://dx.doi.org/10.1007/978-1-4471-4769-5_13
[4] Rogelio E Cardona-Rivera and R Michael Young. 2014. Games as conversation. In *Proceedings of the 3rd Workshop on Games and NLP at the 10th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* 2–8.
[5] Dinosaur Polo Club. 2015. Mini Metro. (2015).
[6] Chris Crawford. 2003. *Chris Crawford on game design.* New Riders.
[7] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. 2013. Towards a video game description language. (2013).
[8] M Seif El-Nasr, Anders Drachen, and Alessandro Canossa. 2013. Game analytics. *New York, Sprint* (2013).
[9] H Paul Grice. 1975. Logic and conversation. *1975* (1975), 41–58.
[10] Andrew Hicks. 2012. Creation, evaluation, and presentation of user-generated content in community game-based tutors. In *Proceedings of the International Conference on the Foundations of Digital Games.* ACM, 276–278.
[11] Kristine Jørgensen. 2013. *Gameworld interfaces.* MIT Press.
[12] Stein C Llanos and Kristine Jørgensen. 2011. Do players prefer integrated user interfaces? A qualitative study of game UI design issues.
[13] John McCarthy and Patrick J Hayes. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence* (1969), 431–450.
[14] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *POPL.*
[15] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *SNAPL: The 2nd Summit on Advances in Programming Languages.*
[16] J Osborn, Ben Samuel, Michael Mateas, and Noah Wardrip-Fruin. 2015. Playspecs: Regular expressions for game play traces. *Proceedings of AIIDE* (2015).
[17] Markus Persson, Jens Bergensten, Michael Kirkbride, Mark Darin, and Carl Muckenhoupt. 2011. Minecraft. (2011).
[18] Frank Pfenning and Carsten Schürmann. 1999. System description: Twelf–a meta-logical framework for deductive systems. In *International Conference on Automated Deduction.* Springer, 202–206.
[19] Tom Schaul. 2013. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on.* IEEE, 1–8.
[20] Adam M Smith, Mark J Nelson, and Michael Mateas. 2010. Ludocore: A logical game engine for modeling videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on.* IEEE, 91–98.
[21] Wube Software. 2012. Factorio. (2012).
[22] José P Zagal, Michael Mateas, Clara Fernández-Vara, Brian Hochhalter, and Nolan Lichti. 2007. Towards an ontological language for game analysis. *Worlds in Play: International Perspectives on Digital Games Research* 21 (2007), 21.